MemX: Virtualization of Cluster-wide Memory

Umesh Deshpande, Beilan Wang, Shafee Haque, Michael Hines, Kartik Gopalan
Computer Science, State University of New York, Binghamton, NY
Contact: kartik@binghamton.edu

***Abstract*—We present MemX – a distributed system that virtualizes cluster-wide memory to support data-intensive and large memory workloads in virtual machines (VMs). MemX provides a number of benefits in virtualized settings: (1) VM workloads that access large datasets can perform low-latency I/O over virtualized cluster-wide memory; (2) VMs can transparently execute very large memory applications that require more memory than physical DRAM present in the host machine; (3) MemX reduces the effective memory usage of the cluster by de-duplicating pages that have identical content; (4) existing applications do not require any modifications to benefit from MemX such as the use of special APIs, libraries, recompilation, or relinking; and (5) MemX supports live migration of large-footprint VMs by eliminating the need to migrate part of their memory footprint resident on other nodes. Detailed evaluations of our MemX prototype show that large dataset applications and multiple concurrent VMs achieve significant performance improvements using MemX compared against virtualized local and iSCSI disks.**

I. Introduction

High-performance, cloud, and enterprise computing environments increasingly use virtualization to improve the utilization efficiency of their computing, storage, and network resources. Applications that execute within virtual machines (VMs) often tend to be data-intensive and latency-sensitive in nature. The I/O and memory requirements of such VMs can easily exceed the limited resources allocated to them by the virtualization layer. Common examples of resource-intensive VM workloads include large database processing, data mining, scientific applications, virtual private servers, and backend support for websites. Often, I/O operations can become a bottleneck due to frequent access to massive disk-resident datasets, paging activity, flash crowds, or competing VMs on the same node. Even though demanding VM workloads are here to stay as integral parts of cluster-infrastructures, state-of-the-art virtualization technology is inadequately prepared to handle their requirements.

Data-intensive and large memory workloads can particularly suffer in virtualized environments due to multiple software layers of indirection before physical disk is accessed. To handle large memory workloads, developers often implement domain-specific *out-of-core* computation techniques [1] that juggle I/O and computation. But these techniques do not overcome one fundamental limitation – the VM's working set cannot exceed the memory within a single physical machine. Further, many recent large-scale web applications, such as social networks [2], online shopping, and search engines, exhibit little spatial locality, where servicing each user request requires access to disparate parts of massive datasets. To compound the problem further, each user query could be processed by multiple tiers of software, adding to the cumulative I/O latency at each tier. For instance, Amazon [3] processes hundreds of internal requests to produce a single HTML page. Simply buying specialized large-memory machines [4] is also not viable in the long-term because cost-per-gigabyte of DRAM tends to increase non-linearly, making these machines prohibitively expensive to both acquire and maintain. Thus low-latency, and possibly locality-independent, I/O to massive datasets is proving to be a critical requirement for new class of cluster-based applications.

This paper presents MemX – a transparent and reliable distributed system that virtualizes cluster-wide memory for data-intensive and large memory VM workloads. MemX is designed to aggregate multiple terabytes of memory from different physical nodes into one or more *virtualized memory pools* that can be accessed over low-latency high-throughput interconnect (such as 1Gbps, 10GigE, or Infiniband). This virtualized memory pool is then made available to unmodified VMs for use as either (a) a low-latency in-memory block-device that can be used to store large-datasets, or (b) a swap device that reduces the execution times for memory-hungry workloads. MemX offers a number of benefits:

- VM workloads that process large datasets can perform low-latency I/O over virtualized cluster-wide memory;
- VMs can transparently execute very large memory applications that require more DRAM than physically present in the host machine;
- Existing applications do not require any modifications such as the use of special APIs, libraries, recompilation, or relinking;
- MemX reduces the effective memory usage of the cluster by de-duplicating pages having identical contents;
- MemX works seamlessly with the live migration of data-intensive VMs by eliminating the need to migrate part of their memory footprint resident on other nodes.

Prior approaches to overcome the disk I/O bottleneck have examined the use of memory-resident databases [5], [6] and caching [7], [8] techniques. Gray and Putzolu [9] predicted in 1987 that "main memory will be begin to look like secondary storage to processors and their secondary caches". Recently Facebook announced [2] that it uses `memcached` [7] – a distributed in-memory key-value store – to cache results of frequent database queries.

A simple view of the role of MemX is that it aims to accelerate the above trend by virtualizing cluster-wide

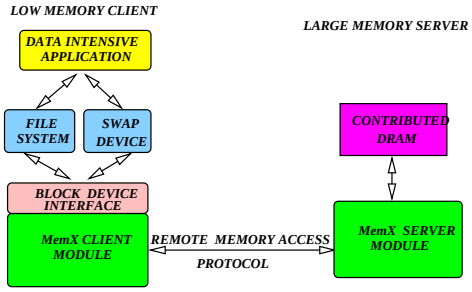

Figure 1. Basic architecture of MemX.

memory as a massive low-latency storage device for large datasets. The key reason why MemX is feasible today as a solution for cluster-wide memory virtualization is that low-latency multi-gigabit network fabrics are becoming commoditized. Interconnects such as Gigabit, 10GigE, and Infiniband offer high throughput (up to 40Gbps) and low-latency (as low as 5μs). Although evaluations in this paper are over 1Gbps Ethernet, MemX itself is agnostic to the networking technology and can scale as both network latency and bandwidth improve. Our current MemX testbed hosts 1.25 Terabyte collective memory. Our prototype is implemented in the Xen [10] environment, but the techniques are easily portable to other virtualization platforms. We compare and contrast different modes in which MemX can operate, namely, (1) MemX-VM within individual VMs, (2) MemX-DD within a common driver domain shared by multiple VMs, and (3) MemX-VMM within the hypervisor, also called Virtual Machine Monitor (VMM), for supporting full virtualization of unmodified operating systems. Using several I/O intensive and large memory benchmarks, we show that applications achieve significant performance speedups using MemX when compared against virtualized disks.

