MemX: Supporting Large Memory Workloads in Xen Virtual Machines

Michael R. Hines and Kartik Gopalan
State University of New York at Binghamton
{mhines,kartik}@cs.binghamton.edu

Abstract

Modern grid computing and enterprise applications increasingly execute on clusters that rely upon virtual machines (VMs) to partition hardware resources and improve utilization efficiency. These applications tend to have memory and I/O intensive workloads, such as large databases, data mining, scientific workloads, and web services, which can strain the limited I/O and memory resources within a single VM. In this paper, we present our experiences in developing a fully transparent distributed system, called MemX*, within the Xen VM environment that coordinates the use of cluster-wide memory resources to support large memory and I/O intensive workloads. Applications using* MemX *do not require specialized APIs, libraries, recompilation, relinking, or dataset pre-partitioning. We compare and contrast the different design choices in* MemX *and present preliminary performance evaluation using several resource-intensive benchmarks in both virtualized and non-virtualized Linux. Our evaluations show that large dataset applications and multiple concurrent VMs achieve significant speedups using* MemX *compared against virtualized local and iSCSI disks. As an added benefit, we also show that live Xen VMs using* MemX *can migrate seamlessly without disrupting any running applications.*

1 Introduction

In modern cluster-based platforms, Virtual Machines (VMs) can enable functional and performance isolation across applications and services. VMs also provide greater resource allocation flexibility, improve the utilization efficiency, enable seamless load balancing through VM migration, and lower the operational cost of the cluster. Consequently, VM environments are increasingly being considered for executing grid and enterprise applications over commodity high-speed clusters.

However, such applications tend to have memory and I/O intensive workloads that can stress the limited resources within a single VM by demanding more memory than the slice available to the VM. Clustered bastion hosts (mail, network attached storage), data mining applications, scientific workloads, virtual private servers, and backend support for websites are common examples of resource-intensive workloads. I/O bottlenecks can quickly form due to frequent access to large disk-resident dataset, frequent paging activity, flash crowds, or competing VMs on the same node. Even though virtual machines with demanding workloads are here to stay as integral parts of modern clusters, significant improvements are needed in the ability of memory-constrained VMs to handle these workloads.

I/O activity due to memory pressure can prove to be particularly expensive in a virtualized environment where all I/O operations need to traverse an extra layer of indirection. Over-provisioning of memory resources (and in general any hardware resource) within each physical machine is not a viable solution as it can lead to poor resource utilization efficiency, besides increasing the operational costs. Although domain-specific out-of-core computation techniques [13, 15] and migration strategies [19, 2, 6] can also improve the application performance up to a certain extent, they do not overcome the fundamental limitation that an application is restricted to using the memory resources within a single physical machine.

In this paper, we present the design, implementation, and evaluation of the *MemX* system for VMs that bridges the I/O performance gap in a virtualized environment by exploiting low-latency access to the memory of other nodes across a Gigabit cluster. *MemX* significantly reduces the execution times for memory and I/O intensive large dataset applications, provides support for multiple concurrently executing VMs to utilize cluster-wide memory, and even supports seamless migration of live VMs with large memory workloads. *MemX* is fully transparent to the user applications – developers do not need any specialized APIs, libraries, recompilation, or relinking for their applications, nor does the application's dataset need any special pre-processing, such as data partitioning across nodes.

We compare and contrast the three modes in which *MemX* can operate with Xen VMs [4], namely, (1) within non-virtualized Linux (*MemX*-Linux), (2) within individual guest OSes (*MemX*-DomU), and (3) within a common driver domain (*MemX*-DD) shared by multiple guest OSes. We demonstrate, via a detailed evaluation using several I/O intensive and large memory benchmarks, that these applications achieve significant speedups using the *MemX* system when compared against virtualized disk-based I/O. The proposed techniques can also work with other VM technologies besides Xen. We focus on Xen mainly due to its open source availability and para-virtualization support.

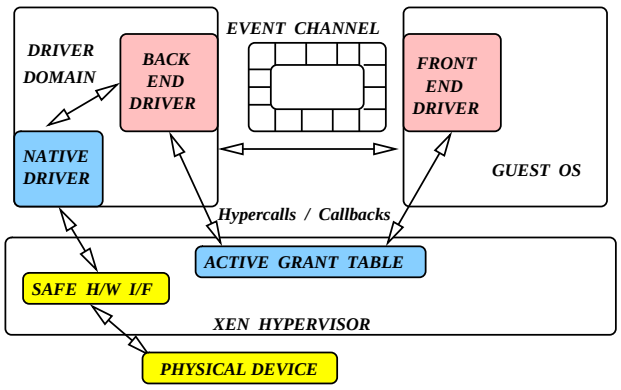

Figure 1. Split Device Driver Architecture in Xen.

