A Fast RPC System for Virtual Machines

Hao Chen, *Member*, *IEEE*, Lin Shi, *Student Member*, *IEEE*,
Jianhua Sun, Kenli Li, and Ligang He, *Member*, *IEEE*

Abstract—Despite the advances in high performance interdomain communications for virtual machines (VM), data intensive applications developed for VMs based on the traditional remote procedure call (RPC) mechanism still suffer from performance degradation due to the inherent inefficiency of data serialization/deserilization operations. This paper presents VMRPC, a lightweight RPC framework specifically designed for VMs that leverages the heap and stack sharing mechanism to circumvent unnecessary data copy and serialization/deserilization. Our evaluation shows that the performance of VMRPC is an order of magnitude better than traditional RPC systems and existing alternative interdomain communication optimization systems. The evaluation on a VMRPC-enhanced networked file system across a varied range of benchmarks further reveals the competitiveness of VMRPC in IO-intensive applications.

Index Terms—Remote procedure call (RPC), virtual machine, shared memory, interdomain communication

1 Introduction

The virtual machine technology offers a number of benefits in the design and implementation of systems software. These include the ability of making more efficient use of hardware resources and minimizing the network overhead by colocating multiple modules acting on the same data on the same physical machine. Recently, a large class of communication-intensive distributed applications and software components have been ported to virtual machines, such as high-performance storage systems, network-router systems, and graphics rendering systems [17]. These applications demand a custom communication protocol. Although the researchers have developed high-performance solutions for these applications, there is still room to further improve the performance of virtual-machine-based applications as will be introduced in this paper.

In our previous work vCUDA [18], we also faced performance issues. The study involved building a virtual Compute Unified Device Architecture (CUDA) system in virtual machine monitors (VMM). The task of the virtual CUDA system is to intercept the normal API flow of the CUDA applications in VMs and redirect them to a privileged VM. Redirection was realized by using a traditional RPC system XMLRPC [23]. However, we found that XMLRPC caused severe performance degradation in VMMs, which motivated us to develop a high-throughput interdomain RPC system for data-intensive applications like vCUDA.

In this paper, we present the design and implementation of a new RPC system, Virtual Machine Remote Procedure Call (VMRPC). The main goal of VMRPC is to provide extremely low latency and high throughput between VMs in the same VMM. VMRPC combines the strengths of the local RPC optimization and interdomain communication-optimization techniques to avoid the performance issues that stem from the OS or VMM. Zero copy is also achieved in VMRPC, so that there is no user level or kernel level data copy as in normal RPC operations. Our evaluations show that the performance of VMRPC is 10 fold better than traditional RPC systems in VMMs. We implemented VMRPC in Xen [1], VMWare [21], and KVM [16]. The interface of VMRPC is small and clean, and there are only eight APIs exposed to the user, which makes VMRPC easy to learn and use. As the case studies, we integrated VMRPC into the vCUDA system and a networked file system, and extensively conducted the experimental measurements that validate our design choices and performance gains of VMRPC.

In this paper, we make the following contributions:

- We developed a low latency and high throughput inter-VM RPC tool, geared toward applications that require the dedicated high-performance RPC service in VMMs.
- We proposed a well-defined interface, making VMRPC easy to learn and portable across different VMMs.
- We implemented VMRPC on three representative virtual machine monitors, showing the portability and flexibility of VMRPC.
- We conducted extensive performance evaluation with microbenchmarks and on real systems, quantifying the merits of VMRPC.

Some preliminary results of this work were presented in our IWQoS'10 conference paper [8], however, additional technical details are added in the present paper. In addition, in this paper we demonstrate the potential of improving the performance of system-intensive workloads by using VMRPC to enhance a networked file system.

- *H. Chen, L. Shi, J. Sun, and K. Li are with the School of Information Science and Engineering, Hunan University, Chang Sha, Hunan 410082, P.R. China. E-mail: {haochen, shilin, jhsun}@aimlab.org, lkl510@263.net.*
- *L. He is with the Department of Computer Science, University of Warwick, Coventry CV47AL, United Kingdom. E-mail: liganghe@dcs.warwick.ac.uk.*

Manuscript received 16 Feb. 2012; revised 5 June 2012; accepted 17 June 2012; published online 22 June 2012.
Recommended for acceptance by D. Kaeli.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-02-0109.
Digital Object Identifier no. 10.1109/TPDS.2012.199.

2 Background

In this section, we motivate the design of VMPRC by enumerating several bottlenecks of traditional RPC. More background can be found in Section 1 of the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.199. Four major factors that affect the performance of traditional RPC systems in virtualized environments are as follows:

Problem 1. *high latency, by using socket-like communication APIs. In VMMs, a socket-like API has to pass through the TCP/IP protocol stack in both the hostOS and guestOS, which adds extra overhead to the communication path. Although the progress has been made in optimizing this kind of communication in VMMs, it is still less competitive than native asynchronous communication mechanisms.*

Problem 2. *low bandwidth data channel, layered on top of TCP/IP protocol stack. The TCP/IP protocol was originally developed for transferring data over an unreliable network. It performs poorly when being used between coresident VMs due to the virtualization overhead. For example, it has been reported that the page flipping mechanism in Xen would degrade the performance of network I/O [15], [24].*

Problem 3. *complex and expensive serialization/deserialization procedure. Serialization/deserialization is a standard operation in RPC systems. This operation is expensive because it involves a large amount of computation for looking up data tables, walking the data structure to pack them properly. In a typical RPC, the serialization/deserialization operations commonly occur four times, resulting in enormous computation overhead.*

Problem 4. *too many system calls involved in each RPC operation. Traditional RPC systems have two inherent problems. First, their performance is architecturally limited by the cost of invoking system calls, copying data between the user space and the kernel space, and possible thread rescheduling. Second, in VMs some system calls must be trapped and handled by the VMM, leading to significant context-switch overhead. In summary, the system calls in VMs are more expensive than in nonvirtualized environments.*

3 Design

In designing VMRPC, we used the following goals as guidelines:

- **Nonintrusiveness**: VMRPC should not add extra complexities to the system level components, and only depend on the primitives exported by VMMs.
- **High performance**: VMRPC should enable low latency, high throughput RPCs with low CPU consumption.
- **Portability**: VMRPC should provide support for different VMMs, and be easy to port across VMMs.
- **Simplicity**: VMRPC's interface should be small, clean, and easy to use.
- **Security**: VMRPC should not break the isolation principle already established in VMMs.

