Executing MPI Programs on Virtual Machines
in an Internet Sharing System [*]

Zhelong Pan[1], Xiaojuan Ren[1], Rudolf Eigenmann[1], Dongyan Xu[2]

[1]Purdue University
School of Electrical and Computer Engineering
West Lafayette, IN 47907 USA
{zpan,xren,eigenman}@purdue.edu

[2]Purdue University
Department of Computer Science
West Lafayette, IN 47907 USA
dxu@cs.purdue.edu

Abstract

Internet sharing systems aim at federating and utilizing distributed computing resources across the Internet. This paper presents a user-level virtual machine (VM) approach to MPI program execution in an Internet sharing framework. In this approach, the resource consumer has its own operating system running on top of, and isolated from, the operating system of the resource provider. We propose an efficient socket virtualization technique to optimize VM network performance. Socket virtualization achieves the same network bandwidth as the physical network. In our LAN environment, it reduces the latency overhead from 172% (using existing TUN/TAP technique) to 35.6%. Performance results on MPI benchmarks show that our virtualization technique incurs small overhead compared with the physical host platform, while gaining in return a higher degree of guest isolation and customization. We also describe the key mechanisms that allow the employment of VMs in an existing Internet sharing system.

1 Introduction and Motivation

Significant computing power can be aggregated from machines connected by the Internet. Systems, such as BOINC [1], have begun to exploit this power by encouraging Internet users to contribute their idle computing resources. Computation-intensive jobs, such as those in SETI@home [10], can achieve high computing capacity by gathering idle CPU cycles. Our Internet sharing system, *iShare* [17], is one such system. It employs

Peer-to-Peer (P2P) techniques for publishing, discovering and federating Internet resources such as CPU cycles, software, and disk space. IShare's initial focus was to manage these resources for applications that need single-node platforms. In this paper, we present techniques that allow distributed MPI programs to execute in an Internet sharing environment like iShare.

Meanwhile, many users are still reluctant to participate in Internet sharing systems, due to lack of safeguards between resource providers and consumers. Resource providers are concerned with the potential harm inflicted by Internet sharing systems, especially when the installation or use of the system requires administrator (or root) privileges. Moreover, resource providers distrust guest jobs, because erroneous or malicious applications submitted by a consumer may harm the provider's host machine. On the other hand, resource consumers distrust other guest jobs on the same host machine, because these jobs may peek, forge or terminate their own jobs.

Recently, virtual machine (VM) techniques have been proposed to address some of the above issues [7]. Virtual machines enhance system security by providing strict isolation between the host and guest jobs. The resource provider starts a VM, which provides a complete machine and OS illusion, for a consumer. Different consumers use different VMs running on top of the physical host machine. In this way, each job is executed in a sandboxing environment – a confined execution environment for running untrusted programs.

In this paper, we build on the VM technique to provide an environment for running distributed, message-passing (MPI) programs in Internet sharing systems. We deploy VMs completely at user level, using the User-Mode Linux (UML) [5] VM platform. In addition, we implement an efficient, secure VM network,

which can be easily set up without any network administration effort. Our VM solution works in concert with techniques to discover and select machines on the Internet for an MPI program in an environment of fluctuating resources.

The key challenge we address is to provide an efficient, secure VM network. In our design, we assign the same host machine IP address to the VM, and implement the TCP/IP socket functionality for the VM, called *guest socket*, based on the socket functionality of the host machine, called *host socket*. We work on socket functions, because they are widely used in the Internet and in numerous MPI implementations. Multiple VMs on the host use the same IP address, and can be uniquely identified by different ports on the host machine. To be shown in Section 5, communication overhead of our VM network implementation is low. To prevent potential abuse of network resources in VM networks [23], we set security policies to ensure that guest socket communications only happen between specific machines, ports and applications. Our prototype of guest socket functions are developed based on User-Mode Linux (UML).

Another challenge is the management of resources in a non-dedicated cycle-sharing environment, where the availability of resources fluctuates. Our system schedules an MPI program on multiple VMs, each of which is mapped to and started on a physical machine. To ensure reliable MPI program executions, iShare's resource allocation algorithm leverages results of the node availability prediction [19].

Our contributions are summarized as follows:

1. We develop a method for efficient communication between VMs. The key idea is to enable guest socket functions to access host socket functions directly, avoiding overheads that other VM network implementations [5, 13, 21, 23] incur.

2. We enhance security in our Internet sharing system by employing a user-level VM and a VM network approach for running (MPI) programs on remote, untrusted hosts. A number of security policies are proposed to work with our VM networking technique.

3. We install and employ a VM using normal user privileges on a host machine. The setup of the VM network requires no administration effort.