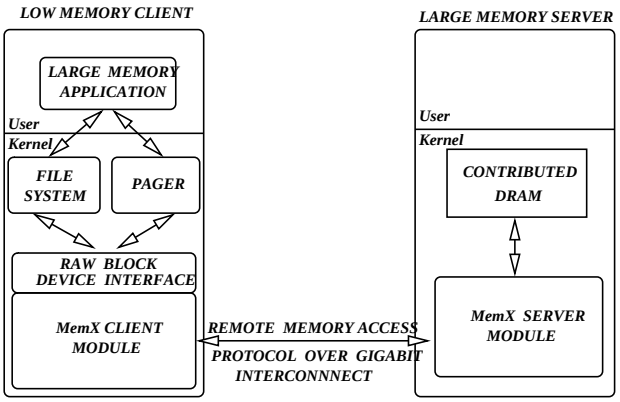

Figure 2. *MemX*-Linux: Baseline operation of *MemX* in a non-virtualized Linux environment.

2 Split Driver Background

Xen is a virtualization technology that provides close to native machine performance through the use of *para-virtualization* – a technique by which the guest OS is co-opted into reducing the virtualization overhead via modifications to its hardware dependent components. In this section, we review the background of the Xen I/O subsystem as it relates to the design of *MemX*. Xen exports I/O devices to each guest OS as virtualized views of "class" devices, as opposed to real physical devices. For example, Xen exports a block device or a network device, rather than a specific hardware make and model. The actual drivers that interact with the native hardware devices can either execute within Dom0 – a privileged domain that can directly access all hardware in the system – or within Isolated Driver Domains (IDD), which are essentially driver specific virtual machines. IDDs require the ability to hide PCI devices from Dom0 and expose them to other domains. In the rest of the paper, we will use the term *driver domain* to refer to either Dom0 or the IDD that hosts the native device drivers.

Physical devices can be multiplexed among multiple concurrently executing guest OSes. To enable this multiplexing, the privileged driver domain and the unprivileged guest domains (DomU) communicate by means of a split device-driver architecture shown in Figure 1. The driver domain hosts the *backend* of the split driver for the device class and the DomU hosts the *frontend*. The backends and frontends interact using high-level device abstractions instead of low-level hardware specific mechanisms. For example, a DomU only cares that it is using a block device, but doesn't worry about the specific type of block device.

Frontends and backends communicate with each other via the grant table – an in-memory communication mechanism that enables efficient bulk data transfers across domain boundaries. The grant table enables one domain to allow another domain to access its pages in system memory. The access mechanism can include read, write, or mutual exchange of pages. The primary use of the grant table in device I/O is to provide a fast and secure mechanism for unprivileged domains (DomUs) to receive indirect access to hardware devices. They enable the driver domain to set up a DMA based data transfer directly to/from the system memory of a DomU rather than performing the DMA to/from driver domain's memory with the additional copying of the data between DomU and driver domain.

The grant table can be used to either share or transfer pages between the DomU and driver domain depending upon whether the I/O operation is synchronous or asynchronous in nature. For example, block devices perform synchronous data transfer, that is, the driver domain knows at the time of I/O initiation as to which DomU requested the block I/O. In this case the frontend of the block driver in DomU will notify the Xen hypervisor (via the `gnttab_grant_foreign_access` hypercall) that a memory page can be shared with the driver domain. The DomU then passes a grant table reference ID via the event channel to the driver domain, which sets up a direct DMA to/from the memory page of the DomU. Once the DMA is complete, the DomU removes the grant reference (via the `gnttab_end_foreign_access` call).

On the other hand, network devices receive data asynchronously, that is, the driver domain does not know the target DomU for an incoming packet until the entire packet has been received and its header examined. In this situation, the driver domain DMAs the packet into its own page and notifies the Xen hypervisor (via the `gnttab_grant_foreign_transfer` call) that the page can be transferred to the target DomU. The driver domain then transfers the received page to target DomU and receives a free page in return from the DomU.

3 Design and Implementation

Here we discuss different design alternatives for *MemX*, justify our decisions, and present implementation details.

3.1 *MemX*-Linux

Figure 2 shows the operation of *MemX*-Linux, i.e., *MemX* in a non-virtualized (vanilla) Linux environment. *MemX*-Linux provides several new features compared to its predecessor [12]. Below we summarize the architecture of *MemX*-Linux for completeness and use it as a baseline for comparison with other virtualized versions of *MemX*, which are the primary focus of this paper. Two main components of *MemX*-Linux are the *client module* on the low memory machines and the *server module* on the machines with unused memory. The two communicate using a *remote memory access protocol* (RMAP). Both client and server components execute as isolated Linux kernel modules.

