Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances

Anton Burtsev[†], Kiran Srinivasan, Prashanth Radhakrishnan,
Lakshmi N. Bairavasundaram, Kaladhar Voruganti, Garth R. Goodson
[†]*University of Utah* *NetApp, Inc.*
aburtsev@flux.utah.edu, {skiran, shanth, lakshmib, kaladhar, goodson}@netapp.com

Abstract

Enterprise-class server appliances such as network-attached storage systems or network routers can benefit greatly from virtualization technologies. However, current inter-VM communication techniques have significant performance overheads when employed between highly-collaborative appliance components, thereby limiting the use of virtualization in such systems. We present *Fido*, an inter-VM communication mechanism that leverages the inherent relaxed trust model between the software components in an appliance to achieve high performance. We have also developed common device abstractions - a network device (MMNet) and a block device (MMBlk) on top of Fido.

We evaluate MMNet and MMBlk using microbenchmarks and find that they outperform existing alternative mechanisms. As a case study, we have implemented a virtualized architecture for a network-attached storage system incorporating Fido, MMNet, and MMBlk. We use both microbenchmarks and TPC-C to evaluate our virtualized storage system architecture. In comparison to a monolithic architecture, the virtualized one exhibits nearly no performance penalty in our benchmarks, thus demonstrating the viability of virtualized enterprise server architectures that use Fido.

1 Introduction

Enterprise-class appliances [4, 21] are specialized devices providing services over the network to clients using standardized protocols. Typically, these appliances are built to deliver high-performance, scalable and highly-available access to the exported services. Examples of such appliances include storage systems (NetApp [21], IBM [14], EMC [8]), network-router systems (Cisco [4], Juniper [16]), etc. Placing the software components of such appliances in separate virtual machines (VMs) hosted on a hypervisor [1, 25] enables multiple benefits —fault isolation, performance isolation, effective resource utilization, load balancing via VM migration, etc. However, when collaborating components are encapsulated in VMs, the performance overheads introduced by current inter-VM communication mechanisms [1, 17, 26, 28] is prohibitive.

We present a new inter-VM communication mechanism called *Fido* specifically tailored towards the needs of an enterprise-class appliance. Fido leverages the relaxed trust model among the software components in an appliance architecture to achieve better performance. Specifically, Fido facilitates communication using read-only access between the address spaces of the component VMs. Through this approach, Fido avoids page-mapping and copy overheads while reducing expensive hypervisor transitions in the critical path of communication. Fido also enables end-to-end zero-copy communication across multiple VMs utilizing our novel technique called *Pseudo Global Virtual Address Space*. Fido presents a generic interface, amenable to the layering of other higher-level abstractions. In order to facilitate greater applicability of Fido, especially between components developed by different collaborating organizations, Fido is non-intrusive, transparent to applications and dynamically pluggable.

On top of Fido, we design two device abstractions, *MMNet* and *MMBlk*, to enable higher layers to leverage Fido. MMNet (Memory-Mapped Network) is a network device abstraction that enables high performance IP-based communication. Similarly, MMBlk is a block device abstraction. MMNet performs consistently better on microbenchmarks in comparison to other alternative mechanisms (XenLoop [26], Netfront [1] etc) and is very close in performance to a loopback network device interface. Likewise, MMBlk outperforms the equivalent open-source Xen hypervisor abstraction across several microbenchmarks.

As a case study, we design and implement a full-fledged virtualized network-attached storage system architecture that incorporates MMNet and MMBlk. Microbenchmark experiments reveal that our virtualized system does not suffer any degradation in throughput or latency in most test cases as compared to a monolithic storage server architecture. TPC-C macrobenchmark results reveal that the difference in performance between our architecture and the monolithic one is almost imperceptible.

To summarize, our contributions are:

- A high-performance inter-VM communication mechanism - Fido, geared towards software architectures of enterprise-class appliances.
- A technique to achieve end-to-end zero-copy communication across VMs - *Pseudo Global Virtual Address Space*.
- An efficient, scalable inter-VM infrastructure for connection management.
- Two high-performance device abstractions (MMNet and MMBlk) to facilitate higher level software to leverage the benefits of Fido.
- A demonstration of the viability of a modular virtualized storage system architecture utilizing Fido.

The rest of the paper is organized as follows. In Section 2, we present the background and the motivation for our work. Section 3 discusses the design and implementation of Fido and the abstractions - MMNet and MMBlk. Next, we evaluate Fido and the abstractions using standard storage benchmarks in Section 4. A case study of a network attached storage system utilizing Fido is presented in Section 5. In Section 6, we discuss related work. Finally, in Section 7 we present our conclusions.

2 Background and Motivation

In this section, we first provide an overview of appliance architectures and the benefits of incorporating virtualization in them. Next, we present the performance issues in such virtualized architectures, followed by a description of existing inter-VM communication mechanisms and their inadequacy in solving performance issues.

2.1 Enterprise-class Appliance Architectures

We are primarily concerned about the requirements and applicability of virtualization technologies to enterprise-class server appliances. Typically, these appliances provide a specialized service over the network using standardized protocols. High-performance access and high-availability of the exported network services are critical concerns.

Enterprise appliances have some unique features that differentiate them from other realms in which virtualization technologies have been adopted aggresively. In particular, the software components in such an architecture are extremely collaborative in nature with a large amount of data motion between them. This data flow is often organized in the form of a pipeline. An example of an enterprise appliance is a network-attached storage system [21, 27] providing storage services over standardized protocols, such as NFS and CIFS. Such a storage system consists of components such as a protocol server, a local file system, software RAID, etc. that operate as a pipeline for data.

2.2 Virtualization Benefits for Appliances

Virtualization technologies have been highly successful in the commodity servers realm. The benefits that have made virtualization technologies popular in the commodity server markets are applicable to enterprise-class server appliances as well:

- **High availability:** Components in an enterprise appliance may experience faults that lead to expensive disruption of service. Virtualization provides fault isolation across components placed in separate VM containers, thereby enabling options such as micro-reboots [2] for fast restoration of service, leading to higher availability.
- **Performance isolation/Resource allocation:** Virtualization allows stricter partitioning of hardware resources for performance isolation between VMs. In addition, the ability to virtualize resources as well as to migrate entire VMs enables the opportunity to dynamically provide (or take away) additional resources to overloaded (or underloaded) sections of the component pipeline, thus improving the performance of the appliance as a whole.
- **Non-disruptive upgrades:** Often, one needs to upgrade the hardware or software of enterprise systems with little or no disruption in service. The different software components of an appliance can be migrated across physical machines through transparent VM migration, thereby enabling non-disruptive hardware upgrades. The mechanisms that enable higher availability can be leveraged for non-disruptive software upgrades.

Such benefits have prompted enterprise-appliance makers to include virtualization technologies in their systems. The IBM DS8000 series storage system [7] is an example of an appliance that incorporates a hypervisor, albeit in a limited fashion, to host two virtual fault-isolated and performance-isolated storage systems on the same physical hardware. Separation of production and test environments, and flexibility of resource allocation are cited as reasons for incorporating virtualization [7].

2.3 Performance issues with virtualization

Encapsulating the software components of an appliance in VMs introduces new performance issues. First, device access may be considerably slower in a virtualized environment. Second, data transfer between components that used to happen via inexpensive function calls now crosses protected VM boundaries; since such data transfer is critical to overall performance, it is important that the inter-VM communication between the component VMs be optimized. The first issue is often easily solved in appliances, as devices can be dedicated to components. We address the second performance issue in this paper.