4. We develop new resource management methods that complement the VM operations in our Internet sharing system, iShare. These methods enable reliable guest job execution in an environment with fluctuating computing resources.

The remainder of this paper is organized as follows. Section 2 provides background about the UML virtual machine platform, on which our techniques are based. Section 3 presents our VM networking solution for message-passing programs. Section 4 gives an overview of using VMs in a real-world Internet sharing system. Section 5 presents experimental results of our VM solution. Section 6 discusses related work. Section 7 concludes this paper.

2 Background: User-Mode Linux

User-Mode Linux (UML) [5] adapts the Linux kernel, so that it can execute as a set of host Linux processes. A user-space virtual machine (VM) using this adapted Linux kernel can be created on an unmodified host OS. The VM uses emulated hardware constructed from services provided by the host. Processes running in the guest OS of the VM see a self-contained environment. They do not have any access to host resources other than those explicitly provided to the VM. Hence, UML provides a sandboxing environment to the guests.

The key technique in UML implementation is its tracing thread, which is a special thread intercepting all system calls from processes in the guest OS. The tracing thread can intercept a guest system call and its arguments, annul the system call, and divert the process into the user-space kernel to be executed. In our system, the tracing thread diverts socket system calls to our implementation of guest socket functions, which will then call host socket functions.

Although the tracing mechanism in UML does not introduce significant overhead to computation-intensive programs, it does to programs with frequent system calls. Other virtual machine platforms with better performance, for example, VMware [12] and Xen [6], can be adopted for MPI program execution in a networked environment. In this paper, we choose to use UML because of the following three requirements of Internet sharing systems: (1) Internet sharing systems favor a pure user-level solution, with no root or administration privilege required. (2) They should not make significant changes to the host environment, to minimize the impact on the resource provider. (3) The systems aim at providing a realistic guest environment to the resource consumer. Furthermore, consumers may configure their own guest environments, for example, guest OS versions, libraries and software. UML can be installed at user level and it meets these three requirements. The next section describes our user-level solution to enabling efficient VM communication over the Internet.

3   VM Networking in iShare

To use VMs for MPI program execution in an Internet sharing system, a VM network is needed to enable communication between multiple VMs hosted on distributed physical machines. The VM network is expected to achieve the following three goals.

1. Efficiency: It is desirable that VM network performance (latency and bandwidth) be as close as possible to physical network performance.

2. Security: Policies are needed for restricting the power of using the VM network, limiting potential abuse. Network security in its entirety is beyond the scope of this paper.

3. No root privilege requirement: Resource providers should be allowed to offer machines for remote use "as is", without performing installation or configuration that requires administrator privileges. This requirement is a key enabler for a large range of machines to become available for Internet sharing.

There exist several solutions to VM networking, characterized by dynamic IP allocation, network address translation, and link layer virtualization. For the following reasons, these solutions are not well suited for use in message-passing program execution in an Internet sharing environment.

1. With dynamic IP allocation, the virtual network interface of a VM is emulated by using the physical interface of the underlying host, as shown in the bridged connection in VMware [12]. This solution requires dynamic allocation of new Internet IPs for VMs. This is not practical because it could prohibitively inflate the number of IP addresses needed and it also requires system administration tasks.

2. Network address translation, such as the NAT network in VMware [12], is a commonly used technique in border routers and firewalls. It maps IP addresses and TCP/UDP ports on VMs to IP addresses and ports on host machines. These mappings need to be recorded or explicitly added. NAT networking is insufficient, because the mappings have to be created manually to enable incoming traffic [23].

3. Virtual networks, such as VNET [23] and VIO-LIN [13, 21], incur non-trivial performance penalty due to their link layer virtualization approach.

The TUN/TAP VM networking solution suggested by UML belongs to the first category (dynamic IP allocation). In addition to the aforementioned disadvantages, TUN/TAP decreases network bandwidth and increases network latency significantly, to be demonstrated in Section 5.

The key idea in our approach is to realize guest socket functions directly via host socket functions. VMs are connected in the same way as physical machines. Communication between VMs is essentially the same as communication between physical machines. Our solution achieves all the three design goals for VM networking. First, a guest socket system call is essentially a host socket system call without further translation or redirection. The overhead of virtualization is insignificant. Second, VM network communication is based on user-level socket calls on the host and restricted by our security policies. Third, because VMs are connected in the same way as physical machines, no system administration operations (e.g., IP allocation and routing) are needed to set up VM networks. By contrast, the current TUN/TAP solution in UML requires root privileges.

Our user-level solution guarantees that the whole VM system can be installed and operate under a normal user account. It enables resource providers to use their unmodified administrative procedures for managing and providing security for this account.