II. Basic Architecture of MemX

Figure 1 shows the basic architecture of MemX. which allows VMs in a cluster to pool their memory in a distributed fashion. Two main components of MemX are the *client* and the *server* modules. Any VM in the cluster can switch into the role of either a client or a server, depending upon its memory usage. A VM that is low on memory can switch to client mode, whereas a VM (or physical machine) with excess (unused) memory can switch to server mode. Clients communicate with servers across the network using a MemX-specific protocol, called the *remote memory access protocol* (RMAP). Both client and server components execute as transparent kernel modules in all configurations.

[Client Module] The client module provides a *virtualized block device* interface to the large dataset applications executing on the client VM. This block device can either be configured as (a) a low-latency volatile file-system for storing large datasets, or (b) a low-latency primary swap device, or (c) memory-mapped I/O space for large memory application. To the rest of the VM, the block device looks like 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. 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 wired LAN, irrespective of the spatial locality. The client module also contains a small bounded-sized write buffer to quickly service write I/O requests. The next section describes how the client module can be configured to operate in exclusive mode (within a VM) or in shared mode (within a driver domain) with different tradeoffs.

[Server Module] A server module stores pages in memory for any client across the LAN. Servers broadcast periodic resource announcement messages which the client modules use to discover the available memory servers. Servers also include feedback about their memory availability and load within both resource announcement as well as regular page transfers to clients. When a server reaches capacity, it declines to serve any new write requests from clients, which then try to select another server. Like the client module, server can also operate in either exclusive mode (within a VM) or in shared mode (in a driver domain). In both modes, pages hosted by one server can be migrated *live* to another server, without interrupting client execution.

[Remote Memory Access Protocol (RMAP)] Client and server modules communicate using a custom-designed layer-2 reliable datagram protocol, called RMAP. RMAP is a lightweight protocol that includes the following features: (1) reliable message-oriented communication, (2) flow-control, and (3) fragmentation and re-assembly. While clients and servers could technically communicate over TCP or UDP, this choice comes burdened with unwanted protocol processing overhead. 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, since MemX is meant for use within a single subnet. We also considered the use of the more recent SCTP (Streaming Control Transmission Protocol), but decided in favor or developing RMAP for a couple of reasons. First, SCTP requires the use of socket interface, which significantly delays the processing of received packets due to scheduling and context switching overheads. In contrast, RMAP registers a custom receive handler with the device driver and performs most receive processing in the kernel's interrupt context (or softirq) allowing for very low latency response times. Secondly, we observed in our experiments that maximum throughput obtained with Linux kernel's current implementation of SCTP is around 300Mbps, which was clearly inadequate for our needs. Thus

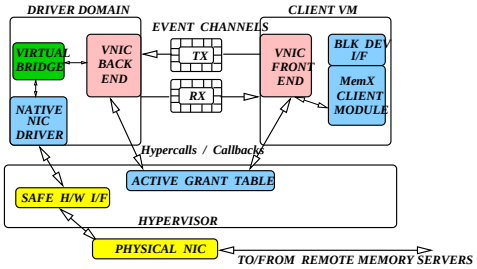

Figure 2. MemX-VM mode: MemX client operates within the client VM.

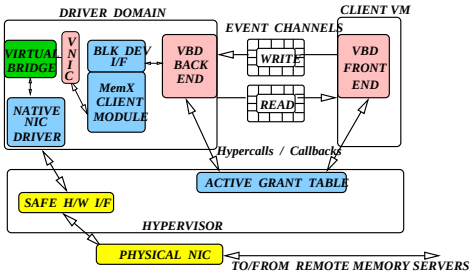

Figure 3. MemX-DD: A shared MemX client within a privileged driver domain multiplexes I/O requests from multiple guests.

RMAP completely bypasses the TCP/IP protocol stack and communicates directly with the network device driver. A fixed-size transmission window is maintained to control the transmission rate. 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/re-assembly 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.

III. Modes of Operation

In this section, we describe the various configurations in which MemX can operate and compare their relative merits.

A. MemX-VM: MemX Client Module in VM

In order to support memory intensive large dataset applications within a VM environment, the first design option is to place the MemX client module within the guest OS in the VM. This option is shown in Figure 2. On one side, the client module exposes a block device interface for large memory applications within the VM. On the other side, the MemX client module communicates with remote MemX servers via a virtualized network interface (VNIC). VNIC is an interface exported by a special VM called the *driver domain* which has the privileges to directly access all I/O devices in the system. (Driver domain is usually synonymous with Domain 0 in Xen).

The VNIC interface is organized as a split device driver consisting of a *frontend* and a *backend*. The frontend resides in the VM and the backend in the driver domain. Frontend and backend communicate with each other via a lockless producer-consumer ring buffer to exchange *grant references* to their memory pages. The grant reference is essentially a token that one VM gives to another co-located VM granting it the permission to access/transfer a memory page. A VM can request the hypervisor to create a grant reference for any page of its memory. The primary use of the grant references in device I/O is to provide a secure communication mechanism between an unprivileged VM and the driver domain so that the former can receive indirect access to hardware devices, such as network cards and disk. The driver domain can set up a DMA-based data transfer directly to/from the system memory of a VM rather than performing the DMA to/from driver domain's memory with the additional copying of the data between VM and driver domain. The grant table can be used to either *share* or *transfer* pages between a VM and driver domain depending upon whether the I/O operation is synchronous or asynchronous in nature.