3.1 Nonintrusiveness

There are several different approaches to implementing a high performance RPC system in virtualized environments.

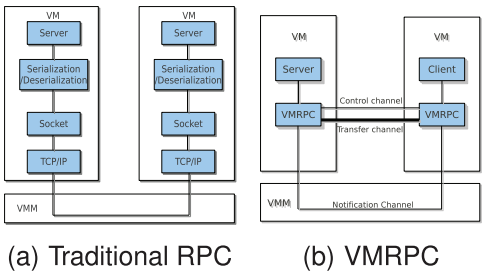


Fig. 1. Architecture comparison between VMRPC and traditional RPC.

A straightforward way is to modify the VMM to support a new data transfer mechanism. However, it is not preferable to add extra functionalities to the VMM, which may introduce security vulnerabilities and complicate the implementation of the VMM. Another solution is to develop a customized kernel module in the hostOS and/or guestOS[1] to establish a fast kernel-level communicating channel. However, that also means VMRPC would be tightly bound to specific kernel versions or operating systems. Finally, we decided to implement VMRPC using only the primitives exported to the user level by VMMs, without any modifications to the VMMs, and any modules/patches added to the hostOS/guestOS.

3.2 High Performance

The way to achieve high performance in VMRPC is mainly influenced by the issues exposed by traditional RPC systems as discussed in Section 2. *Problem 1* can be resolved by replacing the socket interface with a VMM platform-specific notification mechanism like the event channel in Xen. We solve *Problem 2* by utilizing the shared memory mechanism, as adopted by many existing inter-domain communication tools such as Xway or Xenloop. In order to overcome *Problem 3* and *Problem 4*, we realize memory sharing at the user level, where it is possible to eliminate data serialization/deserialization. Since the OS and VMM are bypassed in the main control flow of RPC, VMRPC can minimize the frequency of system calls. In general, VMRPC combines the strengths of the local RPC and inter-VM communication optimizations to achieve the performance goal.

3.3 Portability

Under the guidance of the portability principle, VMRPC consists of three subsystems: notification channel, control channel, and transfer channel, as shown in Fig. 1. The modular design of VMRPC makes it possible to separate most functionalities from the underlying VMM implementation, thereby facilitating the process of porting VMRPC to different VMMs.

3.4 Simplicity

Different from Xway or Xenloop, VMRPC is not binary compatible to legacy applications. A clean and well-defined interface is crucial to VMRPC. Traditional RPC frameworks are often invasive, requiring language tools and code generators to work. In contrast, VMRPC needs neither

1. In this paper, the term hostOS refers to a privileged OS (or Dom0 according to Xen's terminology). The term guestOS refers to a nonprivileged virtual machine (or DomU in Xen).

Interface Definition Language (IDLs) nor code generators, because the IDL is replaced by a standard C function calling convention, and the code generator is replaced by a convenient C preprocessor macro. By keeping the VMRPC interface clean and small, it is possible for developers to start writing high-quality code without having to go through a long learning process.

3.5 Security

VMRPC is specifically tailored toward the needs of high performance applications, and we make some reasonable assumptions as in Fido [7]. First, we assume that the software components in VMs are nonmalicious, and granting read-only access to shared memory is acceptable. Second, the possibility of the corruptions propagating from a faulty VM to a communicating VM via read-only access of memory is low. Despite these assumptions, we incorporate several strategies such as managed memory allocation, protection for sharing stack, and control flow verification, etc., into VMRPC to strengthen the system security to a significant extent, which is detailed in Section 4.3.

4 Implementation

To validate the design goals as discussed above, we have implemented VMRPC in three VMMs: Xen, VMWare Workstation, and KVM. Since we first implemented VMRPC in Xen, we use it as a representative VMM to describe the implementation details.

Fig. 1 depicts the architectural differences between the traditional RPC and VMRPC. VMRPC consists of three components: notification channel, control channel, and transfer channel. The transfer channel is a preallocated shared data zone dedicated to large capacity and high-speed data transfer. The control channel is also realized as a shared zone between two processes, while it is much smaller as compared to the transfer channel, and only used to store the control information of RPC like command index, function index, call flags, parameters, and stack content. The control channel can be regarded as the substitute of the External Data Representation (XDR) protocol in traditional RPC systems. The notification channel serves as an asynchronous notification mechanism, similar to hardware interrupts or software signals. Its main task is to trigger the RPC actions and synchronize concurrent accesses to the shared memory. The notification channel does not carry any actual payload, the RPC related information resides in the control channel and the transfer channel. In VMRPC, the notification channel is the only place where the OS and VMM must be involved.

From Fig. 1, it is obvious to see the advantages of VMRPC as compared with traditional RPC systems. First, moving the communication and control to the user level leaves the kernel (and VMM) only responsible for context switching. Second, VMRPC circumvents the TCP/IP protocol stack, and directly exploits the VMM platform-specific shared memory mechanism to represent and transfer data in the user space. Meanwhile, the expensive serialization/deserialization operations are also eliminated. Last, the VMM's built-in notification mechanism ensures the minimized latency for the RPC operations.