We will discuss issues involved in our guest socket implementation and security policies in Section 3.1 and Section 3.2, respectively.

3.1   Socket Implementation Issues

In our VM networking technique, socket system calls from a guest process are intercepted by UML's tracing mechanism. These calls are diverted to our socket implementation, which includes the socket system calls and all the file access functions. Because file operations can also be performed on a socket, attention should be paid to system calls on file operations as well. Essentially, these functions invoke the corresponding host socket calls. As guest socket calls invoke host sockets directly, the following issues, not present in the original UML, need to be resolved:

3.1.1   File Descriptor Virtualization

Guest socket operations use guest-domain file descriptors; the kernel (UML) calls host socket functions, using host-domain descriptors, to realize the guest socket operations. However, a file descriptor in the host domain is not the one in the guest domain, due to virtualization.

Our UML kernel maintains a mapping from guest-domain descriptors to host-domain descriptors for live sockets. Whenever a new descriptor is created for a guest socket call, in *socket()*, *socketpair()* or *accept()*, the kernel allocates a guest-domain file descriptor and maps it to the host-domain descriptor, returned from the corresponding host socket function call. Upon guest socket operations, the kernel retrieves host-domain file descriptors from the mapping and passes them to the corresponding host socket function calls. During the operations, the reference counter associated with a guest-domain descriptor is incremented/decremented in the same way as in the original Linux approach. When the counter reaches zero, mostly due to *close()*, both the host-domain descriptor and the guest-domain descriptor will be freed.

3.1.2 Page Fault Manipulation

For socket operations like *send()* and *recv()*, a buffer in the user address space is passed to the UML kernel. Page faults may happen, when the buffer is accessed during the kernel-mode execution. In this situation, the page fault handler of the guest OS should be invoked. However, in UML, by default, the page fault handler in the host OS is invoked.

Similar situations have happened to the original UML, for example, in system call *read()*. UML solves this problem by providing special memory copying utilities, which transfer data between kernel address space and user address space [5]. These utility functions have a code path that gets executed when a fault happens during the memory copy. This code path will emulate the page fault in the user address space, which will then be processed by the guest page fault handler. Our socket implementation makes use of the above mechanism to handle page faults that happen during socket calls.

3.1.3 Blocking Mode Handling

If a guest socket is in blocking mode (so is the corresponding host socket), the socket operations like *send()* and *recv()* should not exit the kernel to the user until the relevant data or buffer is available. However, the blocked host socket call in the kernel would block the entire guest OS, thus the VM would be blocked as well. As a result, the other processes in the guest would be blocked inappropriately, leading to a deadlock in a multi-programming environment.

To avoid such deadlock situation, our implementation checks, before calling the host socket functions, if the data or buffer is available. It relinquishes the processor while this condition is false. In this way,

the host socket function never gets blocked, while the guest socket function enters/leaves the blocking state correctly.

Asynchronous socket programming could be used for the data checking mentioned above. In this solution, signals would be utilized to notify the kernel whether the relevant socket has available data. However, the asynchronous socket is not well implemented in all Linux systems. For example, our experimental system, RedHat 9.0, does not support asynchronous sockets.

We adopt a different solution, periodic polling, in which the socket system call implementation polls the host socket. If the data or buffer is not available, the process that is making the socket call sleeps and yields the processor to other processes. At each process rescheduling action, the kernel *scheduler* polls the pending host sockets and wakes up the corresponding processes upon the occurrences of certain socket events. To minimize the duration of sleep, the idle process periodically polls the sockets as well. When data or buffer is available, the relevant process exits the polling loop to execute the corresponding host socket functions. This approach yields low communication latency, as we will see in Section 5.

3.2 Security Policies

By directly connecting the virtual network interface in a VM to the physical interface of the host, our socket implementation provides an efficient way of accessing network resources. To prevent potential abuse of network resources in VM networks, we set security policies to ensure that guest socket communications only happen between specific machines and ports. We add three security policies to our socket implementation. Essentially, these policies make the VM communication tamper-resistant by restricting the usage of host sockets.

3.2.1 TCP/UDP Sockets Only

Our socket implementation is applied only to Internet domain sockets using TCP or UDP protocols. All other socket types (e.g., Unix domain sockets) are handled by the original UML socket functions. (This is because it is more secure for a guest application to use Unix domain sockets within the guest domain than within the host domain.) The support for TCP/UDP sockets is sufficient for running MPI programs.