Client Module: The client module provides a virtualized *block device* interface to the large dataset applications executing on the client machine. This block device can either be configured as a low-latency primary swap device, or treated as a low-latency volatile store for large datasets accessed via the standard file-system interface, or memory mapped to the the address space of an executing large memory application. To the rest of the client system, the block device appears to be a simple I/O partition with a linear I/O space that is no different from a regular disk partition, except that the access latency happens to be over an order of magnitude smaller than disk. Internally, however, the client module maps the single linear I/O space of the block device to the unused memory of multiple remote servers, using a memory-efficient radix-tree based mapping. The client module also bypasses a standard request-queue mechanism used in Linux block device interface, which is normally used to group together spatially consecutive block I/Os on disk. This is because, unlike physical disks, the access latency to any offset within this block device is almost constant over a gigabit LAN, irrespective of the spatial locality. The client module also contains a small bounded-sized write buffer to quickly service write I/O requests.

Server Module: A server module stores pages in memory for any client node across the LAN. The server modules do not have any externally visible interface on the server machines, except for basic initialization and control. They communicate with clients over the network using a custom-designed *remote memory access protocol* (RMAP) that is described later. Servers broadcast periodic resource announcement messages which the client modules can use to discover the available memory servers. Servers also include feedback about their memory availability and load during both resource announcement as well as regular page transfers with clients. When a server reaches capacity, it declines to serve any new write requests from clients, which then try to select another server, if available, or otherwise write the page to disk. The server module is also designed to allow a server node to be taken down while live; our RMAP implementation can *disperse, re-map, and load-balance* an individual server's pages to any other servers in the cluster that are capable of absorbing those pages, allowing the server to shut down without killing any of its client's applications.

Remote Memory Access Protocol (RMAP) RMAP is a tailor-designed light-weight window-based reliable datagram protocol for remote memory access within the same subnet. It incorporates the following features: (1) Reliable Packet Delivery, (2) Flow-Control, and (3) Fragmentation and Reassembly. While one could technically communicate over TCP, UDP, or even the IP protocol layers, this choice comes burdened with unwanted protocol processing. For instance, *MemX* does not require TCP's features such as byte-stream abstraction, in-order delivery, or congestion control. Nor does it require IP routing functionality, being a single-subnet system. Thus RMAP bypasses the TCP/IP protocol stack and communicates directly with the network device driver. Every RMAP message is acknowledged except for soft-state and dynamic discovery messages. All client nodes keep a fixed-size window to control the transmission rate, which works well for purely in-cluster communication. Another consideration is that while the standard memory page size is 4KB (or sometimes 8KB), the maximum transmission unit (MTU) in traditional Ethernet networks is limited to 1500 bytes. Thus RMAP implements dynamic fragmentation/reassembly for page transfer traffic. Additionally, RMAP also has the flexibility to use *Jumbo frames*, which are packets with sizes greater than 1500 bytes (typically between 8KB and 16KB), that enable transmission of complete 4KB pages using a single packet.

Additional Features: *MemX* also includes several additional features that are not the specific focus of this paper. These include a soft-state refresh mechanism for tracking liveness of clients and servers, server load balancing, RAID-like reliability over remote memory, named remote memory data spaces that can be shared by multiple clients, support for multiple block device minor numbers by a single client module (important in providing remote memory access to multiple DomUs in virtualized environment), and zero data copies during network communication (or a single copy if fragmentation is required).

3.2 *MemX*-DomU (Option 1): *MemX* Client Module in DomU

In order to support memory intensive large dataset applications within a VM environment, the simplest design option is to place the *MemX* client module within the kernel of each guest OS (DomU), whereas remote server modules continue to execute within non-virtualized Linux kernel. This option is illustrated in Figure 3. The client module exposes the block device interface for large memory

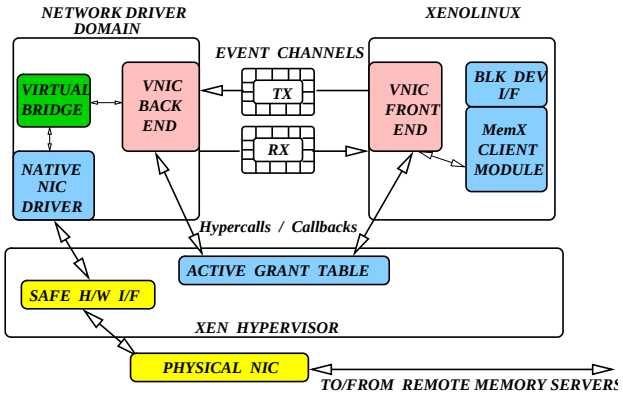

Figure 3. *MemX*-DomU: Inserting *MemX* client module within DomU's Linux kernel. Server executes in non-virtualized Linux.