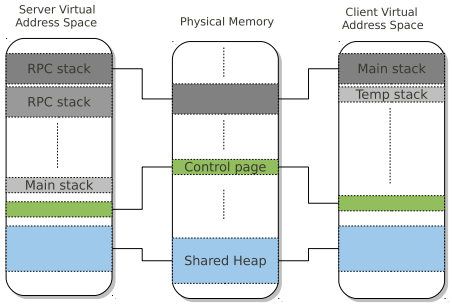


Fig. 2. Virtual address mapping in VMRPC.

In short, the efficiency of VMRPC comes from the "making the common case fast" approach to avoiding unnecessary synchronization, kernel-level thread management, and data copy between different address spaces on the same machine.

4.1 Memory Mapping

As shown in Fig. 2, VMRPC utilizes the user-level memory mapping to set up the control channel and the transfer channel. In Xen, this process is straightforward: the client first allocates a new virtual memory space, then the server maps the corresponding physical page frames into its own virtual address space by using the memory introspection API of Xenaccess: *user_va_map_range*. The case for VMWare is somewhat different. The server calls *VMCISharedMem_Create* to create a shared memory service, and then the client attaches to the service by calling *VMCISharedMem_Attach*. The following are some important issues related to memory mapping that arose in developing VMRPC.

Efficiency of mapping: When we map 100 M memory from a VM to Dom0 (host in VMWare), IVSHMEM [14] in KVM, and Xenaccess consume 5.5 millisecond and 1.5 seconds, respectively, while VMCI [20] in VMWare takes 23 seconds. During the execution of VMCI, we observed that the system's temporary folder (/tmp directory in Linux) generated a randomly named file whose size is exactly 100 M bytes. We speculate that the inefficiency of VMCI stems from the factor that it is not a shared memory mechanism that is directly built on top of page table mapping, based on the observation of the file system activities occurring in the mapping process. Further optimization to Xenaccess's mapping is possible, but it is beyond the scope of this paper.

Although the cost of memory mapping is relatively high (as compared with the runtime overhead) except with IVSHMEM, these operations are performed only once at the initialization stage. After the establishment of the memory mapping, all subsequent communications between two address spaces will be performed through logical channels that are pairwise shared between the client and the server.

Avoid demand paging: Most modern operating systems implement demand paging in virtual memory management. The OS allocates a physical page only if an attempt is made to access it. While this strategy works well in most cases, it is not desirable for our design of memory mapping. In VMRPC, when a mapping operation is performed on the server side, we must ensure that there are sufficient physical pages to be mapped. This limitation can be resolved by performing a write access to all the pages

belonging to that shared memory region, which guarantees there are enough physical memory frames to be mapped to each page in the shared virtual memory region.

Avoid page swapping: Another issue is that the page swapping strategy adopted by the operating systems may swap out the share pages to the disk, which will lead to inconsistent mapping between the server and the client. We prevent this situation from happening by using the page lock mechanism provided by the OS, such as *mlock* in Linux and *virtuallock* in Windows. These functions may be subject to the OS restrictions (such as the total number of pages that can be locked simultaneously), but so far, they have not caused any problems in our development environment.

Offset handling: None of Xenaccess, VMCI, and IVSHMEM provides support for mapping virtual memory at a designated address. In Xen, the client's shared memory is mapped to an arbitrary address in the server's address space. As a result, the pointer arguments (if any) of the function calls on the client side are incomprehensible to the server side functions. They need to be shifted to appropriate addresses on the server side in order for the RPC operations to execute properly. Since the length and content of shared memory is identical on both sides, all VMRPC needs to do is to add or subtract a constant offset to each pointer argument. VMRPC cannot do this automatically due to its inability of distinguishing pointers from other types of arguments, because there is no explicit type information available in VMRPC. We provide another API *vmrpc_offset* that must be called to compute the offset between two address spaces.

4.2 Transfer Channel

The transfer channel is built on top of *RPC heap*. The RPC heap is a preallocated memory region that is mapped into both the server and client address space, and large volume of data can be directly transferred through it. The RPC heap differs from the standard heap provided by operating system, but they can be used interchangeably by applications. VMRPC provides management APIs to specify the size of a RPC heap and create or destroy a RPC heap.

Zero copy: For interdomain RPC operations, exploiting shared memory is a straightforward way to avoid copying from the user space to the kernel space and vice versa. We ensure data accessibility by mapping the memory in the address space of the source process to the address space of the destination process, so that there are no user level copies. Kernel level copies are also avoided by removing the kernel and VMM from the critical path of data transfer.

RPC heap size: Since the mapped address of shared region in the server is completely random, it is difficult to change the size of a shared region dynamically once established. VMRPC relies on users to estimate the usage and size of a RPC heap. Oversized RPC heap is wasteful because physical memory pages are locked and spare pages are not allocatable to other applications until the end of a RPC operation. While undersized RPC heap may cause allocation failure in the shared region.

Heap management: We implemented a simple heap management interface. When a piece of data needs to be shared, the user should use *vmrpc_malloc* instead of the regular *C* function *malloc* to allocate memory blocks. When the memory is no longer in use, *vmrpc_free* should be called, which operates the same way as the standard *free* except it operates in VMRPC's heap. VMRPC also provides the APIs such as *vmrpc_heap_setup* and *vmrpc_heap_destroy* to create and destroy user-defined RPC heaps.

4.3 Control Channel

Since in VMMs the client and server reside in the same machine (although located in different VMs), it is unnecessary to pack and represent data in the complicated ways. In the following, we discuss the techniques related to the control channel.