3.2.2 Specific Machines Only

VM communication using our socket implementation is limited to a list of machines, which is generated dynamically as described in Section 4. No data can be sent to or received from other machines. This list is passed to the kernel at the time when the VM is initiated. In iShare, this list is generated by the resource allocator, which discovers and schedules machines appropriate for a requested job. The list is neither contained in the guest OS image, nor in the host file system. It only exists in the kernel address space of the VM. This solution allows for flexible VM setup, and also disallows access to the list from the host.

In order to limit network access within the provided list of machines, a hash table is created from the list. The peer machine address of a socket connection or the one specified in *sendto()/recvfrom()* is checked against this table. The socket operation is denied, if the machine address is not found in the table. If the IP addresses of physical machines should not be revealed to the user, a virtual private IP address space could be applied to virtual machines. The kernel would be in charge of the address translation between virtual IPs of the VMs and real IPs of the physical machines.

3.2.3 Specific Ports Only

A range of ports are passed to the kernel during VM initiation. Our socket implementation does not allow the user application to access any port out of this range. By default, we do not allow access to the ports below 1024. As a result, a VM cannot talk to the reserved ports on the physical machines.

In this paper, we use a direct mapping from the port number in the guest address space to the one in the host address space. The port number in the guest address space is used by guest socket calls, while the one in the host address space is used by the host socket calls made by our socket implementation. In addition, a port number translation between the guest and the host could be exploited to support arbitrary ports used by various applications. Different VMs should agree on the same translation mechanism, so that their kernels could talk to each other through the host ports. In this case, the guest might use any ports, mapped to different ones on the host.

4 Resource Management in iShare

Managing the nodes (machines) that participate in the execution of a distributed message-passing program is an important issue in the large-scale Internet sharing system like iShare. Resource providers must be able to maintain autonomy in deciding when and how much machine cycles are available for resource consumers. For programs executed in such an environment, resource availability can fluctuate substantially. Our resource management strategy aims to deliver the best-effort performance of program execution in this situation. To optimize the usage of available resources with minimum interruptions from resource failures (e.g., resource revocation or violation of resource constraints), we take the approach of fault-aware resource allocation. The key idea is to pro-actively allocate the resources delivering better performance, with lower probability of resource failures during the program execution.

This section describes the realization of VM-based MPI execution in the iShare Internet sharing system. We will focus on the resource management subsystem, including the resource publishing/discovery and resource allocation mechanisms.

4.1 Principles of VM-based MPI Execution in iShare

To run an MPI program within VMs, the iShare middle-ware needs to allocate the resources (physical machines and VM images), set the resource configurations for VMs (socket connections, memory size, etc), stage the program binary code into VMs, and start the VMs and the program.

Enabled by the techniques described in Section 3, our solution obeys the following security principles. First, the startup of VMs does not require any root or administrator privilege, providing a normal user account to run iShare software. Second, the root or administrator may easily control the resource used by iShare using common OS utilities. For example, they can easily set disk quotas or limit file or device access. Third, a guest job is executed in a separate VM, reducing the possibility of impacting other jobs on the host. Fourth, after a job finishes, all execution states will be removed from the host.

4.2 Architecture of Resource Management Subsystem

The overall architecture of the resource management subsystem is shown in Figure 1. All participants reside in a Peer-to-Peer(P2P) network, represented by the large circle in Figure 1. The P2P overlay provides the services for resource publishing and discovery. A participant can publish resources by inserting resource meta-data to the P2P overlay, or search for requested resources and start program execution on these resources. Generally, there are three types of
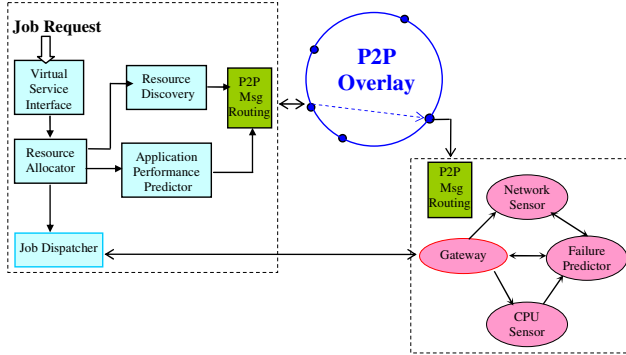
Figure 1. Resource management in iShare. The large circle is the P2P overlay, on which all participating nodes reside. Node A is an iShare server and node B is an iShare client. The boxes present the middle-ware modules on the related nodes.

node functionalities in iShare: clients (resource consumers), servers (resource providers), and persistent storages (repositories for VM images and other data). A participant can set its configurations as any arbitrary combination of these three types.