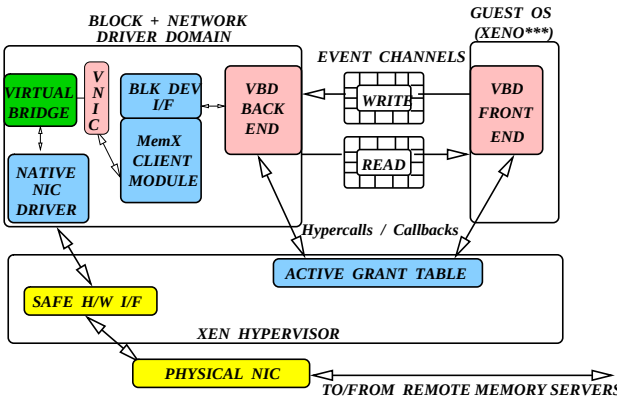

Figure 4. *MemX*-DD: A shared *MemX* client within a privileged driver domain multiplexes I/O requests from multiple DomUs.

applications within the DomU as in the baseline, but communicates with the remote server using the virtualized network interface (VNIC) exported by the network driver domain. The VNIC in Xen is organized as a split device driver in which the frontend (residing in the guest OS) and the backend (residing in the network driver domain) talk to each other using well-defined grant table and event channel mechanisms. Two event channels are used between the backend and frontend of the VNIC – one for packet transmissions and one for packet receptions. To perform zero-copy data transfers across the domain boundaries, the VNIC performs a page exchange with the backend for every packet received or transmitted using the grant table. All backend interfaces in the network driver domain can communicate with the physical NIC as well as with each other via a virtual network bridge. Each VNIC is assigned its own MAC address whereas the driver domain's VNIC (in Dom0) uses the physical NIC's MAC address. The physical NIC itself is placed in promiscuous mode by Xen's driver domain to enable the reception of any packet addressed to any of the local virtual machines. The virtual bridge demultiplexes incoming packets to the target VNIC's backend driver.

Compared to the baseline non-virtualized *MemX* deployment, *MemX*-DomU has the additional overhead of requiring every network packet to traverse across domain boundaries in addition to being multiplexed or demultiplexed at the virtual network bridge. Additionally, the client module needs to be separately inserted within each DomU that might potentially execute large memory applications. Also note that each I/O request is typically 4KBytes (or sometimes 8KBytes) in size, whereas network hardware uses a 1500-byte MTU (maximum transmission unit), unless the underlying network supports Jumbo frames. Thus the client module needs to fragment each 4KByte write request into (and reassemble a complete read reply from) at least 3 network packets. In *MemX*-DomU each fragment needs to traverse the domain boundary to reach the the backend. Due to current memory allocation policies in Xen, buffering for each fragment ends up consuming an entire 4KByte page worth of memory allocation, i.e., three times the typical page size. We will contrast this performance overhead in greater detail with *MemX*-DD option below.

3.3 *MemX*-DD (Option 2): *MemX* Client Module in Driver Domain

A second design option is to place the *MemX* client module within a block driver domain (Dom0) and allow multiple DomUs to share this common client module via their virtualized block device (VBD) interfaces. This option is shown in Figure 4. The guest OS executing within the DomU does not require any *MemX* specific modifications. The *MemX* client module executing within the driver domain exposes a block device interface, as before. Any DomU, whose applications require remote memory resources, configures a split VBD. The frontend of the VBD resides in DomU and the backend in the block driver domain. The frontend and backend of each VBD communicates using event channels and the grant table, as in the earlier case of VNICs. The *MemX* client module provides a separate VBD lettered-slice (/dev/memx{a,b,c}, etc.) for each backend that corresponds to a distinct DomU. On the network side, the *MemX* client module attaches itself to the driver domain's VNIC which in turn talks to the physical NIC via the virtual network bridge. For performance reasons, here we assume that the network driver domain and disk driver domain are co-located within a single privileged domain (such as Dom0). Thus the driver domain's VNIC does not need to be organized as another split driver. Rather it is a single software construct that can attach directly to the virtual network bridge. During execution within a DomU, read/write requests to remote memory are generated in the form of synchronous I/O requests to the corresponding VBD's frontend. These requests are sent to the *MemX* client module via the event channel and the grant

table. The client module packages each I/O request into network packets and transmits them asynchronously to remote memory servers using RMAP.

Note that, although the network packets still need to traverse the virtual network bridge, they no longer need to traverse a split VNIC architecture, unlike in *MemX*-DomU. One consequence of not going through a split VNIC architecture is that, while client module still needs to fragment a 4KByte I/O request into 3 network packets to fit the MTU requirements, each fragment no longer needs to occupy an entire 4KByte buffer, unlike in *MemX*-DomU. As a result, only one 4KByte I/O request needs to cross the domain boundary across the split VBD driver, as opposed to three 4KB packet buffers in *MemX*-DomU. Secondly, the *MemX* client module can be inserted once in the driver domain and still be shared among multiple DomUs. Finally, since the guest OS within DomUs do not require any *MemX* specific software components; the DomUs can potentially run any para-virtualized OS and not just XenoLinux.