Control page: The control page serves as a message exchange media. At startup, VMRPC stores the metadata, such as stack size, heap size, and starting addresses of the stack and the heap, in the control page. When the client issues a RPC operation, two types of control information are saved in the control page, call index and ESP value. The call index is used by the server to find the right service function in the RPC dispatch table. The ESP value indicates the stack frame of the current function. Since the client's main stack is mapped to the server's address space, the server also puts the return value in the control page.

RPC stack: In VMRPC, both the server and client have at least two stacks. The first one is their normal execution stack which we call "Server Main Stack" (SMS) and the second one is called "Client Main Stack" (CMS). When initializing a RPC operation, the client stores the CMS metadata in the control page, allowing the server to map the corresponding memory region to its own address space. Thus, the client's main stack becomes shared between the server and the client, which we call "RPC stack." A temporary stack is also set up in the client during the initialization stage. We describe the usage of this stack below.

For each RPC, the client first stores the call index on the top of the current stack in the control page, and switches to the temporary stack and notifies the server. In turn, the server switches from the current SMS to the RPC stack using the value kept in the control page. When the RPC finishes, the server switches back to the SMS and writes the return value to the control page. Then, the client replaces the return address on the RPC stack with the corresponding value on the temporary stack, takes the return value from the control page and makes this modified RPC stack as its execution stack. By now, a complete two-way RPC operation is accomplished.

Return address reservation: The control flow information in the client must be carefully reserved and restored because it may be modified during the process of stack sharing. For example, the return address in the RPC stack will be overwritten when the service function is performed on the server side. Thus, the client must be able to restore the original return address and change the corresponding value on the RPC stack, when the control flow is switched back.

Isolation issue: There is no doubt that sharing memory between VMs would compromise the isolation principle advocated by most virtualization solutions. We minimize this side effect in two ways. First, VMRPC achieves sharing based on the standard VMM interfaces, leaving the responsibility of guaranteeing isolation to VMMs. Second, VMRPC works in user level, which decreases the impact of fault propagation and system crash on system dependability.

Security issue: Being able to access the shared stack means that the server can alter the control flow of the client. Malicious intentions would lead to denial-of-service attack or exposure of sensitive information. In VMRPC, we assume the server is trusted, but the client is not. When the guestOS issues a RPC operation, the shared stack is configured as read-only to the guestOS until the server transfers the control back to the client. The server will validate the integrity of the return address on the stack and clear all sensitive information to ensure the security of stack sharing. Thus, the client cannot change the control flow or spy the data flow of the server.

4.4 Notification Channel

We implement the notification channel in VMRPC for two main reasons. First, in order to protect the shared stack and heap from concurrent access that may result in nondeterministic behaviors, we need to synchronize these concurrent accesses. Second, the RPC operation requires a way to allow both communicating parties to respond to a remote call or a return value. The VMM-specific asynchronous mechanisms, such as the event channel in Xen, the VMCI datagram in VMWare, and the interrupts in IVSHMEM are essential for VMRPC to build such a notification channel.

4.5 VMRPC User Interface

VMRPC provides a simple and clear interface to users, consisting of only eight APIs. The details on how to use these APIs can be found in Sections 2 and 3 of the supplementary file, available online, with an illustrative example.

5 Evaluation

In this section, we first evaluate the performance of VMRPC by comparing it with two traditional RPC systems (XMLRPC and ICE) and an interdomain communication system (Xenloop), and then conduct the experiments in a case study. Another case study can be found in the supplementary file, available online.

5.1 Test Setup

Unless otherwise stated, we conducted all the experiments on the following machine: Core Duo 6550 2.6 GHz CPU with 3 GB of memory running Xen 3.1, VMWare Workstation 6.0, and KVM 0.14.0 for linux. The hostOS and guestOS in Xen/VMWare and KVM are Fedora 8 (Linux kernel 2.6.21) and Fedora 13 (Linux kernel 2.6.33.3, IVSHMEM patch for KVM only works in newer kernels), respectively. In each test, the server ran in the Dom0 (or the host in VMWare and KVM) and the client ran in the DomU (or the guest in VMWare and KVM). When testing Xenloop, we used Linux kernel 2.6.18 and Xen 3.2 to meet the requirements of Xenloop. All the evaluation results are averaged across 10 runs.

5.2 Latency

We measured the cross-domain round-trip latency with a null RPC (defined as an empty function without any arguments and return value), which excludes the extra time spent on specific computations and data transfer. For completeness, we also conducted performance measurements about the native socket interface by transferring 1 byte data from the client to the server.

TABLE 1
The Results of Latency Test (in Microseconds)

μs	VMRPC	Socket	ICE	XMLRPC
Xen	15	40	66	320
VMWare	84	90	135	551
KVM	50	133	183	1033

As shown in Table 1, the numbers indicate the latency in microseconds averaged across 100,000 null RPC operations. We can observe that ICE is a highly efficient RPC system that incurs only a little overhead on top of the native socket. However, VMRPC is much faster than all other options due to its inherent mechanisms implemented. In contrast, the performance of VMRPC in VMWare is poorer than that in Xen and KVM, but still better than that in socket and traditional RPC tools. KVM incurs larger latency than Xen and VMWare in three other tests.

A white paper of ICE [13] states that the optimization of latency is constrained by the limitation of physical hardware like the network card. However, that limitation does not exist in VMRPC because there is no real network device in the guestOS. In addition, according to that white paper, the end-to-end performance is easily dominated by the actual processing cost, not the RPC protocol, and improving the latency further would not yield any noticeable improvement. This conclusion is not entirely correct for a RPC system that focuses only on the "local" calls. Low latency can improve the performance of RPC on the same hardware significantly, which is demonstrated in the case study on vCUDA (see Section 4 of the supplementary file, available online).