In Figure 1, nodes A and B show the examples of a server and a client, respectively. On a server (node A), the resource usage is monitored by a *CPU Sensor* and a *Network Sensor*. The *Failure Predictor* forecasts the occurrences of resource failures using monitoring data. External nodes can query the resource usage and fault prediction through the *Gateway*. When a client (node B) requests a program execution via the *Virtual Service Interface*, resource allocation is invoked. The *Resource Allocator* discovers a list of resource candidates in the P2P overlay, and collects resource information on each candidate. With the resource information, the *Application Performance Predictor* estimates the program execution time on each candidate, used for resource selection. The *Job Dispatcher* stages the binary and VMs onto each of the selected resources, and starts program execution.

4.3 Resource Publishing/Discovery

In iShare, resources are organized into a hierarchical structure and mapped into the underlying P2P overlay. In an Internet sharing system, resources, i.e., machines, programs and VM images, can be classified into categories based on their functional features, described by meta-data, which form a tree representing the hierarchical name space, shown on the left side of Figure 2. To avoid the centrality of traditional hierarchical ser-

vices, meta-data are published and inserted into the P2P overlay, by hashing path names in the tree. This leads to the fact that meta-data with similar semantics are likely to be co-located in the P2P overlay, thus reducing the search cost (in terms of the number of nodes searched) for a given query. Figure 2 describes the idea of mapping the hierarchical resource space to the P2P overlay. Detailed designs and evaluations of resource publishing and discovery can be found in [20].

The discovery of idle cycles for running MPI programs involves three steps. First, available physical machines are searched. Second, VM images with matched configurations, e.g., OS version and software customization, are located or created on-demand. Finally, resource usage and resource constraints of the discovered physical machines are checked to make sure that they can start up VMs and run the MPI programs.

4.4 Resource Allocation

Resource allocation chooses the final resource set from the resource candidates returned by resource discovery. To make the decision, minimum job execution time is taken as the cost model, evaluated with a prediction algorithm. The prediction algorithm learns the program execution time by fitting program inputs, dynamic and static resource features, and resource failure probabilities into a local linear regression model [15]. Program inputs are input parameters specified by the user who initiates the execution. Resource features are collected by parsing the resource meta-data and querying the sensors on each resource. Resource failure probabilities are calculated by a Semi-Markov Process model implemented within the monitoring entity in each resource [18].

In the virtual environment, the time required to transfer and start the VM images is a crucial factor in predicting program execution time. In view of the image transfer time, slower machines could be selected if the VM images are already located in their disks or if faster links connect them with the location of image storages.

5 Experimental Results

This section presents experimental results on VM network performance and MPI program performance. [1] We compare with a physical machine and with the
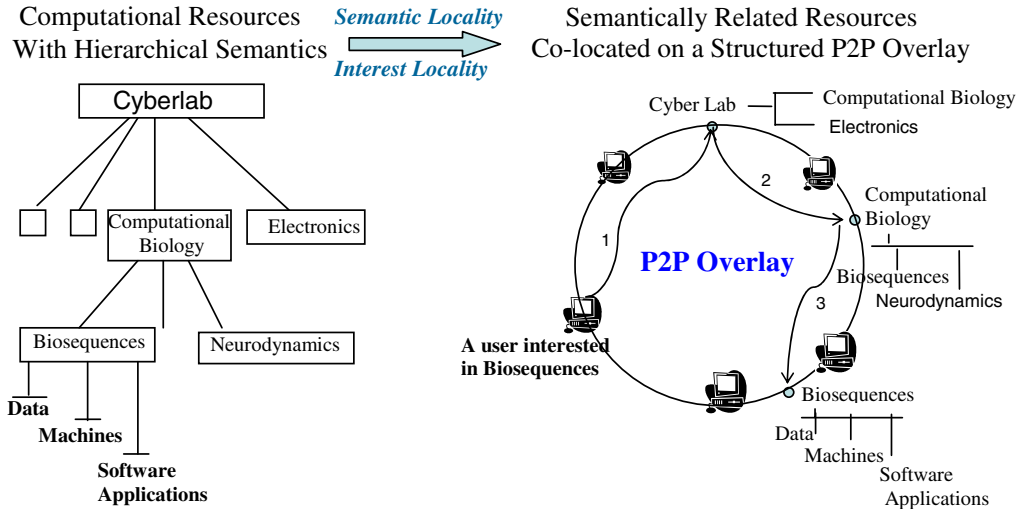

Figure 2. Resource discovery in iShare. The large circle represents a P2P ring, with arrows indicating P2P routing messages to discover resources.

original UML networking solution. We measure effective network bandwidth and latency as well as overall program performance for MPI applications run on the VMs.