2.4 Inter-VM communication mechanisms

Current inter-VM communication mechanisms rely on either copying (XenLoop [26], XenSocket [28]) or page mapping/unmapping (Netfront [1]) techniques. Both of these techniques incur performance overheads in the critical data path, making them unsuitable for data-traffic intensive server appliances like storage systems. Moreover, the data throughput and latency results obtained with these mechanisms do not satisfy the requirements of an appliance. From another perspective, some of these mechanisms [26, 28] are designed for a specific kind of data traffic - network packets. In addition, they do not offer the flexibility of layering other types of data traffic on top of them. Thereby, restricting the applicability of their solution between different kinds of components in an appliance. All these reasons made us conclude that we need a specialized high-performance inter-VM communication mechanism. Moreover, since multiple component VMs process data in a pipeline fashion, it is not sufficient to have efficient pair wise inter-VM communication; we require efficient end-to-end transitive inter-VM communication.

3 Design and Implementation

In this section, we first describe the design goals of Fido, followed by the inherent trust model that forms the key enabler of our communication mechanism. We then present Fido, our fast inter-VM communication mechanism. Finally, we describe MMNet and MMBlk, the networking and disk access interfaces that build on the communication abstraction provided by Fido.

3.1 Design Goals

The following are the design goals of Fido to enable greater applicability as well as ease of use:

- **High Performance:** Fido should enable high throughput, low latency communication with acceptable CPU consumption.
- **Dynamically Pluggable:** Introduction or removal of Fido should not require a reboot of the system. This enables component VMs to leverage Fido without entailing an interruption in service.
- **Non-intrusive:** In order to limit the exposure of kernel data structures Fido should be built in a non-intrusive fashion. The fewer the dependencies with other kernel data structures, the easier it is to port across kernel versions.
- **Application-level transparent:** Leveraging Fido should not require applications to change. This ensures that existing applications can start enjoying the performance benefits of Fido without requiring code-level changes.
- **Flexible:** Fido should enable different types of data transfer mechanisms to be layered on top of it with minimal dependencies and a clean interface.

Specifically, being non-intrusive, dynamically pluggable and application transparent extends Fido's applicability in appliances where the components might be independently developed by collaborating organizations.

3.2 Relaxed Trust Model

Enterprise-class server appliances consist of various software components that are either mostly built by a single organization or put together from pre-tested and qualified components. As a result, the degree of trust between components is significantly more than in typical applications of virtualization. In fact, the various components collaborate extensively and readily exchange or release resources for use by other components. At the same time, in spite of best efforts, the various components may contain bugs that create a need for isolating them from each other.

In an enterprise server appliance, the following trust assumptions apply. First, the different software components in VMs are assumed to be non-malicious. Therefore, read-only access to each other's address spaces is acceptable. Second, most bugs and corruptions are assumed to lead to crashes sooner than later; enterprise appliances are typically designed to fail-fast; as well, it has been shown that Linux systems often crash within 10 cycles of fault manifestation [10]. Therefore, the likelihood of corruptions propagating from a faulty VM to a communicating VM via read-only access of memory is low. However, VMs are necessary to isolate components from crashes in each other.

3.3 Fido

Fido is an inter-VM shared-memory-based communication mechanism that leverages the relaxed trust model to improve data transfer speed. In particular, we design Fido with a goal of reducing the primary contributors to inter-VM communication overheads: hypervisor transitions and data copies. In fact, Fido enables zero-copy data transfer across multiple virtual machines on the same physical system.

Like other inter-VM communication mechanisms that leverage shared memory, Fido consists of the following features: (i) a shared-memory mapping mechanism, (ii) a signaling mechanism for cross-VM synchronization, and (iii) a connection-handling mechanism that fa-

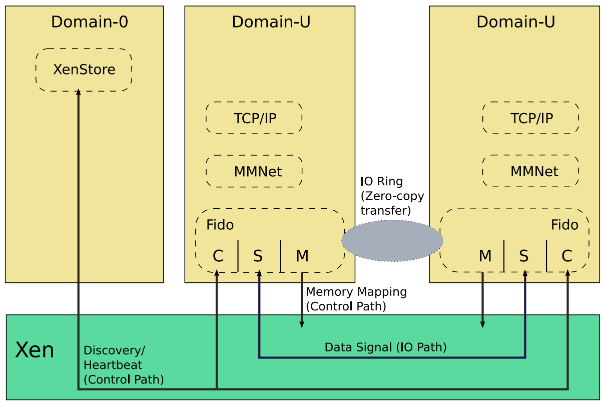

Figure 1: Fido Architecture. *The figure shows the components of Fido in a Linux VM over Xen. The two domUs contain the collaborating software components. In this case, they use MMNet and Fido to communicate. Fido consists of three primary components: a memory-mapping module (M), a connection module (C), and a signaling module (S). The connection module uses XenStore (a centralized key-value store in dom0) to discover other Fido-enabled VMs, maintain its own membership, and track VM failures. The memory-mapping module uses Xen grant reference hypervisor calls to enable read-only memory mapping across VMs. It also performs zero-copy data transfer with a communicating VM using I/O rings. The signaling module informs communicating VMs about availability and use of data through the Xen signal infrastructure.*

cilitates set-up, teardown, and maintenance of shared-memory state. Implementation of these features requires the use of specific para-virtualized hypervisor calls. As outlined in the following subsections, the functionality expected from these API calls is simple and is available in most hypervisors (Xen, VMWare ESX, etc.).

Fido improves performance through simple changes to the shared-memory mapping mechanism as compared to traditional inter-VM communication systems. These changes are complemented by corresponding changes to connection handling, especially for dealing with virtual-machine failures. Figure 1 shows the architecture of Fido. We have implemented Fido for a Linux VM on top of the Xen hypervisor. However, from a design perspective, we do not depend on any Xen-specific features; Fido can be easily ported to other hypervisors. We now describe the specific features of Fido.

3.3.1 Memory Mapping

In the context of enterprise-class appliance component VMs, Fido can exploit the following key trends: (i) the virtual machines are not malicious to each other and hence each VM can be allowed read-only access to the entire address space of the communicating VM and (ii) most systems today use 64-bit addressing, but individual virtual machines have little need for as big an address space due to limitations on physical memory size. Therefore, with Fido, the entire address space of a *source* virtual machine is mapped read-only into the *destination* virtual machine, where source and destination refer to the direction of data transfer. This mapping is established *a priori*, before any data transfer is initiated. As a result, the data transfer is limited only by the total physical memory allocated to the source virtual machine, thus avoiding limits to throughput scaling due to small shared-memory segments. Other systems [9, 26] suffer from these limits, thereby causing either expensive hypervisor calls and page table updates [20] or data copies to and from the shared segment when the data is not produced in the shared-memory segment [17, 28].

In order to implement this memory mapping technique, we have used the grant reference functionality provided by the Xen hypervisor. In VMWare ESX, the functional equivalent would be the hypervisor calls leveraged by the VMCI (Virtual Machine Communication Interface [24]) module. To provide memory mapping, we have not modified any guest VM (Linux) kernel data structures. Thus, we achieve one of our design goals of being non-intrusive to the guest kernel.

3.3.2 Signaling Mechanism

Like other shared-memory based implementations, Fido needs a mechanism to send signals between communicating entities to notify data availability. Typically, hypervisors (Xen, VMWare, etc.) support hypervisor calls that enable asynchronous notification between VMs. Fido adopts the Xen signaling mechanism [9] for this purpose. This mechanism amortizes the cost of signaling by collecting several data transfer operations and then issuing one signaling call for all operations. Again, this bunching together of several operations is easier with Fido since the shared memory segment is not limited. Moreover, after adding a bunch of data transfer operations, the source VM signals the destination VM only when it has picked up the previous signal from the source VM. In case the destination VM has not picked up the previous signal, it is assumed that it would pick up the newly queued operations while processing the previously enqueued ones.

3.3.3 Connection Handling

Connection handling includes connection establishment, connection monitoring and connection error handling between peer VMs.

Connection State: A Fido connection between a pair of VMs consists of a shared memory segment (*metadata segment*) and a Xen event channel for signaling between the VMs. The metadata segment contains shared data structures to implement producer-consumer rings (*I/O rings*) to facilitate exchanging of data between VMs (similar to Xen I/O rings [1]).