Two ring buffers 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 transfer with the backend for every packet received or transmitted. All VNIC backends in the driver domain can communicate with the physical NIC as well as with each other via a virtual network bridge. Each VM's VNIC is assigned its own MAC address whereas the driver domain's VNIC uses the physical NIC's MAC address. The physical NIC itself is placed in promiscuous mode by the driver domain to enable the reception of any packet addressed to any of the local VMs. The virtual bridge demultiplexes incoming packets to the target VM's backend driver.

Some of the sources of overhead in the MemX-VM architecture are as follows. Due to the nature of the split-driver VNIC architecture, MemX-VM configuration requires every network packet to traverse across the domain boundary between the VM and the driver domain. In addition network packets must be multiplexed or demultiplexed at the virtual network bridge. Additionally, the client module has to be separately loaded in each VM that might potentially execute large memory applications. Finally, each I/O request is typically 4KB (or sometimes 8KB) in size, whereas most typical Ethernet hardware uses a 1500-byte MTU (maximum transmission unit), unless the underlying network supports jumbo frames. Thus the client module must fragment each 4KB write request into (and reassemble a complete read reply from) at least 3 network packets. Each fragment needs to cross the domain boundary to reach the backend. Depending upon the memory management mechanism in the VM, each fragment may consume an entire 4KB 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.

B. MemX-DD: MemX Client Module in Driver Domain

A second design option, shown in Figure 3, is to place the MemX client module within the driver domain (Domain 0). Each VM can then be assigned a split virtual block device (VBD) interface, with no MemX specific configuration or changes required in the guest OS. The frontend of the split VBD resides in guest and the backend in the driver domain. The frontend and backend of each VBD communicate using a producer-consumer ring buffer.

On one side, the MemX client module in driver domain exposes a block device interface to rest of the driver domain. On the other side, the MemX client module communicates with the physical network card. The back-end of each split VBD is configured to communicate with the MemX client through a separate block device (/dev/memx{a,b,c}, etc). Any VM can configure its split VBD as either swap device for transparent cluster-wide paging, or a file system for low-latency storage. Synchronous I/O requests, in the form of block read and write operations, are conveyed by the split VBD interface to the MemX client module in the driver domain. The client module packages each I/O request into network packets and transmits them asynchronously to remote memory servers using RMAP.

Note that, unlike in MemX-VM, network packets no longer need to pass through a split VNIC architecture (although packets still need to traverse software bridge in the driver domain). Consequently, while client module may still need to fragment a 4KB I/O request into multiple network packets to fit the MTU requirements, each fragment no longer needs to occupy an entire 4KB buffer. As a result, only one 4KB I/O request needs to cross the domain boundary across the split VBD driver, as opposed to three 4KB packet buffers in MemX-VM. Further, the MemX client module can be inserted once in the driver domain and be shared among multiple guests.

However, unlike MemX-VM, MemX-DD does not currently support seamless migration of live VMs using remote memory. This is because part of the VM'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 are investigating extensions to live VM migration to transfer this internal state as well.

C. MemX-VMM: Expanding Guest-physical Address Space

The previous two options assume that the VM is running a para-virtualized operating system that performs its I/O operations through a split-driver interface (either VNIC or VBD). This section describes an additional option – MemX-VMM – that allows MemX to support unmodified operating systems in VMs, i.e., those executing in *full-virtualization* mode. When a VM boots up, the virtual machine monitor (VMM or hypervisor) presents the VM with a logical view of physical memory called the *guest-physical address space* (GPAS). Traditional virtualization architectures, such as Xen and VMWare, do not permit the GPAS size to exceed the actual DRAM in the physical machine. MemX-VMM, on the other hand, presents the VM with a larger GPAS than the actual DRAM in the local machine, making the guest OS believe that it has a large amount of memory at boot time. Of course, not all GPAS memory can reside on the local machine. Thus an actively used subset (the working set) of the GPAS is stored locally whereas rest of the GPAS is stored in cluster-wide memory. It then becomes the VMM's task to perform this mapping of GPAS partly into the local memory and partly into the cluster-wide memory.

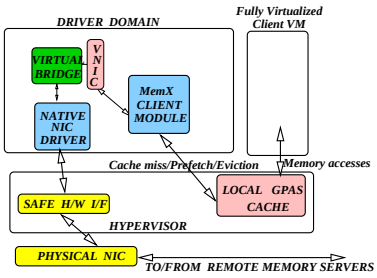

Figure 4. MemX-VMM supports GPAS larger than local DRAM for fully virtualized VMs.

In order to provide the illusion of a larger GPAS, MemX-VMM utilizes a *shadow-paging* mechanism, which works as follows. The VMM marks the memory containing the VM's page-tables as read-only. Thus any write accesses by a VM to page-table memory is intercepted by the VMM. The VMM maintains another table mapping the guest-physical page frame numbers (GFN) in GPAS to machine-physical page frame numbers (MFN) – also called the *G2M table*. Depending on whether a page resides in local memory or in the network, MemX-VMM marks the corresponding entry in the G2M table as being locally resident or not.

The VMM combines the two tables – VM's page tables and the G2M table – and dynamically constructs *shadow page tables* that map the virtual page frame numbers accessed by the VM to the corresponding MFNs. During the VM's execution, MemX-VMM transparently intercepts all page-faults generated by the VM. If the page-fault is for a page that resides in cluster-wide memory, we call it a *network page fault*. To service a network page fault, the VMM communicates with a MemX client module that resides in the driver domain. MemX client fetches the page from cluster-wide memory (as in the case of MemX-DD) and returns it to the VMM, which then maps the received page to the VM's address space, marks it as locally resident in the G2M table, and updates the shadow page table with the new mapping. Along with servicing the network page fault, the MemX client module also prefetches a window of pages around the location of the fault in anticipation of their being accessed by the VM in the near future. Thus MemX-VMM treats the subset of GPAS pages in local memory as a *local cache* for each VM. MemX-VMM also periodically evicts infrequently used pages from local memory to cluster-wide

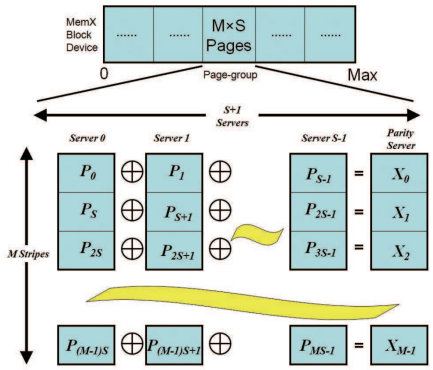

Figure 5. A page-group of $M \times S$ pages is spread over $S+1$ servers and M stripes. The $S+1^{th}$ server stores parity information for each stripe.

memory using a least recently used (LRU) eviction policy.