5.3 Throughput

Simply put, the throughput of any RPC system can be calculated as follows: throughput $= \frac{actual\ payload\ per\ RPC}{the\ execution\ time\ of\ a\ RPC}$. The actual payload is the data that a servant function processes, not including the RPC control information, the communication protocol messages, and the extra bytes resulting from serialization. We can easily obtain its value by defining a function with a fixed-length string as the argument. The denominator is more complex than the numerator. LPRC [2] analyzed seven aspects related to RPC's total execution time. Some are relatively stable, while others depend on dynamic characteristics of RPC operations. For example, serialization overhead is mainly affected by the size and complexity of actual payload, and the transfer overhead depends on the data volume after serialization.

VMRPC eliminates some kind of overhead from traditional RPC systems, but it also introduces some additional overhead: pointer argument conversion in *vmrpc_offset*, and heap management in *vmrpc_malloc* and *vmrpc_free*. To study the efficiency of VMRPC in the worst case, in the following tests, the overheads described above are all included in total cost of VMRPC.

Since the throughput of RPC is heavily dependent on the type of the data transferred, we define three types of data with different complexity: *byte sequence*, *fixed-length structure sequence*, and *variable-length structure sequence*. The definitions and actual payloads are summarized in Table 2.

TABLE 2
Definition of Three Data Types for Throughput Test

Type	Definition	Volume(Bytes)
Byte seq	`char ByteSeq[LEN*10];`	LEN*10
Fixed seq	`typedef struct { int i; int j; double d; } Fixed; Fixed FixedSeq[LEN];`	16*LEN
Variable seq	`typedef struct { char *s; double d; } Var; Var VarSeq[LEN];`	18*LEN

The LEN in Table 2 represents the length of sequences ranging from 100 to 10,000,000, so we can observe the throughput for a wide range of payloads (from 1 KB to about 100 MB). We analyze the receive and send operations separately. Fig. 3 shows the measured throughput as a function of the sequence length.

The gradual increase of throughput as the message size increases indicates that the performance is dominated by the per-message call overhead at small message sizes. As we expect, it is obvious that the throughput of VMPRC outperforms other RPC systems significantly. As shown in Figs. 3a and 3d, VMRPC achieves up to 10 times the throughput of ICE and XMLRPC in peak values (sequence length reaching 10,000). The relative discrepancy between VMRPC and ICE/XMLRPC is widening with the increasing complexity of messages, although the absolute value of the VMRPC throughput decreases (comparing (a) with (c), and (d) with (f)). VMRPC in KVM achieves the best performance approximating 2,500 MB/s (see the scale of y-axis) in three VMMs, even though KVM is a full-virtualization-based VMM. Reasoning for this large gap requires a comprehensive comparison of the inherent memory sharing mechanisms among three VMMs, which is beyond the scope of this paper.

With the increase of message size, the performance becomes dominated by the overhead of actual data transfer. ICE and XMLRPC decrease rapidly due to the serialization, copy, transfer, and context switch overhead. In VMRPC, the costs of memory allocation, pointer address conversion, and memory copy also lead to the decline of the throughput. A strange phenomenon is that VMRPC in VMWare sees a large drop when the workload exceeds 1,000,000, while it is stable in Xen and KVM with the same load. Further