Connection Establishment: In order to establish an inter-VM connection between two VMs, the Fido module in each VM is initially given the identity (Virtual Machine ID - *vmid*) of the peer VM. One of the communicating VMs (for example, the one with the lower vmid) initiates the connection establishment process. This involves creating and sharing a metadata segment with the peer. Fido requires a centralized key-value DB that facilitates proper synchronization between the VMs during the connection setup phase. Operations on the DB are not performance critical, they are performed only during setup time, over-the-network access to a generic transactional DB would suffice. In Xen, we leverage XenStore—a centralized hierarchical DB in Dom0—for transferring information about metadata segment pages via an asynchronous, non-blocking handshake mechanism. Since Fido leverages a centralized DB to exchange metadata segment information, it enables communicating VMs to establish connections dynamically. Therefore, by design, Fido is made *dynamically pluggable*.

From an implementation perspective, Fido is implemented as a loadable kernel module, and the communication with XenStore happens at the time of loading the kernel module. Once the metadata segment has been established between the VMs using XenStore, we use the I/O rings in the segment to bootstrap memory-mapping. This technique avoids the more heavy-weight and circuitous XenStore path for mapping the rest of the memory read-only. The source VM's memory is mapped into the paged region of the destination VM in order to facilitate zero-copy data transfer to devices (since devices do not interact with data in non-paged memory). To create such a mapping in a paged region, the destination VM needs corresponding `page` structures. We therefore pass the appropriate kernel argument `mem` at boot time to allocate enough `page` structures for the mappings to be introduced later. Note that Linux's memory-hotplug feature allows dynamic creation of `page` structures, thus avoiding the need for a boot-time argument; however, this feature is not fully-functional in Xen para-virtualized Linux kernels.

Connection Monitoring: The Fido module periodically does a heartbeat check with all the VMs to which it is connected. We again leverage XenStore for this heartbeat functionality. If any of the connected VMs is missing, the connection failure handling process is triggered.

Connection Failure Handling: Fido reports errors detected during the heartbeat check to higher-level layers. Upon a VM's failure, its memory pages that are mapped by the communicating VMs cannot be deallocated until all the communicating VMs have explicitly unmapped those pages. This ensures that after a VM's failure, the immediate accesses done by a communicating VM will not result in access violations. Fortunately, this is guaranteed by Xen's inter-VM page sharing mechanism.

Data Transfer: This subsection describes how higher layer subsystems can use Fido to achieve zero-copy data transfer.

- **Data Representation:** Data transferred over the Fido connection is represented as an array of pointers, referred to as the scatter-gather (SG) list. Each I/O ring entry contains a pointer to an SG list in the physical memory of the source VM and a count of entries in the SG list. The SG list points to data buffers allocated in the memory of the source VM.
- **IO Path:** In the send data path, every request originated from a higher layer subsystem (i.e., a client of Fido) in the source guest OS is expected to be in an SG list and sent to the Fido layer. The SG list is sent to the destination guest OS over the I/O ring. In the receive path, the SG list will be picked up by the Fido layer and passed up to the appropriate higher layer subsystem, which in turn will package it into a request suitable for delivery to the destination OS. Effectively, the SG list is the generic data structure that enables different higher layer protocols to interact with Fido without compromising the zero-copy advantage.
- **Pointer Swizzling:** A source VM's memory pages are mapped at an arbitrary offset in the kernel address space of the destination VM. As a result, the pointer to the SG list and the data pointers in the SG list provided by the source VM are incomprehensible when used as-is by the destination VM. They need to be translated relative to the offset where the VM memory is mapped in. While the translation can be done either by the sender or the receiver, we chose to do it in the sender. Doing the translation in the sender simplifies the design of transitive zero-copy (Section 3.3.4).

3.3.4 Transitive Zero-Copy

As explained in Section 2, data flows through an enterprise-class software architecture successively across the different components in a pipeline. To ensure high performance we need true end-to-end zero-copy. In Section 3.3.1, we discussed how to achieve zero-copy between two VMs. In this section, we address the challenges involved in extending the zero-copy transitively across multiple component VMs.

Translation problems with transitive zero-copy: In order to achieve end-to-end zero-copy, data originating in a *source component VM* must be accessible and comprehensible in downstream component VMs. We ensure accessibility of data by mapping the memory of the source component VM in every downstream component VM with read permissions. For data to be comprehensi-

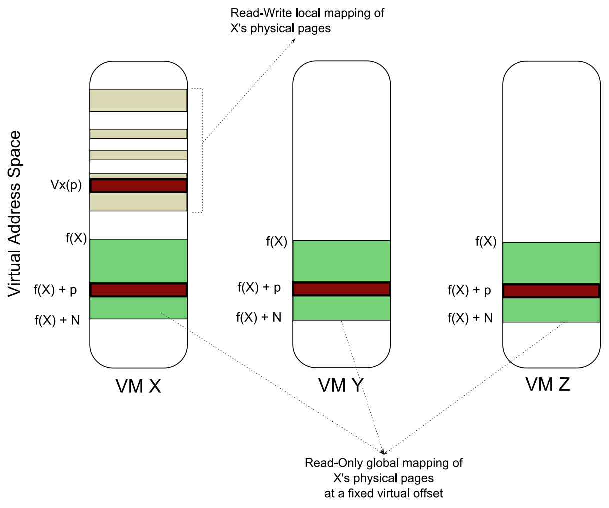
Read-Write local mapping of X's physical pages
Virtual Address Space
Vx(p)
f(X)
f(X) + p
f(X) + N
VM X
VM Y
VM Z
Read-Only global mapping of X's physical pages at a fixed virtual offset

Figure 2: *PGVAS technique between VMs X, Y and Z.*

ble in a downstream component VM, all data references that are resolvable in the source VM's virtual address space will have to be translated correctly in the destination VM's address space. Doing this translation in each downstream VM can prove expensive.

Pseudo Global Virtual Address Space: The advent of 64-bit processor architecture makes it feasible to have a global virtual address space [3, 11] across all component VMs. As a result, all data references generated by a source VM will be comprehensible in all downstream VMs; thus eliminating all address translations.

The global address space systems (like Opal[3]) have a single shared page table across all protected address spaces. Modifying the traditional guest OS kernels to use such a single shared page table is a gargantuan undertaking. We observe that we can achieve the effect of a global virtual address space if each VM's virtual address space ranges are distinct and disjoint from each other. Incorporating such a scheme may also require intrusive changes to the guest OS kernel. For example, Linux will have to be modified to map its kernel at arbitrary virtual address offsets, rather than from a known fixed offset.

We develop a hybrid approach called *Pseudo Global Virtual Address Space* (PGVAS) that enables us to leverage the benefits of a global virtual address space without the need for intrusive changes to the guest OS kernel. We assume that the virtual address spaces in the participating VMs are 64-bit virtual address spaces; thus the kernel virtual address space should have sufficient space to map the physical memory of a large number of co-located VMs. Figure 2 illustrates the PGVAS technique. With PGVAS, there are two kinds of virtual address mappings in a VM, say *X*. *Local mapping* refers to the traditional way of mapping the physical pages of *X* by its guest OS, starting from virtual address zero. In addition, there is a *global mapping* of the physical pages of *X* at a virtual offset derived from X's id, say *f(X)*. An identical global mapping exists at the same offset in the virtual address spaces of all communicating VMs. In our design, we assume VM ids are monotonically increasing, leading to `f(X) = M*X + base`, where `M` is the maximum size of a VM's memory, `X` is X's id and `base` is the fixed starting offset in the virtual address spaces.