However, compared to the non-virtualized baseline case, *MemX*-DD still has the additional overhead of using the split VBD and the virtual network bridge. Also note that, unlike *MemX*-DomU, *MemX*-DD does not currently support seamless migration of live Xen VMs using remote memory. This is because part of the guest's internal state (page-to-server mappings) that resides in the driver domain of *MemX*-DD is not automatically transferred by the migration mechanism in Xen. We plan to enhance Xen's migration mechanism to transfer this internal state in a host-independent manner to the target machine's *MemX*-DD module. Additionally, we plan to support per-DomU reservation of remote memory to maintain isolation guarantees.

3.4 (Option 3): *MemX* Server in DomU

Technically speaking, we can also execute the *MemX* server module within a guest OS, coupled with Options 1 or 2 above. This could enable one to initiate a VM solely for the purpose of providing remote memory to other low-memory client VMs that are either across the cluster or even within the same physical machine. However, practically, this option does not seem to provide any significant functional benefits whereas the overheads of executing the server module within a DomU are considerable. The bottom-line is that equivalent remote memory functionality can be provided more efficiently by a *MemX* server module running in a non-virtualized environment. Consequently, we do not pursue this option further.

3.5 (Option 4): Expanding the Pseudo-physical Address Space of Guest OS

Another alternative to supporting large memory applications with remote memory is to enable the guest OS to view a larger *pseudo-physical* memory address space than the available physical memory within the local machine. This option would require fundamental modifications to the memory management in both the Xen hypervisor as well as the guest OS. In particular, at boot time, the guest OS would believe that it has a large "physical" memory – or the so called pseudo-physical memory space. It then becomes the Xen hypervisor's task to map each DomU's large pseudo-physical address space partly into local physical memory, partly into remote memory, and the rest to secondary storage. This is analogous to the large conventional virtual address space available to each process that is managed transparently by traditional operating systems. The functionality provided by this option is essentially equivalent to that provided by *MemX*-DomU and *MemX*-DD. However, this option requires the Xen hypervisor to take up a prominent role in memory address translation process, something that original design of Xen strives to minimize. This option also runs the risk of paying a significant penalty for *double (page) faults* – the situation where the paging daemon in DomU mistakenly attempts to swap out a page that already resides on a swap device managed by the hypervisor (or vice-versa), resulting in the target page being brought into physical memory and getting swapped out of the system again immediately. Due to the above problems and lack of functional benefits, we do not pursue this option further.

4 Performance Evaluation

In this section we evaluate the performance of the different variants of *MemX*. Our testbed consists of eight machines, each having 4 GB of memory, 64-bit dual-core 2.8 Ghz processor, and Gigabit Broadcom Ethernet NIC. Our Xen version is 3.0.4 and XenoLinux version 2.6.18. Backend *MemX* servers run on vanilla Linux 2.6.20. Collectively, this provides us with over 24GB of effectively usable cluster-wide memory after accounting for roughly 1GB of local memory usage per node. The local memory of client machines is either 512MB or 1GB in all tests. In addition to the three *MemX* configurations described earlier, namely *MemX*-Linux, *MemX*-DomU, and *MemX*-DD, we also include a fourth configuration – *MemX*-Dom0 – for the sole purpose of baseline performance evaluation. This additional configuration corresponds to the *MemX* client module executing within Dom0 itself, but not as part of the backend for a VBD. Rather, the client module in *MemX*-Dom0 serves large memory applications executing within Dom0. Furthermore, "disk" baseline refers to virtualized disk within Dom0, which is exported as a VBD to guest-VMs.

4.1 Microbenchmarks

Table 1 compares different *MemX*-combinations and virtual disk in terms of latency and bandwidth. RTT is the

	MemX-Linux	*MemX*-Dom0	*MemX*-DD	*MemX*-DomU	Virtual Disk
RTT (μs)	85	95	95	115	8300
Write b/w ($Mbps$)	950	950	915	852	295
Read b/w ($Mbps$)	916	915	915	840	295

Table 1. Latency and bandwidth comparison.

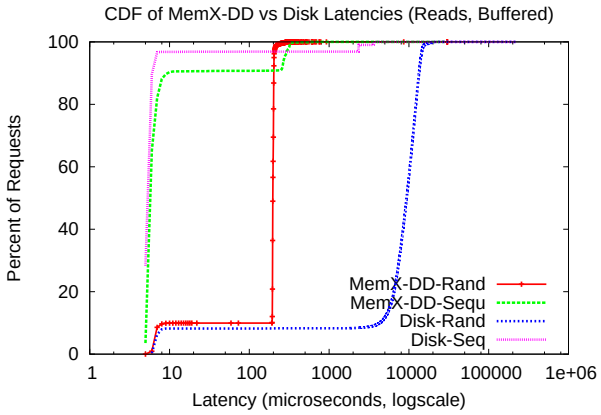

Figure 5. Sequential/random read latency distributions.