We have implemented a basic MemX-VMM prototype in the Xen that can execute unmodified Linux VMs with cluster-wide GPAS in full-virtualization mode. We are currently optimizing the performance and improving the stability of our implementation due to issues arising from Xen-specific implementation of the shadow-paging mechanism described above. Additionally, we are also improving support for executing unmodified Windows operating system. Consequently, a more in-depth discussion of implementation and performance evaluation for MemX-VMM is deferred for future work.

D. MemX Server in a VM

Another execution mode, orthogonal to the above three, is to execute the MemX server module itself within a VM. This option provides a significant functional benefit by enabling the server VM to migrate live from one physical machine to another with minimal interruption to client operations. The MemX server VM may be migrated live for any number of reasons, including load balancing the cluster-wide memory usage, or before a physical machine is shut down for maintenance. However this option does introduce additional overheads in network communication compared to a MemX server executing in a non-virtualized setting.

IV. Key Features and Optimizations

A. Scalability

One of the key goals of the MemX system is to provide memory-constrained VMs with access to multi-terabyte cluster-wide memory. However, there is a fundamental challenge in scaling the system to terabyte levels: the MemX client needs to track where in the cluster each page of memory is located. Each page is identified by the pair of (server ID, offset), where server ID is the server holding the page and offset is the logical index of the page in the MemX block device presented by the client module (/dev/memx from earlier discussion). Suppose that MemX client VM uses a simple in-memory hash table to store the location of each of its pages in the cluster, that each page is 4KB in size, and that it takes 6 bytes to store each identifier pair. To track 1 terabyte of cluster-wide memory, the minimum amount of *kernel* memory required just to store the hash table in the client VM is 1.6GB; for 10 terabytes it is 16GB and so on. Clearly, the memory pressure on client can become substantial as MemX scales up.

To alleviate this memory pressure in tracking the page locations, we define the notion of a *page-group*. A page-group, shown in Figure 5, is a group of $M \times S$ consecutive pages in the MemX block device used by the client VM. The individual pages of the page-group are striped across S MemX servers in M logical stripes (rows in the figure). An additional server holds the computed parity for each stripe for fault-tolerance (discussed below in Section IV-B). Now, instead of tracking the location of every page in the cluster, the MemX client tracks only the beginning offset of each page-group and the set of $S+1$ servers holding the pages and parity of the page-group. When the K^{th} page in the page-group is accessed by the VM, the MemX client module simply requests the page from server number $K\%S$ for the page-group. This simple optimization reduces the memory required to track cluster-wide pages by a factor of $M \times S$.

To illustrate with an example, if $M = 256$ and $S = 4$ then each page-group is of size $M \times S = 1024$ pages. The page at block offset 2155 in the block device belongs to page-group number 3 $(2155/1024)$ and is the 107^{th} page $(2155\%1024)$ within the page-group. If servers S0, S1, S2, S3, and S4 hold the data and parity pages of the page-group then page 2155 resides in server S3 (since $107\%4 = 3$).

Note that the MemX block device consists of multiple page-groups and each page-group can be mapped to a different set of $S+1$ servers for load balancing. For instance, page-group 1 could be mapped to server set $\{$S1, S2, S3, S4, S5$\}$, page-group 2 mapped to a different server set $\{$S6, S7, S8, S9, S10$\}$, page-group 3 mapped to yet another server set $\{$S2, S4, S6, S8, S10$\}$, and so on. When a page-group is created, the client module selects the servers responsible for the page-group based on their available capacity.

B. Fault-Tolerance

Being a distributed system, MemX is designed to operate correctly in the presence of various types of faults, including packet loss, server failure, and client failure. Loss of packets (both MemX data and control) is handled through a reliable Remote Memory Access Protocol (RMAP) described earlier in the Section II. Here, we describe how the failure of any server or client is handled.

[Soft-State Refresh] MemX clients and servers track each others' liveness through two soft-state refresh mechanisms – regular packet exchanges and periodic heartbeat messages. First, if three consecutive RMAP packet transmissions from a server to a client VM fail, then the server module marks the client VM as unreachable. If the client remains unreachable after a fixed amount of time, then any client-specific state, which includes unshared memory pages and data structures,

is purged. Similarly, if three successive packet transmissions are lost from a client VM to a server that holds its pages, then the client VM initiates steps to recover the data lost from a server failure (described below). MemX servers also periodically announce their availability in the network through low-frequency broadcast messages. During quiet periods, when there is no data exchange, the absence of three successive announcement messages leads the client VMs to initiate steps to recover from server failure.