To illustrate the benefits, consider a transitive data transfer scenario starting from VM X, leading to VM Y and eventually to VM Z. Let us assume that the transferred data contains a pointer to a data item located at physical address p in X. This pointer will typically be a virtual reference, say V_x*(p)*, in the local mapping of X, and thus, incomprehensible in Y and Z. Before transferring the data to Y, X will encode p to a virtual reference, $f(X)+p$, in the global mapping. Since global mappings are identical in all VMs, Y and Z can dereference the pointer directly, saving the cost of multiple translations and avoiding the loss of transparency of data access in Y and Z. As a result, all data references have to be translated once by the source VM based on the single unique offset where its memory will be mapped in the virtual address space of every other VM. This is also the rationale for having the sender VM do the translations of references in Fido as explained in Section 3.3.1.

3.4 MMNet

MMNet connects two VMs at the network link layer. It exports a standard network device interface to the guest OS. In this respect, MMNet is very similar to Xen NetBack/NetFront drivers. However, it is layered over Fido and has been designed with the key goal of preserving the zero-copy advantage that Fido provides.

MMNet exports all of the key Fido design goals to higher-layer software. Since MMNet is designed as a network device driver, it uses clean and well-defined interfaces provided by the kernel, ensuring that MMNet is totally *non-intrusive* to the rest of the kernel. MMNet is implemented as a loadable kernel module. During loading of the module, after the MMNet interface is created, a route entry is added in the routing table to route packets destined to the communicating VM via the MMNet interface. Packets originating from applications dynamically start using MMNet/Fido to communicate with their peers in other VMs, satisfying the *dynamic pluggability* requirement. This seamless transition is completely *transparent to the applications* requiring no application-level restarts or modifications.

MMNet has to package the Linux network packet data structure `skb` into the OS-agnostic data-structures of Fido and vice-versa, in a zero-copy fashion. The `skb` structure allows for data to be represented in a linear data buffer and in the form of a non-linear scatter-gather list of buffers. Starting with this data, we create a Fidocompatible SG list (Section 3.3.3) containing pointers to the `skb` data. Fido ensures that this data is transmitted to the communicating VM via the producer-consumer I/O rings in the metadata segment.

On the receive path, an asynchronous signal triggers Fido to pull the SG list and pass it to the corresponding MMNet module. The MMNet module in turn allocates a new `skb` structure with a custom destructor function and adds the packet data from the SG onto the non-linear part of the `skb` without requiring a copy. Once the data is copied from kernel buffers onto the user-space, the destructor function on the skb is triggered. The `skb` destructor function removes the data pointers from the non-linear list of the `skb` and requests Fido to notify the source VM regarding completion of packet data usage.

Though MMNet appears as a network device, it is not constrained by certain hardware limitations like the MTU of traditional network devices and can perform optimizations in this regard. MMNet presents an MTU of 64KB (maximum TCP segment size) to enable high performance network communication. In addition, since MMNet is used exclusively within a physical machine, MMNet can optionally disable checksumming by higher protocol layers, thereby reducing network processing costs.

3.5 MMBlk

MMBlk implements block level connection between virtual machines. Conceptually MMBlk is similar to Xen's BlkBack/BlkFront block device driver [1]. However, like MMNet, it is layered on top of the Fido

We implement MMBlk as a split block device driver for the Linux kernel. In accordance to a block device interface, MMBlk receives read and write requests from the kernel in the `bio` structure. `bio` provides a description of read/write operations to be done by the device along with an array of pages containing data.

MMBlk write path can be trivially implemented with no modifications to the Linux code. Communicating VMs share their memory in a read-only manner. Thus, a writer VM only needs to send pointers to the `bio` pages containing write data. Then, the communicating VM on the other end can either access written data or in the case of a device driver VM, it can perform a DMA straight from the writer's pages. Note, that in order to perform DMA, the `bio` page has to be accessible by the DMA engine. This comes with no additional data copy on a hardware providing an IOMMU. An IOMMU enables secure access to devices by enabling use of virtual addresses by VMs. Without an IOMMU, we rely on the `swiotlb` Xen mechanism implementing IOMMU translation in software. `swiotlb` keeps a pool of low memory pages, which are used for DMA. When translation is needed, `swiotlb` copies data into this pool.

Unfortunately, implementation of a zero-copy read path is not possible without intrusive changes to the Linux storage subsystem. The problem arises from the fact that on the read path, pages into which data has to be read are allocated by the reader, i.e., by an upper layer, which creates the `bio` structure before passing it to the block device driver. These pages are available read-only to the block device driver domain and hence cannot be written into directly. There are at least three ways to handle this problem without violating fault-isolation between the domains. First, the driver VM can allocate a new set of pages to do the read from the disk and later pass it to the reader domain as part of the response to the read request. The reader then has to copy the data from these pages to the original destination, incurring copy costs in the critical path. The second option is to make an intrusive change to the Linux storage subsystem whereby the `bio` structure used for the read contains an extra level of indirection, i.e., pointers to pointers of the original buffers. Once the read data is received in freshly allocated pages from the driver VM, the appropriate pointers can be fixed to ensure that data is transferred in a zero-copy fashion. The third option is similar to the first one, instead of copying we can perform page-flipping to achieve the same goal. We performed a microbenchmark to compare the performance of copying versus page-flipping and observed that page-flipping outperforms copying for larger data transfers (greater than 4K bytes). We chose the first option for our implementation, experimenting with page-flipping is part of future work.

4 Evaluation

In this section, we evaluate the performance of MMNet and MMBlk mechanisms with industry-standard microbenchmarks.

4.1 System Configuration

Our experiments are performed on a machine equipped with two quad-core 2.1 GHz AMD Opteron processors, 16 GB of RAM, three NVidia SATA controllers and two NVidia 1 Gbps NICs. The machine is configured with three additional (besides the root disk) Hitachi Deskstar E7K500 500GB SATA disks with a 16 MB buffer, 7200 RPM and a sustained data transfer rate of 64.8 MB/s. We use a 64-bit Xen hypervisor (version 3.2) and a 64-bit Linux kernel (version 2.6.18.8).

4.2 MMNet Evaluation

We use the `netperf` benchmark (version 2.4.4) to evaluate MMNet. `netperf` is a simple client-server based user-space application, which includes tests for measuring uni-direction bandwidth (STREAM tests) and end-to-end latency (RR tests) over TCP and UDP.

We compare MMNet with three other implementations: i) *Loop:* the loopback network driver in a sin-