average round trip time for a single 4KB read request from a *MemX* client to a server, measured in kernel using the on-chip time stamp counter (TSC). This is the latency that the I/O requests from the VFS (virtual filesystem) or the system pager would experience. *MemX*-Linux, as a base case, provides an RTT of 85μs. Following close behind are *MemX*-Dom0, *MemX*-DD, and *MemX*-DomU in that order. The virtualized disk base case performs as expected at an average 8.3ms. These RTT numbers show that accessing the memory of remote machine over the network is about two orders of magnitude faster than from local virtualized disk. Also, the Xen VMM introduces negligible overhead of 10μs in *MemX*-Dom0 and *MemX*-DD over *MemX*-Linux. Similarly the split network driver architecture, which needs to transfer 3 packet fragments for each 4KB block across the domain boundaries, introduces an overhead of another 20μs in *MemX*-DomU over *MemX*-Dom0 and *MemX*-DD. Bandwidth measurements are performed using a custom benchmark which issues a stream of sequential asynchronous 4KB I/O requests, with the the range of I/O offsets being at least twice the size of client memory. We observe that the bandwidth reduction for *MemX*-DD and and *MemX*-DomU is small over the baseline – about 35Mbps for *MemX*-DD and 98Mbps for *MemX*-DomU. Virtual disk bandwidth trails far behind all the variants of *MemX*.

Figure 5 compares the read latency distribution for a user level application that performs either sequential or random I/O on either *MemX* or the virtual disk. Random read latencies are an order of magnitude smaller with *MemX*-DD (around 160μs) than with disk (around 9ms). Sequential read latency distributions are similar for *MemX*-DD and disk primarily due to filesystem prefetching. RTT distribution for buffered write requests (not shown) are similar for both *MemX*-DD and disk, mostly less than 10μs due to write buffering. Note that these RTT-values are measured from user-level, which adds a few tens of microseconds to the kernel-level RTTs in Table 1.

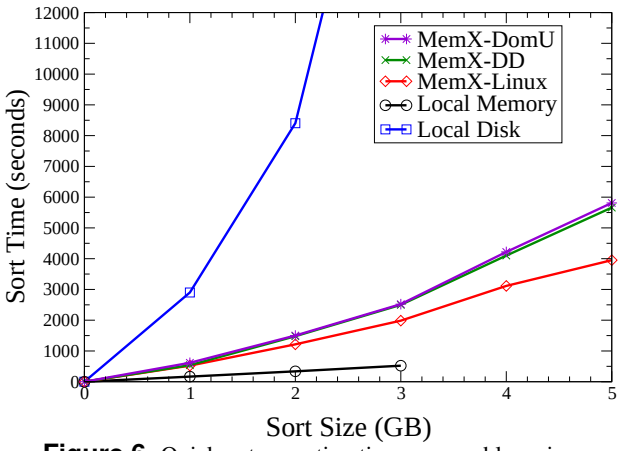

Figure 6. Quicksort execution times vs. problem size.

4.2 Application Speedups

We now evaluate the execution times of a few large memory applications using our testbed. Again, we include *both* *MemX*-Linux and virtual disk as base cases to illustrate the overhead imposed by Xen virtualization and the gain over the virtualized disk respectively. Figure 6 shows the performance of a very large sort of increasingly large arrays of integers, using a C implementation of Quicksort. We also include a base case plot for pure in-memory sorting using a vanilla-Linux 4 GB node. From the figure, we ceased to even bother with the disk case beyond 2GB problem sizes due to the unreasonably large amount of time it takes to complete, potentially for days. The sorts using *MemX*-DD, *MemX*-domU, and *MemX*-Linux however finished within 100 minutes, where the distinction between the different modes is very small. Table 2 lists the execution times for much larger problem sizes including (1) ray-tracing based graphics rendering application called POV-ray [22], (2) Integer Sort (IS-NPB) benchmark in the NAS Parallel Benchmark (NPB) suite [21], (3) Datacube (DC-NPB) benchmark in the NPB suite, (4) Sql-bench [20] benchmark on a `mysql` database that is stored in either remote memory or virtual disk, (5) NS2 [17] – the popular network simulation tool that is used to simulate a delay partitioning algorithm on a 6-hop wide-area network path, and (6) Quicksort again on a 5GB dataset. Again, both the *MemX* cases outperform the virtual disk case for each of the benchmarks.

4.3 Multiple Client VMs

In this section, we evaluate the overhead of executing multiple client VMs using *MemX*-DD. In most data centers or grid clusters, a high-speed backend interconnect, such as

Appli-cation	Mem Size	Client Mem	*MemX*-Linux	*MemX*-DD	Virtual Disk
Quicksort	5GB	512 MB	65 min	93 min	> 10 hrs
Povray	(6GB)	512 MB	48 min	61 min	> 10 hrs
Povray	(13GB)	1 GB	93 min	145 min	> 10 hrs
IS-NPB	(6GB)	512 MB	83 min	126 min	> 10 hrs
DC-NPB	(10GB)	1 GB	152 min	217 min	> 10 hrs
sql-bench	(4GB)	512 MB	114 min	208 min	> 10 hrs
NS2	(5GB)	1 GB	175 min	228 min	> 10 hrs

Table 2. Execution time comparisons for various large memory application workloads.