[RAID-like recovery from server failure] It is particularly important for client VMs that they recover any lost data in the event that any server holding its pages dies. MemX client VMs employ a RAID-like recovery mechanism to tolerate single-server failures. Each page-group (described earlier in Section IV-A and Figure 5) is mapped by the client VM over $(S+1)$ servers, where S servers hold data pages and one server holds parity pages. A *page stripe* is defined as the set of pages P_i such that i/S is the same (using integer division). For example, if $S = 4$ then pages P_0, P_1, P_2, P_3 belong to the same stripe. Similarly, P_4 through P_7 are in the same stripe, and so on. Obviously, pages in same stripe reside in different servers. The parity block X_k for the k^{th} stripe resides in the $(S+1)^{th}$ server. Upon failure of the i^{th} server in a page-group , the client VM recovers the lost data pages for every stripe by performing a bitwise XOR over the remaining pages in other S servers. The recovered pages are then stored in a new server that does not already hold any data or parity pages of the page-group (assuming there are more than S MemX servers in the network and there is sufficient free space). During the recovery phase, read/write operations cannot be performed on any page-group that is affected by the failed server. Optionally, the affected I/O operations can be temporarily stored in the client VM till the recovery completes, although at the expense of transient memory pressure on the client.

Note that, unlike traditional disk-based RAID systems, all MemX servers do not need to have the same memory capacity, since the client VM has the flexibility of mapping different page-groups (and their parity) to different sets of $S+1$ servers. This flexibility also allows the parity data to be distributed over multiple MemX servers (at the granularity of page-group) and prevents any single server from becoming a bottleneck for parity I/O. Also, though handling failure of more than one server at a time is more complex, technically it can be accomplished by extending the above scheme using principles from RAID-6 and higher.

C. De-duplication

De-duplication refers to eliminating redundancy by storing only one copy of pages that have identical content. There are two approaches to de-duplication.

[Local De-duplication] In this mode, each MemX server performs de-duplication across its local pages. When a MemX server receives a new page of data from a client VM, it computes a 64-bit hash key over the page contents (essentially a simple XOR over successive 64-bit words). This hash key is used to search a hash table stored at the server. Each entry in the hash table consists of a 64-bit hash key computed from page contents, and the corresponding page frame number as hash value. Given the hash key, if an entry is found, it means that a page corresponding to the key already exists at the server with *possibly* the same contents as the incoming page. MemX server then proceeds with a bitwise memory comparison of the contents of these two pages. If they match, a reference to the existing page is stored and the incoming page is discarded. A share-count associated with each page indicates the extent of sharing for each page stored in the MemX server. If no matching entry is returned or if the incoming page's contents are not identical to any existing page, then the incoming write is committed to a freshly allocated page and a new hash key is inserted into the hash table. If a MemX server receives a write request to a page which is presently being shared, then the share-count is decremented, a new memory page is allocated to store the incoming write, and its hash key is computed and stored. The entire sharing is completely transparent to the client VM.

[Global De-duplication] An alternative to the local de-duplication described above is to search for a matching page over all the pages stored cluster-wide. This can potentially lead to even greater reduction in memory usage. One way to implement global sharing could be in a client-driven fashion. Before selecting a server to host a new page, a MemX client VM could potentially query all servers for matching hash keys and send the new page to those servers having matching keys one at a time until a server with an exact match is found. One could reduce the overhead of having to send the page to multiple MemX servers by computing multiple hash keys for each page (using different hash functions), and selecting the server at which all hash keys match. Another approach for implementation is that the client could pick one MemX server and offload the key search and page matching overhead to that server, which might potentially be less loaded than the client VM itself. However, neither of the above two implementation options for global sharing is compatible with the notion of page-groups (described in Section IV-A). Thus at the moment we have implemented only the local sharing mode which has simpler logic, less overhead, and reasonable performance.

D. Live VM and Page Migration

Most virtualization technologies, such as Xen, VMWare and KVM, support the ability of VMs to migrate live from one physical machine to another with minimal downtime, even as the VM workloads continue to execute [11]. Both MemX server and client VMs allow low memory VMs to continue using the cluster-wide memory even as the respective VMs are live migrating within the network. Mi-

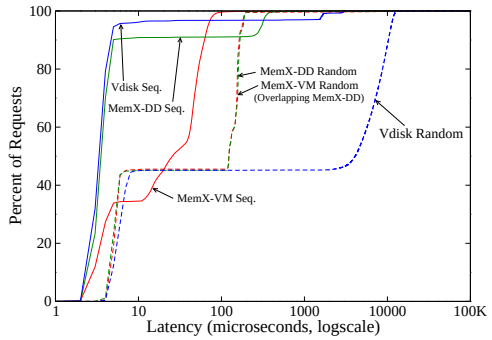

Figure 6. Sequential/random read latency distributions.

grating a VM in MemX-VM mode has the additional benefit that only the memory state within the client VM needs to be migrated. Any memory pages stored by the client on other MemX servers do not need to move; they remain accessible by the client VM even from its new physical host. One could also migrate a VM hosting a MemX server for various reasons, such as to perform load balancing between physical machines, or before shutting down the physical machine hosting the server VM for maintenance. Additionally MemX server includes the ability to migrate all the client pages it is hosting to another MemX server in the network. This could be done at the administrator's discretion, such as before the MemX server is shut down.

V. Performance Evaluation

We now evaluate the performance of the different variants of MemX. Our testbed consists of 32 nodes. Of these, 15 have each 70GB memory and dual quad-core Intel Xeon E5520 2.25 Ghz CPUs, whereas the rest have each 16GB memory and dual quad-core AMD Opteron 2376 CPUs. The network interfaces consist of a mix of onboard nVidia nForce Gigabit Ethernet, onboard Broadcom Gigabit Ethernet, and Intel 82546GB Dual Port Gigabit Ethernet cards. Nodes acting as MemX servers can collectively provide a maximum of 1.25TB of effectively usable cluster-wide memory. Our experiments use Xen 3.3.1 and Linux 2.6.18.8. We use a range of memory sizes for the client VMs from 512MB to 8GB. The client module is implemented in about 2600 lines of C code and server module in about 1600 lines, with no changes to the core Linux kernel. Virtualized disk in any experiment refers to a virtual block device (VBD) exported to the VM from the driver domain.