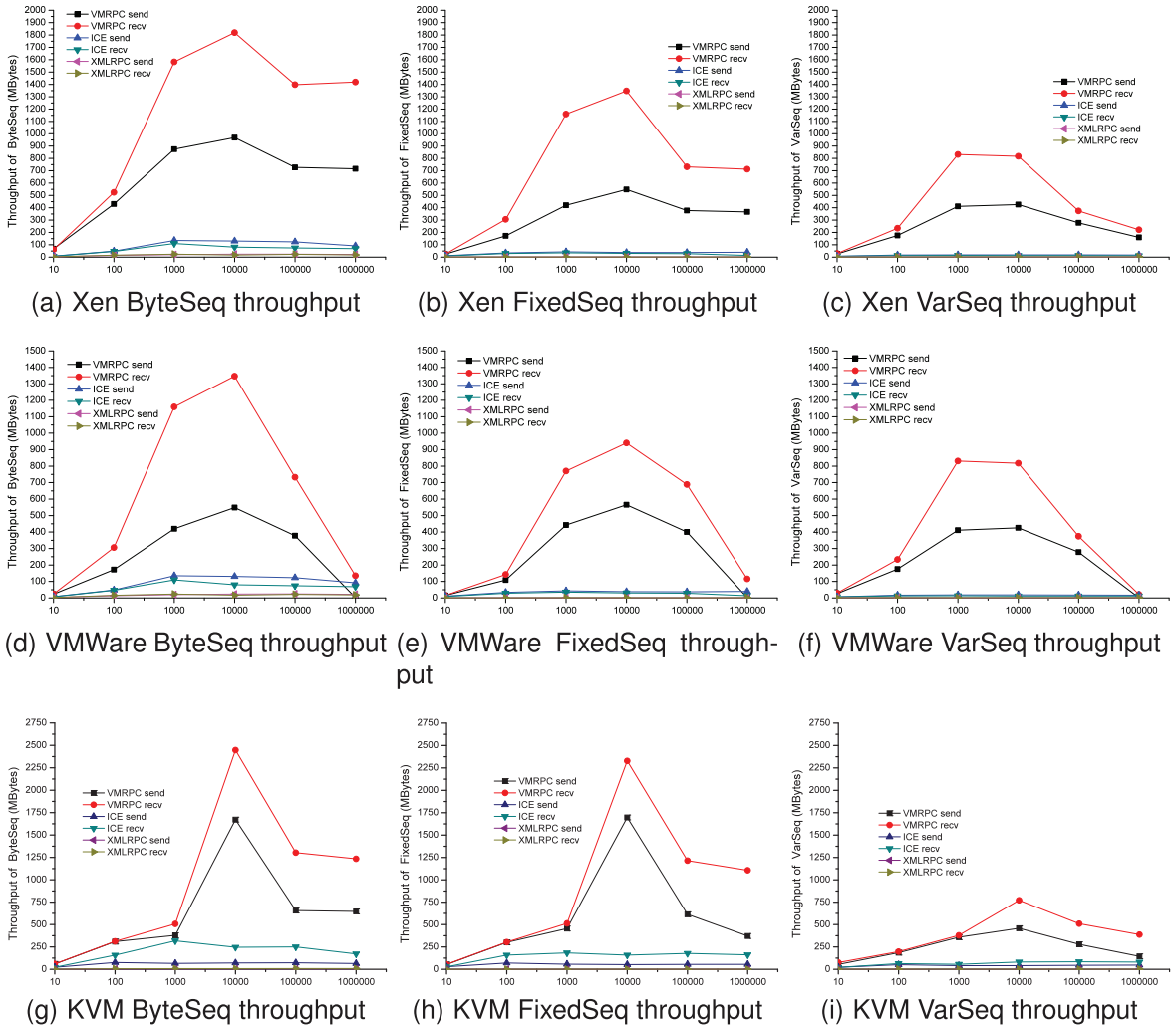


Fig. 3. The results of throughput test of VMRPC.

TABLE 3
Heap Setup Overhead (in Microseconds)

μs	1KB	10KB	100KB	1MB	10MB	100MB
Xenaccess	109	233	1554	34606	166631	1450916
VMCI	1089	1159	1612	11932	113412	22420105
IVSHMEM	47	47	50	109	620	5497

investigation reveals that when writing more than 50 MB data to the shared memory, the performance of VMRPC in VMWare is lower than that in Xen, resulting in higher overhead. We attribute this to the same reason that causes the deficiency in memory mapping.

It is easy to find that *VMRPC recv* performs better than *VMRPC send* because of a reduction in data copy. On the contrary, *ICE recv* is worse than *ICE send* due to an extra copy. However, it is not true for KVM because the send and receive bandwidth in KVM is asymmetric, which we confirmed by a simple bandwidth test. Thus, the results of *ICE recv* are better than *ICE send* in KVM even with the extra data copy. We also notice the curves of *ByteSeq* and *FixedSeq* of VMRPC show a similar trend, because the data of both types are intrinsically contained in a continuous region of virtual address space, resulting in less memory allocations and copies. The data of type *VarSeq* are usually composed by a large number of small and discrete memory blocks, which increase the frequency of allocation and copy. As a result, the turning point of curves of type *VarSeq* comes earlier.

5.4 RPC Heap Setup Overhead

Table 3 shows the time spent on mapping variably sized shared regions. Although the overhead for building a large heap is not ignorable in cases where the heap exceeds a certain size (with VMCI, we need approximately 22 seconds to set up a 100 MB heap), we regard this as the initialization overhead. It would not have any negative impact on the system performance at runtime. The measurement results of IVSHMEM are surprisingly good, and mapping 100 MB memory consumes only 5.4 milliseconds as compared to 1.5 seconds in Xenaccess and 22 seconds in VMCI.

5.5 VMRPC versus Xenloop

Having discussed the design, implementation, and evaluation of VMRPC, we may question that if the performance of VMPRC is superior to a traditional RPC system enhanced by an interdomain communication optimization system. In order to show the comparative advantage of VMRPC over interdomain communication optimization systems, we compare VMRPC with Xenloop [22] because of its desired features such as simplicity, transparency, and high performance. However, the current implementation of VMRPC does not support communication between DomUs in Xen, while Xenloop only offers optimization between two DomUs. Thus, we resort to analyze the relative speedup of throughput in their respective settings. The throughput of ICE between DomUs is presented in the second column of Table 4. We fill the third column with the improved throughput of ICE with Xenloop, and the speedup rate is shown in the fourth column. As a comparison, we run ICE and VMRPC tests between Dom0 and DomU, the seventh column reflects the acceleration ratio of VMRPC versus ICE. From the results, we can see the speedup from VMRPC is much better than that of Xenloop. In addition, the advantage of VMRPC becomes clearer as the complexity of payload grows. VMRPC performs about 122 times better than ICE in *Var recv* as shown in the last row of Table 4.

TABLE 4
Throughput Comparison between Xenloop and VMRPC

Mbps	ICE DomU	ICE DomU Xenloop	Xenloop Speedup	ICE Dom0	VMRPC Dom0	VMRPC Speedup
Byte send	1541	1923	124%	2474	7792	315%
Byte recv	818	846	103%	728	15216	2090%
Fix send	391	401	102%	404	7456	1845%
Fix recv	291	292	100%	270	14912	5523%
Var send	67	66	98%	67	4800	7164%
Var recv	57	57	100%	57	7008	12294%

5.6 Case Study: A VMRPC-Based Networked File System

The performance evaluation in previous sections focuses on synthetic data-intensive benchmarks (another case study on the data-intensive application vCUDA [18] can be found in Section 4 of the supplementary file, available online), which involves little operating system activity, and the results present the best case acceleration potential of performance. In this section, we explore the possibility of speeding up the performance of a networked file system in virtual machines with VMRPC, which further quantifies the advantage of VMRPC in system-intensive workloads.

Our evaluation is based on *stfufs* [19], a FUSE-based [10] networked file system. In virtualized environments, the client and server of a networked file system share the same hardware, but they are partitioned into different virtual machines by the VMM. The networked file systems typically use RPC or customized RPC-like mechanisms to exchange data between the client and the server. However, RPC incurs nontrivial overhead because of data copying, network stack traversing, and data marshalling as discussed in previous sections. To illustrate the superiority of VMRPC, we enhanced *stfufs* by replacing its original communication protocol with VMRPC, and conducted detailed evaluation on a set of synthetic workloads. In the following, we first present how VMRPC's operations are mapped into *sftufs* with a motivating example to demonstrate the processing of typical system calls. Then, we show the evaluation results of *sftufs* with various file system workloads. A brief introduction of FUSE and *stfufs* can be found in Section 1.4 of the supplementary file, available online.