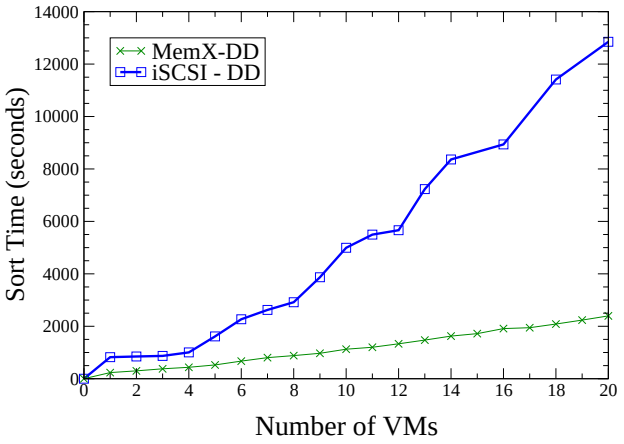

Figure 7. Quicksort execution times for multiple concurrent guest VMs using *MemX*-DD and iSCSI configurations.

iSCSI or FibreChannel, would provide backend storage for guest VMs. To emulate this base case in our cluster, we use five dual-core 4GB memory machines to evaluate *MemX*-DD in a 4-disk parallel iSCSI setup. We used the open source iSCSI target software from IET [23] and the initiator software from open-iscsi.org within Dom0 as a driver domain for all the Xen Guests. One of the five machines executed up to twenty concurrent 100MB Xen Guests, each hosting a 400MB Quicksort application. We vary the number of concurrent guest VMs from 1 to 20, and in each guest we run Quicksort to completion. We perform the same experiment for both *MemX*-DD and iSCSI. Figure 7 shows that at about 10 GB of collective memory and 20 concurrent virtual machines the execution time with *MemX*-DD is about 5 times smaller than with iSCSI setup. Recall that we are using four remote iSCSI disks, and one can observe a stair-step behavior in the iSCSI curve where the level of parallelism reaches 4, 8, 12, and 16 VMs. Even with concurrent disks and competing virtual machine CPU activity, *MemX*-DD provides clear performance edge.

4.4 Live VM Migration

MemX-DomU configuration has a significant benefit when it comes to migrating live Xen VMs [6] to better utilize resources, even though *MemX*-DD has higher bandwidth and lower I/O latency than *MemX*-DomU. Specifically, a VM using *MemX*-DomU can be seamlessly migrated from one physical machine to another, without disrupting the execution of any memory intensive large dataset applications within the VM. First, since *MemX*-DomU is designed as a self-contained pluggable module within the guest OS, any page-to-server mapping information is migrated along with the kernel state of the guest OS without leaving any residual dependencies behind in the original machine. Second reason is that RMAP used for communicating read-write requests to remote memory is designed to be reliable. As the VM carries with itself its link layer MAC address identification during the migration process, any in-flight packets dropped during migration are safely retransmitted to the VM's new location. As a preliminary evaluation, we conducted an experiment to compare the live VM migration performance using iSCSI versus *MemX*-DomU. For the iSCSI experiment, we configured a single iSCSI disk as swap space. Similarly, for the *MemX*-DomU case, we configured the block device exported by client module as the swap device. In both configurations, we ran 1GB Quicksort within a 512 MB guest. The live migration took an average of 26 seconds to complete in the iSCSI setup whereas it took 23 seconds with *MemX*-DomU. While further evaluation is necessary, this preliminary experiment points to potential benefits of using *MemX*-DomU to support large memory applications.

5 Related Work

To the best of our knowledge, *MemX* is the first system in a VM environment that provides unmodified large memory applications with a completely transparent and virtualized access to cluster-wide remote memory over commodity gigabit Ethernet LANs. Several prior efforts have focused upon remote memory access in *non-virtualized environments* [7, 10, 16, 18, 11, 12, 24, 8]. Distributed shared memory (DSM) systems [9, 1] allow a set of independent nodes to behave as a large shared memory multi-processor, often requiring customized programming to share common data across nodes. We are unaware of any DSM systems to date that work efficiently and transparently within a virtualized environment. Kerrighed [14] and vNuma [5] implement a single system image on top of multiple workstations using DSM techniques. However, they do not target support for multiple concurrent VMs, such as Xen guests.