5.1 VM Network Performance

To measure the end-to-end network performance (bandwidth and latency), we use two 1.7 GHz Pentium IV machines, connected to a 100 Mbps LAN. The host operating system on both machines is RedHat 9.0. The guest OS is RedHat 7.2. Netperf [11] is used as the benchmark to measure network bandwidth and latency. We focus on TCP performance in this paper, because TCP is used by the MPI implementation in our experiments. A 60-second TCP stream performance test is conducted for measuring network bandwidth, via sending/receiving a bulk of data. To measure network latency, a 60-second TCP request/response test is done by sending a 1-byte request and receiving a 1-byte response. We repeat the tests 100 times and use the average as the final result.

We conduct the experiments under three different settings.

1. *PHY*: Physical host network. This represents the upper bound of the performance that a VM network could achieve.

2. *UML_TAP*: VM network via TUN/TAP. This is the networking solution suggested by UML. The setup of this configuration needs root privileges.

Table 1. TCP stream performance, reflecting effective network bandwidth. (PHY: physical network; UML_TAP: VM network via TUN/TAP; UML_HS: Our VM network solution with host socket support.) Higher throughput (in Mbps) and lower CPU usage are better. The table shows that our network solution achieves the same bandwidth as the physical network, better than TUN/TAP.

	PHY	UML_TAP	UML_HS
Throughput	93.94	64.54	93.94
CPU Usage	33%	100%	91%

3. *UML_HS*: VM network with host socket support. This setting is our extension to the UML kernel. Root privilege is not involved.

Table 1 shows network throughput (in Mbps) and CPU usage obtained in TCP stream tests. Our socket implementation (the UML_HS column) achieves the same throughput as the physical network (the PHY column) at a cost of higher CPU usage. By contrast, the existing TUN/TAP network (the UML_TAP column) achieves only 68% of the physical bandwidth. This is because UML_TAP has already saturated the CPU. In this setting, the network performance of UML_TAP is bounded by the CPU, not by the physical network. Our VM networking implementation with host socket support achieves much better network bandwidth than the TUN/TAP solution.
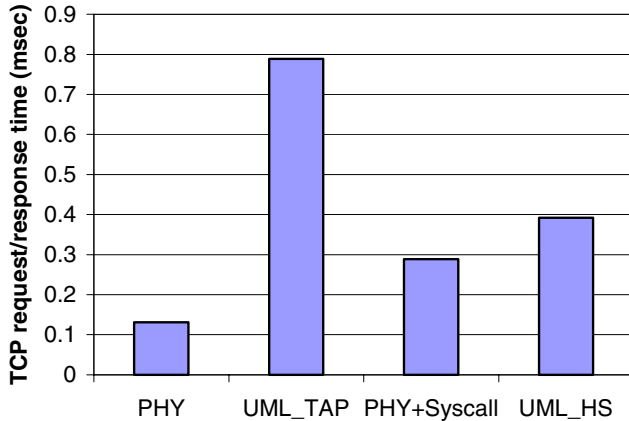
Figure 3. TCP request/response time, reflecting round-trip network latency. Lower bars are better. (PHY: physical host network; UML_TAP: VM network via TUN/TAP; PHY+Syscall: physical network request/response time plus system call overhead introduced by UML's tracing method; UML_HS: Our VM network solution with host socket support.) The figure shows that our network solution drastically reduces the latency, compared with TUN/TAP.

Figure 3 shows the network latency measured in TCP request/response tests. TUN/TAP's latency (the UML_TAP column) is 502% higher than the physical network latency (the PHY column). By contrast, our socket implementation (the UML_HS column) introduces an overhead of 0.261 milliseconds (199%). The percentage of latency overhead in our implementation is less than 40% of the TUN/TAP latency.

It is worth noting that this 0.261-millisecond overhead caused by our network implementation already includes the cost introduced by UML's system call interception. Such interception overhead is inherent in achieving virtualization. In order to measure the interception overhead, we implement a dummy socket call, where the system call returns immediately upon entry. This dummy socket call is invoked in the guest OS 100,000 times, and the average execution time is measured as the overhead per system call introduced by UML. In our experimental setting, the overhead is 0.079 milliseconds per system call. Because there are two socket calls (*send()* and *recv()*) in each request/response pair, we add $0.079 \times 2 = 0.158$ milliseconds to the physical network request/response time. This result is depicted by the PHY+Syscall column in Figure 3. The value represents the real lower bound

of network latency our VM networking solution could achieve.

Compared to this lower bound on VM network latency, our socket implementation introduces an overhead as small as 0.103 milliseconds (35.6%) to the request/response time. (The overhead for TUN/TAP is 172% relative to this lower bound.) We need to point out that, our measurement is done in a fast LAN. In a wide-area Internet environment, this overhead is negligible due to the high latency incurred by the physical network itself.