To better understand how VMRPC works in *sftufs*, we detail the workflow of a representative callback function *stfufs_read*. When triggered by the *read* system call in FUSE, *stfufs_read* first sends control information to the server through the control page, and the function parameters are sent to the server via the shared stack. The transfer channel is not established yet because the read call does not involve actual data transfer at this time, but pointers (if any) pointing to shared heap have reached the server as part of the control page. Then, *stfufs_read* starts the notification channel and suspends. Upon the arrival of the client

TABLE 5
Filebench Workload Characteristics

Workload	Average file size	Number of files	I/O sizes			R/W ratio
			read	write	append	
Mail Server	16KB	1000	1MB	-	16KB	1:1
Web Server	16KB	1000	1MB	-	16KB	10:1
File Server	128KB	10000	1MB	1MB	16KB	1:2

request, the server switches from the main stack to the RPC stack, parses the client's request and issues the local system call. Then, the server puts the data read locally into the transfer channel (shared heap), switches from the RPC stack back to the main stack. At last, a notification is sent to the client asynchronously. When the client is woken up by the notification channel, *stfufs_read* is invoked to copy the data from the shared heap to the destination specified by the second parameter of *stfufs_read* (*static int stfufs_read(const char *path, char *data, size_t size, ...)*). This copy is unavoidable because the memory allocated for "*char *data*" is from the FUSE system, which prevents us from using the shared heap to manage it.

5.6.1 Evaluation of stfufs

We ran all experiments on the same machine as shown in Section 5. We deployed the *sftufs* server and client in the hostOS and guestOS, respectively. The guestOS is configured with 1.5 GB RAM, single VCPU, and bridged network in all VMMs (Xen, VMWare, and KVM). We used *FileBench* [9], an application level workload generator to emulate a wide range of file system workloads. We chose three common server workloads: mail server, web server, and file server. Table 5 summarizes the workload characteristics. Three basic performance metrics: throughput, latency, and CPU time per system call are reported.

Mail server. In mail server workload, FileBench performs a sequence of operations to imitate reading mails (*open*, *read* whole file, and *close*), composing (*open/create*, *append*, *close*, and *fsync*) and deleting mails. The average file size is 16 KB and the read-write ratio is 1:1.

Web server. The web server workload uses a read-write ratio of 10:1, and reads entire files sequentially by multiple threads, as if reading webpages. All the threads append 16 KB to a common web log.

File server. The file server workload emulates a server hosting home directories of multiple users. Each thread performs a series of *create*, *delete*, *append*, *read*, *write*, and *stat* operations, exercising both the metadata and data paths of the file system. The average file size is 128 KB and the read-write ratio is 1:2.

As Table 6 shows, stfufs-VMRPC significantly outperforms stfufs-orig in all cases. In general, VMRPC improved the throughput by 28.33 percent up to 72.94 percent, the latency by 22.4 percent up to 55.68 percent, the CPU cycles by 3.16 percent up to 16.59 percent. Memory sharing and event-based notification facilities helped increase throughput and decrease latency, respectively. The performance gain in CPU time is relatively small as compared to throughput and latency, because *stfufs* uses a socket-based customized communication protocol instead of a general RPC protocol. Thus, *stfufs* does not involve RPC specific overhead. The improvement on CPU time mainly relies on the elimination of buffer management and socket-based communication in *stfufs*.

When writing data larger than 50 MB in VMWare, we did not observe any abnormalities occurring in experiments discussed in previous sections. The reason is that the read and write buffer size in FUSE are 128 and 4 KB, respectively, which are far less than the 50 MB bound. Although the solution for enlarging these buffers exists, we did not go further on large buffer measurements (requiring patches to FUSE and kernel recompilation). Indeed, larger buffer would have a positive effect on throughput evaluation.

Considering the throughput and latency in the mail and web server workloads, in Table 6, we can see that the speedup ratios in Xen, VMWare, and KVM are shown in descending order (observing the columns of throughput and latency vertically). For example, the speedup ratio of throughput for mail server workload is 72.94 percent in Xen, 50 percent in VMWare, and 40.91 percent in KVM, and stfufs-VMRPC decreases the latency for web server workload by 37.50, 25.37, and 22.40 percent in Xen, VMWare, and KVM, respectively. The file server workload does not exhibit such a pattern, where stfufs-VMRPC achieves the best speedup ratio in VMWare.

Next, we examine how the speedup ratios vary among three different workloads in individual VMMs. Observing the horizontal variations in the speedup row of Table 6, we

TABLE 6
Performance Evaluation Results of *stfufs* File System under Different Workloads

		Mail Server			Web Server			File Server		
		Throughput (MB/s)	Latency (ms)	CPU (us/op)	Throughput (MB/s)	Latency (ms)	CPU (us/op)	Throughput (MB/s)	Latency (ms)	CPU (us/op)
KVM	stfufs-orig	2.2	79.8	1488	6.0	36.6	1322	5.0	261.6	4063
	stfufs-VMRPC	3.1	54.3	1441	7.7	28.4	1174	6.9	194.2	3814
	speedup	40.91%	31.95%	3.16%	28.33%	22.40%	11.20%	38.00%	25.76%	6.13%
VMWare	stfufs-orig	4.2	43.0	1160	10.6	20.5	943	6.1	214.2	3978
	stfufs-VMRPC	6.3	25.5	1092	14.0	15.3	805	9.1	139.4	3318
	speedup	50.00%	40.70%	5.86%	32.08%	25.37%	14.63%	49.18%	34.92%	16.59%
Xen	stfufs-orig	8.5	17.6	474	25.2	4.8	445	12.3	106.5	899
	stfufs-VMRPC	14.7	7.8	448	33.4	3.0	387	17.7	72.9	822
	speedup	72.94%	55.68%	5.49%	32.54%	37.50%	13.03%	43.90%	31.55%	8.57%