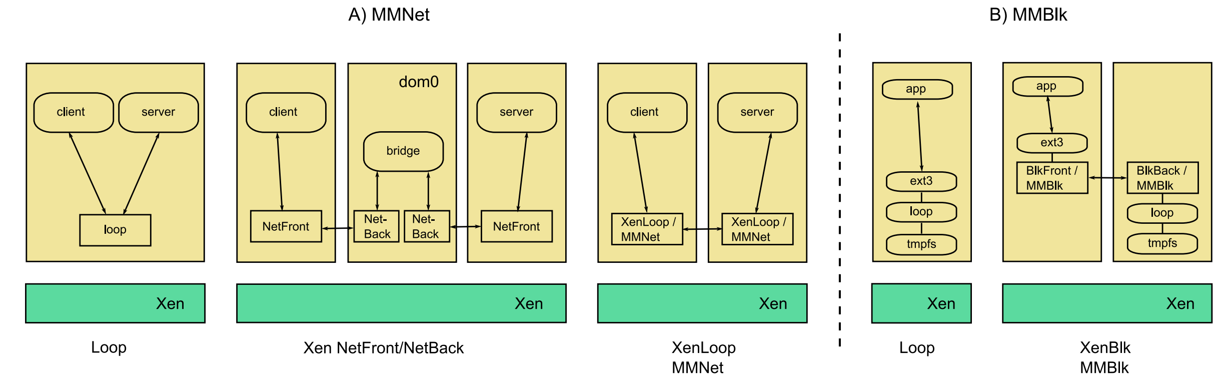

Figure 3: *MMNet and MMBlk Evaluation Configurations*

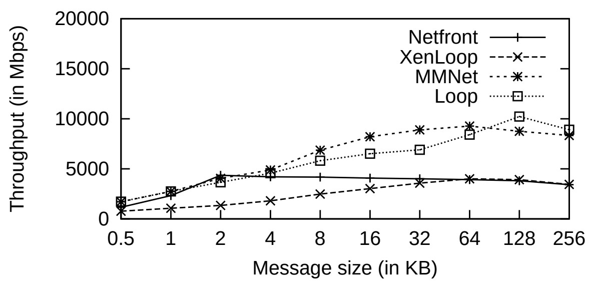

Figure 4: *TCP Throughput (TCP_STREAM test)*

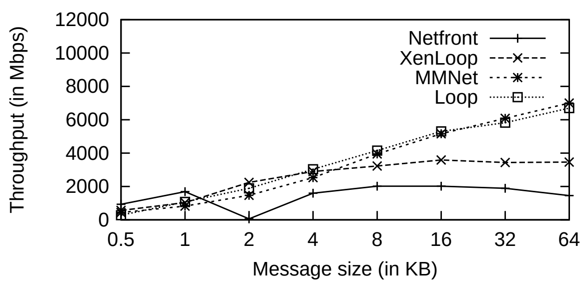

Figure 5: *UDP Throughput (UDP_STREAM test)*

gle VM for baseline; ii) *Netfront:* the default Xen networking mechanism that routes all traffic between two co-located VMs (*domUs*) through a third management VM (*dom0*), which includes a backend network driver; iii) *XenLoop [26]:* an inter-VM communication mechanism that, like MMNet, achieves direct communication between two co-located domUs without going through dom0. These configurations are shown in the Figure 3A.

Unlike MMNet, the other implementations have additional copy or page remapping overheads in the I/O path, as described below:

- **Netfront:** In the path from the sender domU to dom0, dom0 temporarily maps the sender domU's pages. In the path from dom0 to the receiver domU, either a copy or *page-flipping* [1] is performed. In our tests we use page-flipping, which is the default mode.

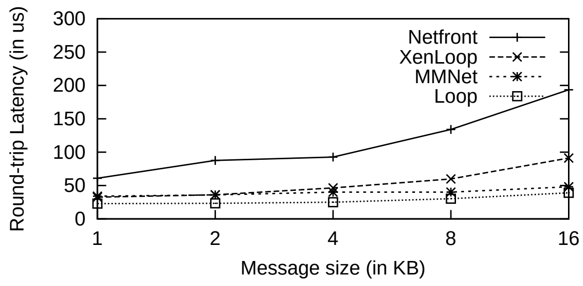

Figure 6: *TCP Latency (TCP_RR test)*

- **XenLoop:** A fixed region of memory is shared between the two communicating domUs. In the I/O path, XenLoop copies data in and out of the shared region.

All VMs are configured with one virtual CPU each. The only exception is the VM in the *loop* experiment, which is configured with two virtual CPUs. Virtual CPUs were pinned to separate physical cores, all on the same quad-core CPU socket. All reported numbers are averages from three trials.

Figure 4 presents TCP throughput results for varying message sizes. The figure shows that MMNet performs significantly better than XenLoop and the default Xen drivers, reaching a peak throughput of 9558 Mb/s at a message size of 64KB.

We see that performance with XenLoop is worse than Netfront. Given that XenLoop was designed to be more efficient than Netfront, this result seems contradictory. We found that the results reported by the XenLoop authors [26] were from tests performed on a single socket, dual-core machine. The three VMs, namely the two domUs and dom0, were sharing two processor cores amongst themselves. In contrast, our tests had dedicated cores for the VMs. This reduces the number of VM switches and helps Netfront better pipeline activity (such as copies and page-flips) over three VMs. In order to verify this hypothesis, we repeated the `netperf` TCP_STREAM experiment (with a 16KB message size) by restricting all the three VMs to two CPU cores and found that XenLoop (4000 Mbps) outperforms Netfront (2500 Mbps).

UDP throughput results for varying message sizes are shown in Figure 5. We see that the MMNet performance is very similar to Loop and significantly better than Netfront and XenLoop. Inter-core cache hits could be the reason for this observation, since UDP protocol processing times are shorter compared to TCP, it could lead to better inter-core cache hits. This will benefit data copies done across cores (for example, in XenLoop, the receiver VM's copy from the shared region to the kernel buffer will be benefited). There will be no benefit for Netfront because it does page remapping as explained earlier.

Figure 6 presents the TCP latency results for varying request sizes. MMNet is almost four times better than Netfront. Moreover, MMNet latencies are comparable to XenLoop for smaller message sizes. However, as the message sizes increase, the additional copies that XenLoop incurs hurt latency and hence, MMNet outperforms XenLoop. Netfront has the worst latency results because of the additional dom0 hop in the network path.

4.3 MMBlk Evaluation

We compare the throughput and latency of MMBlk driver with two other block driver implementations: i) *Loop:* the monolithic block layer implementation where the components share a single kernel space; ii) *XenBlk:* a split architecture where the block layer spans two VMs connected via the default Xen block device drivers. These configurations are illustrated in Figure 3B.

To eliminate the disk bottleneck, we create a block device (using *loop* driver) on TMPFS. In the Loop setup, an ext3 file system is directly created on this device. In the other setups, the block device is created in one (*backend*) VM and exported via the XenBlk/MMBlk mechanisms to another (*frontend*) VM. The frontend VM creates an ext3 file system on the exported block device. The backend and the frontend VMs were configured with 4 GB and 1 GB of memory, respectively. The in-memory block device is 3 GB in size and we use a 2.6 GB file in all tests.

Figure 7 presents the memory read and write throughput results for different block sizes measured using the IOZone [15] microbenchmark (version 3.303). For the Loop tests, we observe that the IOZone workload performs poorly. To investigate this issue, we profiled executions of all three setups. Compared to the split cases, execution of Loop has larger number of wait cycles. From our profile traces, we believe that the two filesystems (TMPFS and ext3) compete for memory – trying to allocate new pages. TMPFS is blocked as most of the memory is occupied by the buffer cache staging ext3's writes. To improve Loop's performance, we configure the monolithic system with 8GB of memory.

We consistently find that read throughput at a particular record size is better than the corresponding write throughput. This is due to soft page faults in TMPFS for new writes (writes to previously unwritten blocks).

From Figure 7A, we see that MMBlk writes perform better than XenBlk writes by 39%. This is because XenBlk incurs page remapping costs in the write path, while MMBlk does not. Further, due to inefficiencies in Loop, on average MMBlk is faster by 45%. In the case of reads, as shown in Figure 7B, XenBlk is only 0.4% slower than the monolithic Loop case. On smaller record sizes, Loop outperforms XenBlk due to a cheaper local calls. On larger record sizes, XenBlk becomes faster leveraging the potential to batch requests and better pipeline execution. XenBlk outperforms MMBlk by 35%. In the read path, MMBlk does an additional copy, whereas XenBlk does page remapping. Eliminating the copy (or page flip) in the MMBlk read path is part of future work.

5 Case Study: Virtualized Storage System Architecture

Commercial storage systems [8, 14, 21] are an important class of enterprise server appliances. In this case study, we examine inter-VM communication overheads in a virtualized storage-system architecture and explore the use of Fido to alleviate these overheads. We first describe the architecture of a typical network-attached storage system, then we outline a proposal to virtualize its architecture and finally, evaluate the performance of the virtualized architecture.

5.1 Storage System Architecture

The composition of the software stack of a storage system is highly vendor-specific. For our analysis, we use the NetApp software stack [27] as the reference system. Since all storage systems need to satisfy certain common customer requirements and have similar components and interfaces, we believe our analysis and insights are also applicable to other storage systems in the same class as our reference system.