A. I/O Latency Comparison With Virtual Disk

We compare MemX-DD and MemX-VM against virtual disk in terms of I/O latency by measuring the round trip time (RTT) for a single 4KB read request from a MemX client to a server. RTT is measured in kernel using the on-chip time stamp counter (TSC). This is the latency that the I/O requests from the virtual file system or the swap daemon would experience. MemX-DD provides an RTT of 95μs, followed closely behind by MemX-VM with 115μs. The virtualized disk base case performs as expected at an average 5.3ms. These performance figures show that accessing the memory of remote machine over the network is about two orders of magnitude faster than from local virtualized disk. The split network driver architecture introduces an overhead of another 20μs in MemX-VM over MemX-DD.

Figure 6 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 (around 160μs) than with disk (around 9ms). We observe almost overlapping curves with MemX-DD and MemX-VM configurations for random reads, due to very negligible difference between their remote memory access latencies. Sequential read latency distributions are similar for MemX-DD and disk primarily due to filesystem prefetching. Sequential read latency distribution for MemX-VM is higher by a maximum of few tens of microseconds. RTT distributions for buffered write requests (not shown) are similar for MemX and disk, mostly less than 10μs due to write buffering. Note that these RTTs are measured from the user level, which adds a few tens of microseconds to the kernel-level RTTs. We expect smaller latencies in a cluster with 10GigE or Infiniband.

B. Bonnie++ Disk I/O Benchmarks

We compare the bandwidth of virtual disk with MemX-VM and MemX-DD using Bonnie++ [12] benchmark suite. For this experiment, we used five MemX servers collectively providing 70GB of memory. We run Bonnie++ experiments in a VM which has 2GB of memory and two virtual CPUs (VCPUs). Figure 7 shows the comparison of MemX with virtual disk for 40GB block and character I/O experiments. In the block experiment we carry out a block read/write operations with 4K chunk size, for which MemX-DD achieves close to the peak network bandwidth of 1Gbps. Character write experiment, which writes to the disk byte by byte, performs worse than block write as it involves large number of system calls. In spite of system call overhead, MemX-DD does twice as well as virtual disk for character write experiments. Virtual disk almost matches the performance of MemX for character read, because reading a character doesn't result in disk I/O once the page is cached. Rewrite experiment of Bonnie++ reads the data of specified chunk size, modifies it and writes it back to the disk. Here virtual disk performance deteriorates further, as rewrite involves twice as many number of disk I/Os as normal reads or writes.

C. Application Speedups

We now compare the execution times of a few large memory applications using MemX versus virtualized disk. Figure 8 shows the performance of a Quicksort application that sorts increasingly large arrays of integers. We also include a base case plot for pure in-memory sorting using a vanilla-Linux 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.

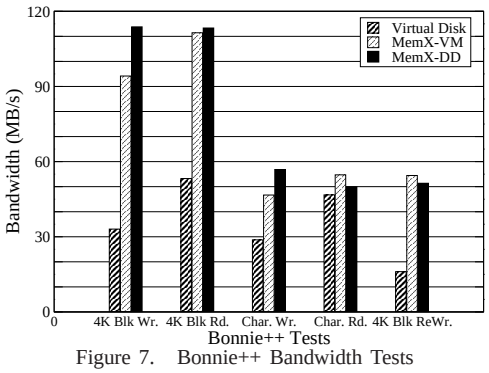

Figure 7. Bonnie++ Bandwidth Tests

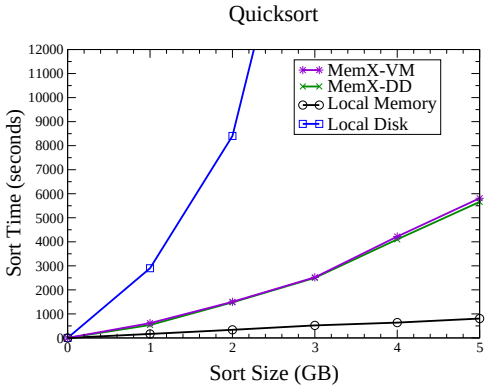

Figure 8. Quicksort execution times vs. problem size.

The sortings using MemX-DD and MemX-VM however finished within 100 minutes for 5GB problem size, the distinction between the two modes being very small. Also note that the performance is still about 3 times faster when using local memory, than with MemX, and consequently there is room for improvement with the use of faster, lower-latency interconnect. Table I lists the execution times for different applications and problem sizes including (1) Sysbench[13] online transaction processing benchmark, (2) Quicksort on a 15GB and 5GB dataset, (3) ray-tracing based graphics rendering application called POV-ray [14], and (4) TPC-H decision support benchmark[15]. Again, MemX outperforms virtual disk for each of these benchmarks.

D. MemX vs. iSCSI for Multiple Client VMs

We now evaluate the overhead of executing multiple client VMs using MemX-DD versus iSCSI. In most enterprise clusters, a high-speed backend interconnect, such as iSCSI or FibreChannel, would provide backend storage for guest VMs. To emulate this case, 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 [16] and the initiator software from open-iscsi.org within the driver domain for all the VMs. One of the five machines executed up to twenty concurrent 100MB VMs, 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 9 shows

Appli-cation	Mem Size	Client Mem	MemX-VM	MemX-DD	Virtual Disk
Sysbench	800GB	4GB	400.08 trans/sec	403.42 trans/sec	229.91 trans/sec
Quicksort	5GB	512 MB	96 min	93 min	$>$ 10 hrs
Quicksort	15GB	2GB	189 min	181 min	$>$ 10 hrs
Povray	6GB	1GB	19 min	20 min	$>$ 3 hrs
Povray	13GB	2 GB	42 min	50 min	$>$ 6 hrs
TPC-H	3GB	2 GB	195.64 QphH@size	226.14 QphH@size	162.09 QphH@size

Table I
COMPARISONS FOR DIFFERENT APPLICATION WORKLOADS.