In summary, the experimental results show that our VM networking solution is efficient in both network bandwidth and network latency.

5.2 Overall MPI Program Performance

We use four 1.5 GHz Pentium IV machines on a 100 Mbps LAN to measure MPI program performance. Table 2 lists the three application benchmarks in the NAS parallel benchmark suite [2], which are used for experiments in this paper. We use the data size of Class A for these benchmarks. MPICH 1.2.6 [9] is installed in the guest OS. SSH on a non-default TCP port is used to start MPI processes in the guest OS. Figure 4 shows the MPI program performance, measured by the speedup relative to program execution on one physical machine.

As indicated by the UML_1 columns in Figure 4, the execution on one original UML virtual machine brings 17% and 27% slowdown to LU and SP (BT's memory requirement exceeds the capacity of one original UML virtual machine.) The equality of the two set of values for UML_1 and UML_HS_1 (one virtual machine with our host socket support) shows that our implementation does not introduce overhead beyond socket calls (there are no socket calls invoked by MPI programs running on a single node).

The results of the UML_HS_4 columns are measured by running the programs on four VMs, each of which resides on a different physical machine. The VM is the UML with our host socket support. BT can be

Table 2. NPB application benchmarks

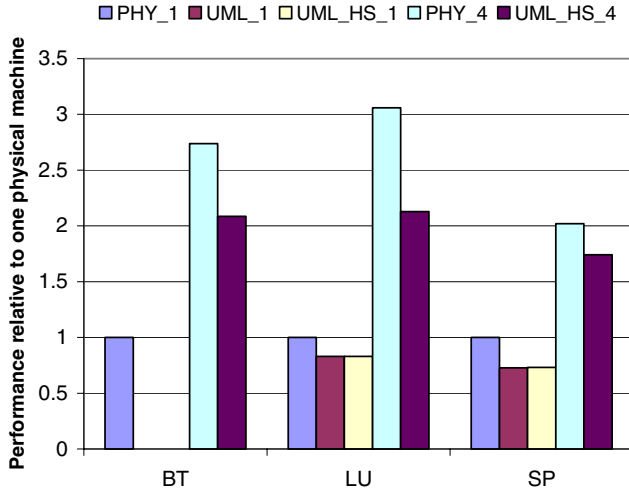|	
BT	Solution of multiple, independent systems of non-diagonally dominant, block tridiagonal equations with a (5×5) block size.
LU	A regular-sparse, block (5×5) lower and upper triangular system solution.
SP	Solution of multiple, independent systems of non-diagonally dominant, scalar, penta-diagonal equations.

Figure 4. MPI program performance relative to one physical machine. The metric is wall-clock_time_on_one_physical_machine / wall-clock_time, which reflects execution speed. Higher bars are better. (PHY_1: one physical machine; UML_1: one virtual machine of original UML; UML_HS_1: one virtual machine of UML with host socket support; PHY_4: four physical machines; UML_HS_4: four distributed virtual machines of UML with host socket support.)

executed under this configuration, since four VMs provide sufficient memory resources. For the other two benchmarks, they run faster on four VMs with the host socket support than on one physical machine. Our solution also achieves a reasonable speedup relative to the execution with one VM.

Comparison of the results for PHY_4 (four physical machines) and UML_HS_4 shows that running MPI programs on four VMs is slower than on four physical machines. Our performance relative to the physical case is 76%, 70% and 86% for BT, LU and SP, respectively. The performance overhead has two sources: (1) UML itself introduces overhead, indicated by the column of UML_1. (2) Our socket communication has larger latency than the physical case, due to the cost of virtualization. This is shown by the UML_HS column in Figure 3.

We conclude that the performance degradation experienced by an MPI application, relative to the speed on the physical host, is acceptable. In return, our VM solution gains a higher degree of guest isolation and customization.

6 Related Work

Many Internet sharing and Grid computing systems have been developed in recent years. Examples are Globus [8] and PUNCH [14, 16]. They provide access to resources within well-defined environments – typically the systems that have fixed installations, requiring non-trivial setup efforts from system administrators. By contrast, the presented techniques and their realization in our iShare system aim to provide a thin, user-level solution to Internet sharing. BOINC [1] provides a tool to share computing resources for the execution of trusted distributed programs, usually in a master-slave fashion. By contrast, our work targets general MPI programs in a system that goes beyond the confines of trusted environments. In [7], virtual machines have been introduced to Grid Computing. Entropia [4] uses virtual machines to provide a sandboxing environment. However, these two systems do not provide solutions to the issue of setting up efficient VM networks.