The data flow in a typical monolithic storage system is structured as a pipeline of requests through a series of components. Network packets are received by the network component (e.g., network device driver). These packets are passed up the network stack for protocol processing (e.g., TCP/IP followed by NFS). The request is then transformed into a file system operation. The file system, in turn, translates the request into disk accesses and issues them to a software-RAID component. RAID converts the disk accesses it receives into one or more disk accesses (data and parity) to be issued to a storage component. The storage component, in turn, performs the actual disk operations. Once the data has been retrieved from or written to the disks, an appropriate re-

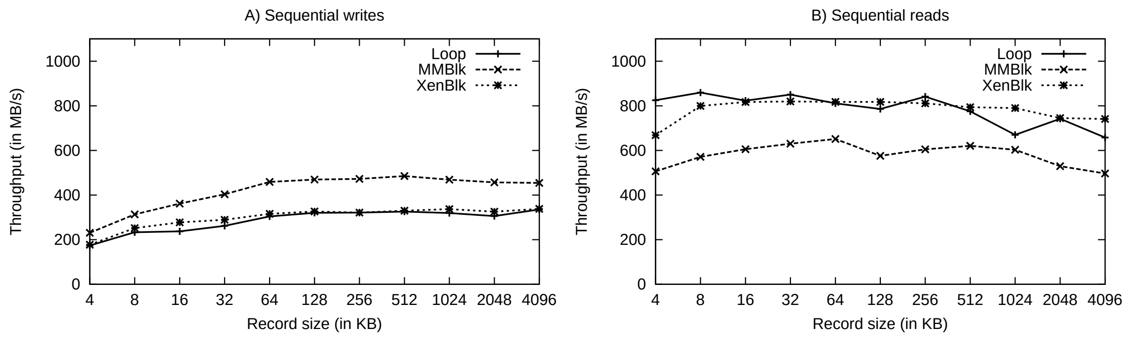

Figure 7: *MMBlk Throughput Results*

sponse is sent via the same path in reverse.

5.2 Virtualized Architecture

We design and implement a modular architecture for an enterprise-class storage system that leverages virtualization technologies. Software components are partitioned into separate VMs. For the purposes of understanding the impact of inter-VM communication in such an architecture as well as evaluating our mechanisms, we partition components as shown in Figure 8. While this architecture is a representative architecture, it might not necessarily be the ideal one from a modularization perspective. Identifying the ideal modular architecture merits a separate study and is outside the scope of our work.

Our architecture consists of four different component VMs — Network VM, Protocols and File system VM, RAID VM and Storage VM. Such an architecture can leverage many benefits from virtualization (Section 2.2):

- Virtualization provides much-needed fault isolation between components. In addition, the ability to reboot individual components independently greatly enhances the availability of the appliance.
- Significant performance isolation across file system volumes can be achieved by having multiple sets of File system, RAID, and Storage VMs, each set serving a different volume. One can also migrate one such set of VMs to a different physical machine for balancing load.
- Component independence helps with faster development and deployment of new features. For instance, changes to device drivers in the Storage VM (say to support new devices or fix bugs) can be deployed independently of other VMs. In fact, one might be able to upgrade components in a running system.

The data flow in the virtualized architecture starts from the Network VM, passes successively through the File system and RAID VMs and ends in the Storage VM, resembling a pipeline. This pipelined processing requires data to traverse several VM boundaries entailing inter-VM communication performance overheads. In order to ensure high end-to-end performance of the system, it is imperative that we address the inter-VM communication performance. As mentioned in Section 2, inter-VM communication performance is just one of the performance issues for this architecture; other issues like high-performance device access are outside our scope.

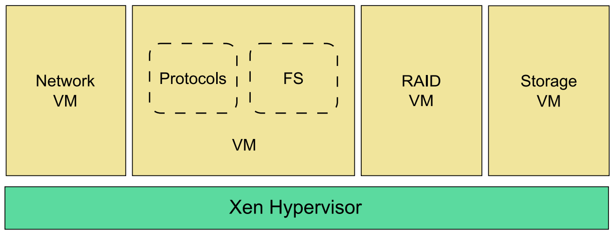

Figure 8: *Architecture with storage components in VMs*

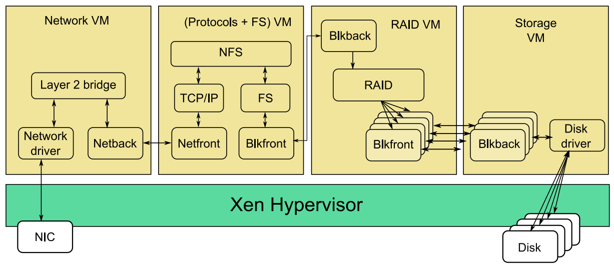

Figure 9: *Full system with MMNet and MMBlk*

5.3 System Implementation Overview

In Figure 9, we illustrate our full system implementation incorporating MMNet and MMBlk between the different components. Our prototype has four component VMs:

- **Network VM:** The network VM has access to the physical network interface. In addition, it has a MMNet network device to interface with the (Protocols + FS) VM. The two network interfaces are linked together by means of a Layer 2 software bridge.
- **Protocols + FS VM:** This VM is connected to the Network VM via the MMNet network device. We run an in-kernel NFSv3 server exporting an ext3 file system to network clients via the MMNet interface. The file system is laid out on a RAID device exported by the RAID VM via an MMBlk block device. This

VM is referred to as FS VM subsequently.

- **RAID VM:** This VM exports a RAID device to the FS VM using the MMBlk block device interface. We use the MD software RAID5 implementation available in Linux. The constituting data and parity disks are actually virtual disk devices exported again via MMBlk devices from the neighboring storage VM.
- **Storage VM:** The storage VM accesses physical disks, which are exported to the RAID VM via separate MMBlk block device interfaces.

The goal of our prototype is to evaluate the performance overheads in a virtualized storage system architecture. Therefore, we did not attempt to improve the performance of the base storage system components themselves by making intrusive changes. For example, the Linux implementation of the NFS server incurs two data copies in the critical path. The first copy is from the network buffers to the NFS buffers. This is followed by another copy from the NFS buffer to the Linux buffer cache. The implication of these data copies is that we cannot illustrate true end-to-end transitive zero-copy. Nevertheless, for a subset of communicating component VMs, i.e., from the FS VM onto the RAID and Storage VMs, transitive zero-copy is achieved by incorporating our PGVAS enhancements (Section 3.3.4). This improves our end-to-end performance significantly.

5.4 Case Study Evaluation

To evaluate the performance of the virtualized storage system architecture, we run a set of experiments on the following three systems:

- **Monolithic-Linux**: Traditional Linux running on hardware, with all storage components located in the same kernel address space.
- **Native-Xen**: Virtualized storage architecture with four VMs (Section 5.3) connected using the native Xen inter-VM communication mechanisms—Netfront/NetBack and Blkfront/Blkback.
- **MM-Xen**: Virtualized storage architecture with MMNet and MMBlk as shown in Figure 9.

Rephrasing the evaluation goals in the context of these systems, we expect that for the MMNet and MMBlk mechanisms to be *effective*, performance of MM-Xen should be significantly better than Native-Xen and for the virtualized architecture to be *viable*, the performance difference between Monolithic-Linux and MM-Xen should be minimal.

5.4.1 System Configuration

We now present the configuration details of the system. The physical machine described in Section 4.1 is used as the storage server. The client machine, running Linux (kernel version 2.6.18), has similar configuration as the server, except for the following differences: two dual-core 2.1 GHz AMD Opteron processors, 8 GB of memory and a single internal disk. The two machines are connected via a Gigabit Ethernet switch.