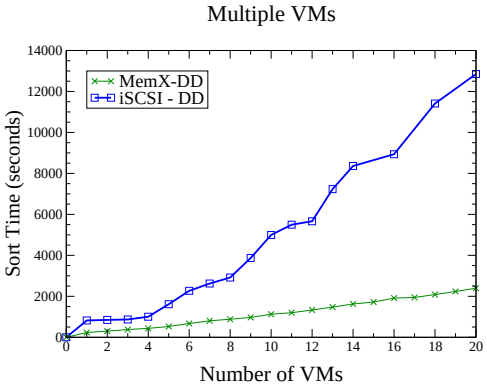

Figure 9. Quicksort execution times for multiple concurrent guest VMs.

that, at about 10 GB of collective memory and 20 concurrent VMs, the execution time with MemX-DD is about 5 times smaller than with iSCSI setup. Thus even with concurrent VMs, MemX-DD provides a clear performance edge.

E. Live VM Migration

MemX-VM configuration has a significant benefit when it comes to migrating live VMs [11] to better utilize resources. Specifically, a VM using MemX-VM can be seamlessly migrated from one physical machine to another, without disrupting the execution of any large memory applications within the VM. First, since MemX-VM 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. We conducted an experiment to compare the live VM migration performance using iSCSI versus MemX-VM. For the iSCSI experiment, we configured a single iSCSI disk as swap space. Similarly, for the MemX-VM 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-VM. While further evaluation is necessary, this experiment points to potential benefits for live VM migration when using MemX.

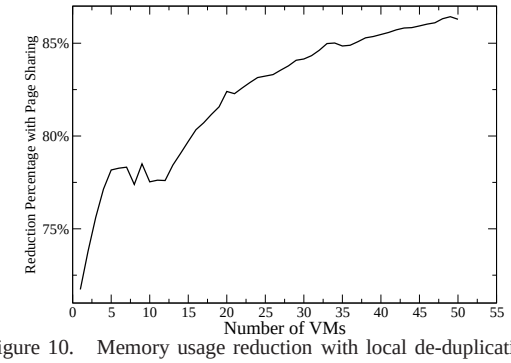

Figure 10. Memory usage reduction with local de-duplication.

Application	W/ Sharing(GB)	W/o Sharing(GB)	Reduction %
VM Creation	59.65	435	86.29
TPC-H	168	443	62.1
Sparse Matrix	36.39	48.44	24.88

Table II
MEMORY USAGE WITH AND WITHOUT DE-DUPLICATION.

F. Local De-duplication

We now evaluate the local de-duplication mechanism. In other words we measure the amount of memory saved by eliminating the storage of duplicate memory pages. We use the MemX-VM mode with 1GB memory and two VCPUs per client VM. We also use 15 servers configured as described above, providing 1.25TB of memory collectively. For the VM creation experiment, we built an ext3 file system on the block device exported by the MemX client module, copied an 8.7GB VM image and corresponding config file to it, and then booted up the VM. We created up to 50 such VMs in our experiments. Similarly, for TPC-H benchmark, 20VMs populated their respective ext3 block devices with 20GB dataset each generated by the TPC-H benchmark. For sparse matrix application, the block device was configured as a swap space. The size of the matrix was 4.8GB with dimensions 20000×20000 and 30% zeros. For the VM creation workload, Table II shows that local de-duplication achieves an 85% reduction in memory usage. Furthermore, the reduction percentage increases with the number of client VMs. We ran VM creation experiment on a range of MemX client VMs from 1 to 50. Figure 10 demonstrates a steadily increasing level of memory savings. With one client VM, the reduction in memory usage is 71.73% whereas with 50 client VMs, the reduction reaches 86.29%. TPC-H and Sparse matrix workloads yield lower, yet significant, memory savings. Once implemented, we expect global de-duplication to yield even higher savings.

G. Comparison With Local Ramdisk

Table III compares the performance of MemX with local Ramdisk to understand the overhead due to network communication. We used a custom I/O microbenchmark inside a VM having memory of size 8GB and 2 VCPUs. For each experiment, a file of size 4GB was read from or written to the block device, with 4KB I/O buffer size. The table shows that Ramdisk provides 5x faster writes and 7x faster reads than MemX over a 1Gbps LAN. Faster 10GigE and Infiniband networks can reduce this performance gap.

	Seq. Write	Rand. Write	Seq. Read	Rand. Read
Ramdisk	337.45	280.29	550.27	305.37
MemX-VM	88.13	58.07	110.06	43.03
MemX-DD	97.63	60.77	110.07	48.42

Table III
I/O PERFORMANCE OF MEMX VERSUS LOCAL RAMDISK (MB/S)

VI. RELATED WORK

To the best of our knowledge, MemX is the first cluster-wide memory virtualization system for I/O intensive and large memory *virtual machines* that is reliable, exploits page-sharing, works with live VM migration, and is designed to scale to multi-terabyte memory capacity. Distributed shared memory (DSM) systems [17], [18] allow distributed parallel applications running on a set of independent nodes to share common data across a cluster. DSM systems often employ heavyweight consistency, coherence, and synchronization mechanisms and may require distributed applications to be written against customized APIs and libraries. In similar spirit, `memcached` [7] provides an API to access a large distributed in-memory key-value store. In *non-virtualized* settings, the use of memory from other machines to support large memory workloads has been explored earlier [19], [20], [21], [22], [23], [8], [24], primarily in 1990s. However, these systems did not address the comprehensive the design and performance considerations in using cluster-wide memory for virtual machine workloads. Additionally, early systems were not widely adopted, presumably due to smaller network bandwidths and higher latencies at that time. A recent position paper [25] also advocates the treatment of cluster memory as a massive low-latency storage, but with focus on developing new APIs for applications. Our work significantly predates this recent effort. Further MemX allows existing applications to scale to cluster-wide memory *without* the need for special APIs. One can also migrate processes [26] or entire VMs [11] from a low-memory node to a memory-rich node. However 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 live VM migration, combining the benefits of both live migration and remote memory access.