Providing network accesses to virtual machines is essential to distributed programs. As we have pointed out in Section 3, dynamic IP allocation and network address translation used in virtual machine systems are not well suited for Internet sharing. Several systems have been developed to solve this problem. VNET [23] implements a virtual local area network spreading over a wide area using layer-2 tunneling. VIOLIN [13, 21] designs a virtual private network using UDP tunneling at the user level. Although these two systems design a functional virtual network for VMs, the extra level of indirection by tunneling introduces significant communication overhead, especially for fast local area networks. This problem does not exist in our design, since our VM network uses the physical network directly with an appropriate level of virtualization. Network virtualization in PlanetLab [3] provides a "safe" version of Linux raw sockets for users to send and receive IP packets. However, this design requires modifications to the host operating system, which is not appropriate for large-scale Internet sharing. Our system constructs guest sockets based on user-level host socket functions, incurring no change to the host OS kernel.

7 Conclusion

Internet sharing has the potential of federating a tremendous amount of resources available world-wide. In such a large-scale environment, the protection of both resource providers and consumers as well as the management of dynamic resources are of particular importance. To this end, the contributions presented in

this paper enable message-passing applications to execute in an Internet sharing environment based on virtual machine (VM) techniques. We provide a user-level method for setting up the VM network as well as techniques to reduce the communication overhead incurred by virtual machines. Experimental results show that our techniques achieve the same network bandwidth as the physical network. The network latency overhead is reduced from 172% to 35.6% in a LAN setting. We have also described how these techniques can be integrated into an existing Internet sharing testbed, iShare, and how they interact with iShare's resource publishing, discovery and allocation functionalities. The virtual machine implementation makes use of UML to execute jobs submitted by iShare users. The guest jobs have a (virtually) dedicated, native view of their own operating systems and networks, which are isolated from those of the underlying physical environment.

References

[1] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, et al. The nas parallel benchmarks-summary and preliminary results. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM Press.

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, et al. Operating system support for planetary-scale network services. In *Symposium on Networked Systems Design and Implementation*, March 2004.

[4] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.

[5] J. Dike. A user-mode port of the linux kernel. In *the 4th Annual Linux Showcase and Conference*, page 6372, October 2000.

[6] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, et al. Xen and the art of virtualization. In *the ACM Symposium on Operating Systems Principles*, 2003.

[7] R. Figueiredo, P. Dinda, and J. Fortes. A case for grid computing on virtual machines. In *International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11:115–128, 1997.

[9] W. Gropp, E. Lusk, N. Poss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[10] http://setiathome.ssl.berkeley.edu/. SETI@home: Search for extraterrestrial intelligence at home.

[11] http://www.netperf.org/. The public netperf homepage.

[12] http://www.vmware.com/. VMware - virtual infrastructure software.

[13] X. Jiang and D. Xu. VIOLIN: Virtual internetworking on overlay infrastructure. Technical Report CSD TR 03-027, Department of Computer Sciences, Purdue University, July 2003.

[14] N. H. Kapadia and J. A. B. Fortes. PUNCH: An architecture for web-enabled wide-area network-computing. *Cluster Computing*, 2:153–164, 1999.

[15] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *HPDC*, pages 47–54, 1999.

[16] I. Park, N. H. Kapadia, R. J. Figueiredo, R. Eigenmann, and et. al. Towards an integrated, web-executable parallel programming tool environment. In *Proc. Supercomputing Conference*, 2000.

[17] X. Ren and R. Eigenmann. iShare - open internet sharing built on peer-to-peer and web. In *European Grid Conference*, Amsterdam, The Netherlands, Feb. 2005.

[18] X. Ren, S. Lee, S. Bagchi, and R. Eigenmann. Resource fault prediction for fine-grained cycle sharing. Fast Abstracts, *International Conference on Dependable Systems and Networks (DSN)*, 2005.

[19] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Resource failure prediction for fine-grained cycle sharing. Technical report, ECE-HPCLab-05202, High-Performance Computing Lab, ECE, Purdue University, 2005.

[20] X. Ren, Z. Pan, R. Eigenmann, and Y. C. Hu. Decentralized and hierarchical discovery of software applications in the iShare internet sharing system. In *International Conference on Parallel and Distributed Computing Systems*, San Francisco, USA, Sep. 2004.

[21] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Towards virtual distributed environments in a shared infrastructure. *IEEE Computer, Special Issue on Virtualization Technologies*, May 2005.

[22] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2001.

[23] A. Sundararaj and P. Dinda. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research and Technology Symposium (VM 2004)*, May 2004.