In the Monolithic-Linux experiments, we run native Linux with eight physical cores and 7 GB of memory. In the Native-Xen and MM-Xen experiments, there are four VMs on the server (the disk driver VM is basically dom0). The FS VM is configured with two virtual CPUs, each of the other VMs have one virtual CPU. Each virtual CPU is assigned to a dedicated physical processor core. The FS VM is configured with 4 GB of memory and the other VMs are configured with 1 GB each. The RAID VM includes a Linux MD [22] software RAID5 device of 480 GB capacity, constructed with two data disks and one parity disk. The RAID device is configured with a 1024 KB chunk size and a 64 KB stripe cache (write-through cache of recently accessed stripes). The ext3 file system created on the RAID5 device is exported by the NFS server in “async” mode. The “async” export option mimics the behavior of enterprise-class networked storage systems, which typically employ some form of non-volatile memory to hold dirty write data [12] for improved write performance. Finally, the client machine uses the native Linux NFS client to mount the exported file system. The NFS client uses TCP as the transport protocol with a 32 KB block size for read and write.

5.4.2 Microbenchmarks

We use the IOZone [15] benchmark to compare the performance of Monolithic-Linux, Native-Xen and MM-Xen. We perform read and write tests, in both sequential and random modes. In each of these tests, we vary the IOZone record sizes, but keep the file size constant. The file size is 8 GB for both sequential and random tests.

Figure 10 presents the throughput results. For sequential writes, as shown in Figure 10A, MM-Xen achieves an average improvement of 88% over the Native-Xen configuration. This shows that Fido performance improvements help the throughput of data transfer significantly. Moreover, MM-Xen outperforms even Monolithic-Linux by 9.5% on average. From Figure 10C, we see that MM-Xen achieves similar relative performance even with random writes. This could be due to the benefits of increased parallelism and pipelining achieved by running VMs on isolated cores. In the monolithic case, kernel locking and scheduling inefficiencies could limit such pipelining. Even with sequential reads, as shown in Figure 10B, MM-Xen outperforms both Monolithic-Linux and Native-Xen by about 13%. These results imply that our architecture has secondary performance benefits when the kernels in individual VMs exhibit SMP inefficiencies.

With random workloads, since the size of the test file

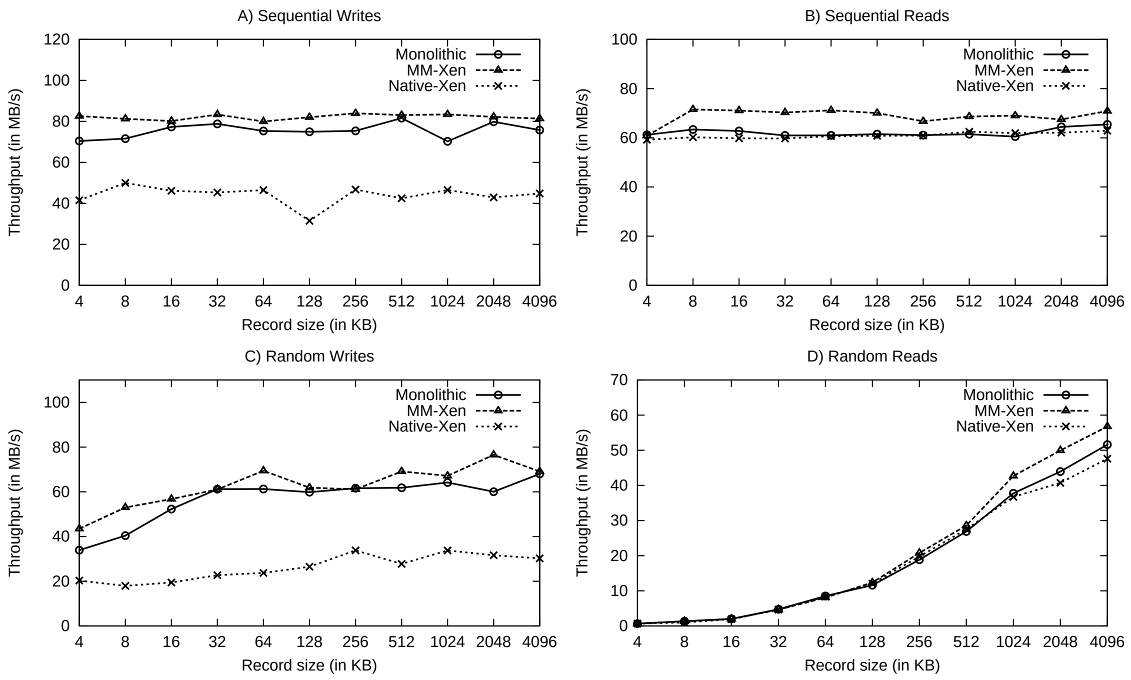

Figure 10: *IOZone Throughput Results*

remains constant, the number of seeks reduces as we increase the record size. This explains why the random read throughput (Figure 10D) increases with increasing record sizes. However, random writes (Figure 10C) do not exhibit similar throughput increase due to the mitigation of seeks by the coalescing of writes in the buffer cache (recall that the NFS server exports the file system in ”async” mode).

Finally, Figure 11 presents the IOZone latency results. We observe that MM-Xen is always better than Native-Xen. Moreover, MM-Xen latencies are comparable to Monolithic-Linux in all cases.

5.4.3 Macrobenchmarks

TPCC-UVa [18] is an open source implementation of the TPC-C benchmark version 5. TPC-C simulates read-only and update intensive transactions, which are typical of complex OLTP (On-Line Transaction Processing) systems. TPCC-UVa is configured to run a one hour test, using 50 warehouses, a ramp-up period of 20 minutes and no database `vacuum` (garbage collection and analysis) operations.

Table 1 provides a comparison of TPC-C performance across three configurations: Monolithic-Linux, Native-Xen, and MM-Xen. The main TPC-C metric is `tpmC`, the cumulative number of transactions executed per minute. Compared to Monolithic-Linux, Native-Xen exhibits a 38% drop in `tpmC`. In contrast, MM-Xen is only 3.1% worse than Monolithic-Linux.

The response time numbers presented in Table 1 are averages of the response times from five types of transactions that TPC-C reports. We see that MM-Xen is within 13% of the average response time of Monolithic-Linux. These results demonstrate that our inter-VM communication improvements in the form of MMNet and MMBlk translate to good performance with macrobenchmarks.

	tpmC (transactions/min)	**Avg. Response Time** (sec)
Monolithic	293.833	26.5
Native-Xen	183.032	350.8
MM-Xen	284.832	30.4

Table 1: *TPC-C Benchmark Results*

6 Related Work

In this section we first present a survey of the different existing inter-VM communication approaches and articulate the trade-offs between them. Subsequently, since we use a shared-memory communication method, we articulate how our research leverages and complements prior work in this area.

6.1 Inter-VM Communication Mechanisms

Numerous inter-VM communication mechanisms already exist. Xen VMM supports a restricted inter-VM communication path in the form of Xen split drivers [9]. This mechanism incurs prohibitive overheads due to data copies or page-flipping via hypervisor calls in the critical path. XenSocket [28] provides a socket-like interface. However, XenSocket approach is not transparent. That is, the existing socket interface calls have to be changed. XenLoop [26] achieves efficient inter-VM communica-