Techniques also exist to migrate applications [19] or entire VMs [2, 6] to nodes that have more free resources (memory, CPU) or better data access locality. Both Xen [6] and VMWare [2] support migration of VMs from one physical machine to another, for example, to move a memory-hungry enterprise application from a low-memory node to a memory-rich node. However large memory applications within each VM are still constrained to execute within the memory limits of a single physical machine at any time. *In fact, we have shown in this paper that* MemX *can be used*

in conjunction with the VM migration in Xen, combining the benefits of both live VM migration and remote memory access. MOSIX [3] is a management system that uses process migration to allow sharing of computational resources among a collection of nodes, as if in a single multiprocessor machine. However each process is still restricted to use memory resources within a single machine.

Another approach is to develop domain specific out-of-core computation techniques such as [13, 15]. Out-of-core solutions tend to be highly application specific, requiring new algorithms for each new application domain. These techniques, while challenging in themselves, divert the efforts of the application developers from the core functionality of the application itself. Although both out-of-core techniques and migration can alleviate the I/O bottleneck to a limited extent, they do not overcome the fundamental limitation that an application is restricted to the memory available within a single machine. *MemX* provides an alternative, and perhaps even a complement, to both approaches by enabling transparent use of cluster-wide memory resources.

6 Conclusions

State-of-the-art in virtual machine technology does not adequately address the needs of memory and I/O intensive workloads that are increasingly common in modern grid computing and enterprise applications. In this paper, we presented the design, implementation, and evaluation of the *MemX* system in the Xen environment that enables memory and I/O intensive VMs to transparently utilize the collective pool of memory across a cluster. Large dataset applications using *MemX* do not require any specialized APIs, libraries, or any other modifications. *MemX* can operate as a kernel module within non-virtualized Linux (*MemX*-Linux), an individual VM (*MemX*-DomU), or a shared driver domain (*MemX*-DD). Preliminary evaluations of our *MemX* prototype using several different benchmarks shows that I/O latencies are reduced by an order of magnitude and that large memory applications speed up significantly when compared to virtualized disk. Additionally, live Xen VMs executing large memory applications over *MemX*-DomU can be migrated without disrupting applications. Our ongoing work includes the capability to provide per-VM reservations over the cluster-wide memory, developing mechanisms to control inter-VM congestion, and enabling seamless migration of VMs in the driver domain mode of operation.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[2] A. Awadallah and M. Rosenblum. The vMatrix: Server Switching. In *Proc. of Intl. Workshop on Future Trends in Distributed Computing Systems, Suzhou, China*, May 2004.

[3] A. Barak and O. Laadan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5), Mar. 1998.

[4] P. Barham, B. Dragovic, K. Fraser, and S. Hand et.al. Xen and the art of virtualization. In *SOSP*, Oct. 2003.

[5] M. Chapman and G. Heiser. Implementing transparent shared memory on clusters using virtual machines. In *Proc. of USENIX Annual Technical Conference*, 2005.

[6] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of NSDI*, 2005.

[7] D. Comer and J. Griffoen. A new design for distributed systems: the remote memory model. *Proc. of the USENIX 1991 Summer Technical Conference*, pages 127–135, 1991.

[8] F. Cuenca-Acuna and T. Nguyen. Cooperative caching middleware for cluster-based servers. In *Proc. of High Performance Distributed Computing*, Aug 2001.

[9] S. Dwarkadas, N. Hardavellas, and L. Kontothanassis et. al. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proc. of Intl. Parallel Processing Symposium*, pages 153–159, April 1999.

[10] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. *SOSP*, 1995.

[11] M. Flouris and E. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Cluster Computing*, 2(4):281–293, 1999.

[12] M. Hines, J. Wang, and K. Gopalan. Distributed Anemone: Transparent Low-Latency Access to Remote Memory in Commodity Clusters. In *Proc. of the International Conference on High Performance Computing (HiPC)*, Dec. 2006.

[13] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large N-dimensional scalar fields. In *Proc. of Eurographics 2003*, pages 343–348, September 2003.

[14] Kerrighed. *http://www.kerrighed.org*.

[15] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *Proc. of ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics*, April 2003.

[16] E. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *USENIX Annual Technical Conference*, pages 177–190, 1996.

[17] S. McCanne and S. Floyd. ns2: Network simulator, http://www.isi.edu/nsnam/ns/.

[18] I. McDonald. Remote paging in a single address space operating system supporting quality of service. Tech. Report, Computing Science, University of Glasgow, 1999.

[19] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration survey. *ACM Computing Surveys*, 32(3):241–299, Sep. 2000.

[20] MySQL AB. The MySQL Benchmark Suite http://dev.mysql.com/doc/refman/5.0/en/mysql-benchmarks.html.

[21] NAS Parallel Benchmarks. *http://www.nas.nasa.gov/Resources/Software/npb.html*.

[22] POV-Ray. The persistence of vision raytracer, http://povray.org/.

[23] The iSCSI Enterprise Target Project. *http://iscsitarget.sourceforge.net/*.

[24] T. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Proc. of the USENIX Annual Technical Conference*, pages 161–175, 2002.