Page-sharing has been employed previously in the context of virtual machines for over-subscribing memory within a single physical machine [27], [28]. The idea is to locate memory pages from different co-located VMs that have the same content and share such pages. This allows more VMs to be consolidated within a single physical machine. Memory Buddies [29] takes this notion one step further by migrating VMs to other physical machines where sharing is maximized. The local page-sharing approach in MemX is orthogonal to the above two approaches and can presumably be used in conjunction with them. On the other hand, the global sharing approach in MemX can be considered as a

converse of the Memory Buddies approach in that it migrates individual pages, rather than entire VMs, to those physical machines where page sharing is maximized.

The notion of virtual disk containers over a pool of physical disks was proposed in [30] that could tolerate disk, server, and network failures. In the domain on remote memory, the RMP system [22] also proposed a RAID-like mechanism for reliability in a non-virtualized remote-memory paging system. While similar in spirit, our design of reliability in MemX differs by using the notion of *page-groups* which is critical for scaling to terabyte memory. Striping and parity computation in MemX are performed within the granularity of page-groups, rather than the entire cluster-wide memory, which reduces memory pressure at the client VMs and speeds up recovery from server failures.

VII. Conclusions

We presented the design, implementation, and evaluation of the MemX system that virtualizes cluster-wide memory for data-intensive and large memory VM workloads. Large dataset applications using MemX do not require any specialized APIs, libraries, or any additional modifications. MemX can operate as a kernel module within an individual VM (MemX-VM), or in a shared driver domain (MemX-DD), or within the hypervisor (MemX-VMM) providing different tradeoffs between performance and functionality. Detailed evaluations of our MemX prototype using a number of benchmarks over 1Gbps Ethernet show that I/O latencies are reduced by an order of magnitude and that large memory applications speed up significantly when compared to virtualized and iSCSI disk. This performance will scale for lower-latency higher-bandwidth interconnects, such as 10GigE and Infiniband. We expect that MemX will accelerate the development of future cluster-based applications that will come to expect almost-constant low-latency access to massive datasets as a norm.

References

[1] J. S. Vitter, "External memory algorithms and data structures: dealing with massive data," *ACM Comput. Surv.*, vol. 33, no. 2, pp. 209–271, 2001.

[2] Scaling memcached at Facebook, *http://www.facebook.com/note.php? note_id=39391378919*.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[4] Cray Inc., "Cray XT4 and XT3 Datasheet http://www.cray.com/downloads/cray_xt4_datasheet.pdf."

[5] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. on Knowl. and Data Eng.*, vol. 4, no. 6, pp. 509–516, 1992.

[6] D. DeWitt, R. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, "Implementation techniques for main memory database systems," in *Proc. of ACM SIGMOD International Conference on Management of Data*, 1984.

[7] Memcached, *http://memcached.org/*.

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

[9] J. Gray and F. Putzolu, "The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time," *SIGMOD Rec.*, vol. 16, no. 3, pp. 395–398, 1987.

[10] 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 *Proc. of ACM Symposium on Operating Systems Principles, Bolton, NY, USA*, 2003, pp. 164–177.

[11] 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 Network System Design and Implementation*, 2005.

[12] R. Coker, "Bonnie++ disk benchmark http://www.coker.com.au/bonnie++/."

[13] Sysbench, "http://sysbench.sourceforge.net/index.html."

[14] POV-Ray, "http://povray.org/."

[15] TPC-H Benchmark, "http://www.tpc.org/tpch/."

[16] iSCSI, *http://iscsitarget.sourceforge.net/*.

[17] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets, "Cashmere-VLM: Remote memory paging for software distributed shared memory," in *Proc. of Intl. Parallel Processing Symposium,*, 1999.

[18] 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*, vol. 29, no. 2, pp. 18–28, Feb. 1996.

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

[20] M. Feeley, W. Morgan, F. Pighin, A. Karlin, and H. Levy, "Implementing global memory management in a workstation cluster," *Operating Systems Review*, vol. 29, no. 5, pp. 201–212, 1995.

[21] T. Anderson, D. Culler, and D. Patterson, "A case for NOW (Networks of Workstations)," *IEEE Micro*, vol. 15, no. 1, pp. 54–64, 1995.

[22] E. Markatos and G. Dramitinos, "Implementation of a reliable remote memory pager," in *USENIX Annual Technical Conference*, 1996.

[23] M. Flouris and E. Markatos, "The network RamDisk: Using remote memory on heterogeneous NOWs," *Cluster Computing*, vol. 2, no. 4, 1999.

[24] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," in *IEEE Cluster Computing*, Sept. 2005.

[25] J. Ousterhout, P. Agrawal et. al., "The case for ramclouds: scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, 2009.

[26] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration survey," *ACM Comp. Surv.*, vol. 32(3), pp. 241–299, 2000.

[27] C. Waldspurger, "Memory resource management in VMware ESX server," in *Operating System Design and Implementation*, Dec 2002.

[28] D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *Proc. of Operating Systems Design and Implementation, OSDI*, 2008.

[29] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. Corner, "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," in *Proc. of VEE*, 2009.

[30] E. Lee and C. Thekkath, "Petal: distributed virtual disks," in *Proc. of Intl. Conf. on Architectural support for programming languages and operating systems (ASPLOS)*, 1996, pp. 84–92.