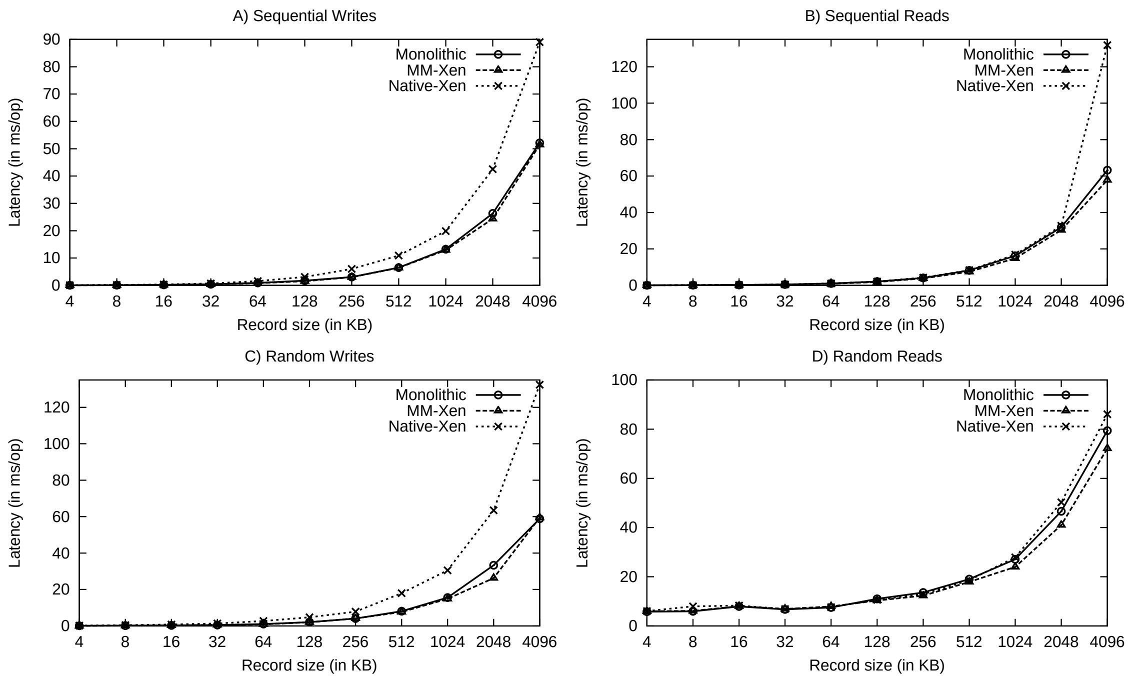

Figure 11: *IOZone Latency Results*

tion by snooping on every packet and short- circuiting packets destined to co-located VMs. While this approach is transparent, as well as non-intrusive, its performance trails MMNet performance since it incurs copies due to a bounded shared memory region between the communicating VMs. The XWay [17] communication mechanism hooks in at the transport layer. Moreover, this intrusive approach is limited to applications that are TCP oriented. In comparison to XWay and XenSocket, MMNet does not require any change in the application code, and MMNet's performance is better than XenLoop and XenSocket. Finally, IVC [13] and VMWare VMCI [24] provide library level solutions that are not system-wide.

6.2 Prior IPC Research

A lot of prior research has been conducted in the area of inter-process communication. Message passing and shared-memory abstractions are the two major forms of IPC techniques. Mechanisms used in Fbufs [6], IO-Lite [23], Beltway buffers [5] and Linux Splice [19] are similar to the IPC mechanism presented in this paper.

Fbufs is an operating system facility for I/O buffer management and efficient data transfer across protection domains on shared memory machines. Fbufs combine virtual page remapping and memory sharing. Fbufs target throughput of I/O intensive applications that require significant amount of data to be transferred across protection boundaries. A buffer is allocated by the sender with appropriate write permissions whereas the rest of the I/O paths access it in read-only mode. Thus, buffers are immutable. However, append operation is supported by aggregating multiple data-buffers into a logical message. Fbufs employ the following optimizations: a) mapping of buffers into the same virtual address space (removes lookup for a free virtual address) b) buffer reuse (buffer stays mapped in all address spaces along the path) and c) allows volatile buffers (sender doesn't have to make them read-only upon send). IO-Lite is similar in spirit to Fbufs, it focuses on zero-copy transfers between kernel modules by means of unified buffering. Some of the design principles behind Fbufs and IO-Lite can be leveraged on top of PGVAS in a virtualized architecture.

Beltway buffers [5] trade protection for performance implementing a zero-copy communication. Beltway allocates a system-wide communication buffer and translates pointers to them across address spaces. Beltway does not describe how it handles buffer memory exhaustion except for the networking case, in which it suggests to drop packets. Beltway enforces protection per-buffer, making a compromise between sharing entire address spaces and full isolation. Compared to us, Beltway simplifies pointer translation across address spaces – it translates only a pointer to buffer, inside the buffer linear addressing is used, so indexes inside the buffer remain valid across address spaces.

`splice` [19] is a Linux system call providing a zero-copy I/O path between processes (i.e. a process can send data to another process without lifting them to user-space). Essentially, Splice is an interface to access the in-kernel buffer with data. This means that a process can forward the data but cannot access it in a zero-copy way. Buffer memory management is implemented through reference counting. Splice ”copy” is essentially a creation of a reference counted pointer. Splice appeared in Linux since 2.6.17 onwards.

7 Conclusion

In this paper, we present Fido, a high-performance inter-VM communication mechanism tailored to software architectures of enterprise-class server appliances. On top of Fido, we have built two device abstractions-MMNet and MMBlk exporting the performance characteristics of Fido to higher layers. We evaluated MMNet and MM-Blk separately as well in the context of a virtualized network-attached storage system architecture and we observe almost imperceptible performance penalty due to these mechanisms. In all, employing Fido in appliance architectures makes it viable for them to leverage virtualization technologies.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP ’03*, New York, 2003.

[2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — A technique for cheap recovery. In *OSDI’04*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[3] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, 1994.

[4] Cisco Systems. Cisco Products. http://www.cisco.com/products.

[5] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the os traffic jam. In *Proceedings of INFOCOM 2008*, 2008.

[6] P. Druschel and L. L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *SOSP ’93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 189–202, New York, NY, USA, 1993. ACM.

[7] B. Dufrasne, W. Gardt, J. Jamsek, P. Kimmel, J. Myyrylainen, M. Oscheka, G. Pieper, S. West, A. Westphal, and R. Wolf. IBM System Storage DS8000 Series: Architecture and Implementation, Apr. 2008.

[8] EMC. The EMC Celerra Family. http://www.emc.com/products/family/celerra-family.htm.

[9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *OASIS*, Oct 2004.

[10] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of Linux Kernel Behavior under Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN’03)*, 2003.

[11] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Softw. Pract. Exper.*, 28(9):901–928, 1998.

[12] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *WTEC’94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[13] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda. Virtual Machine Aware Communication Libraries for High Performance Computing. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007*, Reno, Nevada, USA, November 2007.

[14] IBM Corporation. IBM Storage Controllers. http://www-03.ibm.com/systems/storage/network/index.html.

[15] IOZone. IOZone Filesystem Benchmark. http://www.iozone.org.

[16] Juniper Networks. Juniper Networks Products. http://www.juniper.com/products.

[17] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *VEE ’08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, New York, NY, USA, 2008. ACM.

[18] D. R. Llanos. Tpcc-uva: an open-source tpc-c implementation for global performance measurement of computer systems. *SIGMOD Rec.*, 35(4):6–15, 2006.

[19] L. McVoy. The splice I/O model. http://ftp.tux.org/pub/sites/ftp.bitmover.com/pub/splice.ps.

[20] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *ATEC ’06: Proceedings of the annual conference on USENIX ’06 Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[21] NetApp, Inc. NetApp Storage Systems. http://www. netapp.com/products.

[22] OSDL. Overview - Linux-RAID. http://linux-raid.osdl.org/index.php/Overview.

[23] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *ACM Transactions on Computer Systems*, pages 15–28, 2000.

[24] VMWare. Virtual Machine Communication Interface. http://pubs.vmware.com/ vmci-sdk/VMCI_intro.html.

[25] VMWare. VMWare Inc. http://www.vmware.com.

[26] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: A Transparent High Performance Inter-VM Network Loopback. In *Proc. of International Symposium on High Performance Distributed Computing (HPDC)*, June 2008.

[27] A. Watson, P. Benn, A. G. Yoder, and H. Sun. Multiprotocol Data Access: NFS, CIFS, and HTTP. Technical Report 3014, NetApp, Inc., Sept. 2001.

[28] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Middleware 2007: ACM/IFIP/USENIX 8th International Middleware Conference*, Newport Beach, CA, USA, November 2007.