know that the throughput and latency speedup ratios both follow the relation: mail server > file server > web server in all VMMs. For example, referring the throughput column in KVM, we can observe that the ratio of mail Server, file server, and web server is 40.91, 38, and 28.33 percent, respectively. Intuitively, workload involving large data transfer such as the file server workload would get more acceleration, but Table 6 gives opposite results. We attribute this to the complexity nature of the system-intensive workload. Note that *stfufs* achieves the best baseline performance in Xen due to its para-virtualization architecture.

6 Related Work

We present the related work by organizing the literature into local RPC and interdomain communication optimizations.

LRPC [2] addressed how local RPC can be implemented with minimal overhead. It emulates the native procedure call model, and no extra message passing but the original procedure-call convention is needed. By running the client's thread to perform requested services in the address space of the server, LRPC sets up a simple control transfer model. Bourassa and Zahorjan [6] extended LRPC to the Mach3 operation system, and also changed the language call convention from Modula2+ to C. URPC [3] is very similar to VMRPC in some aspects such as OS-bypass, it optimizes the RPC by moving the communication facilities from kernel to user space. Nevertheless, URPC is still an intra-OS RPC, more precisely, an IPC tool. SHRIMP RPC [4], [5] actually is the adoption of URPC in a distributed memory architecture. FastRPC [11] is designed for splitting monolithic programs into multiple cooperating processes that can be confined by a kernel security framework.

XenSocket [24] is a one-way communication channel between two VMs based on the shared memory. It defines a new socket type, with associated connection establishment and read-write system calls that provide interfaces to the developers by utilizing the underlying inter-VM shared memory mechanism. IVC [12] is a user level communication library intended for message-passing HPC applications. It provides socket style APIs. Both Xway [15] and Xenloop [22] offer a fully transparent interdomain communication channel. The difference between them is that Xway intercepts the TCP/IP stack beneath the socket layer, and Xenloop exploits the netfilter hooks in Linux to intercept the outgoing network packets routed to another VM. Xway needs extensive modifications to the network protocol stack in the operating system. Xenloop is implemented as a kernel module, and one major drawback of Xenloop is that it does not support the communication between the hostOS and the guestOS. Fido [7] is a high-performance interdomain communication mechanism tailored for enterprise appliances. Although VMRPC shares some similarities with Fido based on the techniques implemented in these two systems, such as the shared global address space in Fido and the shared heap and stack in VMRPC, but Fido is not specifically designed as a RPC system. Thus, we believe VMRPC is complementary to Fido as VMRPC can eliminate much of the overhead inherent in RPC systems.

7 Conclusions

In this paper, we have presented the design, implementation, and performance evaluation of VMRPC. VMRPC trades transparency for efficient communications, and provides fast responsiveness and high throughput when deployed for communicating components in virtualized environments. It is specifically designed for the applications that require high-volume data transfer between VMs. Our evaluation shows that the performance of VMRPC is an order of magnitude better than the traditional RPC systems and the existing interdomain communication mechanisms in processing data-intensive applications, and VMRPC also improves the performance of a networked file system by up to 73 percent.

Acknowledgments

The authors are grateful to the anonymous reviewers for their helpful feedback. This research was supported in part by the Program for New Century Excellent Talents in University, the National Natural Science Foundation of China under grants 61272190, 61173166, and 60803130, the National Basic Research Program of China under grant 2007CB310900, the Key Program of National Natural Science Foundation of China under grant 6133005, the Leverhulme Trust of United Kingdom under grant RPG-101, and the Fundamental Research Funds for the Central Universities of China. The corresponding author is Lin Shi.

References

[1] P. Barham, B. Dragovic, K. Fraser, and S. Hand, "Xen and the Art of Virtualization," *Proc. ACM Symp. Operating Systems Principles (SOSP)*, Oct. 2003.

[2] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight Remote Procedure Call," *ACM Trans. Computer Systems*, vol. 8, no. 1, pp. 37-55, Feb. 1990.

[3] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "User-Level Interprocess Communication for Shared Memory Multiprocessors," *ACM Trans. Computer Systems*, vol. 9, no. 2, pp. 175-198, May 1991.

[4] A. Bilas and E. Felten, "Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface," *J. Parallel and Distributed Computing*, vol. 40, no. 1, pp. 138-146, 1997.

[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," *Proc. 21st Ann. Int'l Symp. Computer Architecture (ISCA)*, pp. 142-153, Apr. 1994.

[6] V. Bourassa and J. Zahorjan, "Implementing Lightweight Remote Procedure Calls in the Mach 3 Operation System," Technical Report TR-95-02-01, Dept. of Computer Science and Eng., Univ. of Washington, Feb. 1995.

[7] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L.N. Bairavasundaram, K. Voruganti, and G.R. Goodson, "Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances," *Proc. Conf. USENIX Ann. Technical Conf.*, June 2009.

[8] H. Chen, L. Shi, and J.H. Sun, "VMRPC: A High Efficiency and Light Weight RPC System for Virtual Machines," *Proc. 18th Int'l Workshop Quality of Service (IWQoS)*, June 2010.

[9] FileBench, http://www.fsl.cs.sunysb.edu/vass/filebench/, 2012.

[10] Filesystem in Userspace, http://http://fuse.sourceforge.net/, 2012.

[11] M. Hearn, "Security-Oriented Fast Local RPC," technical report, Dept. of Computer Science, Univ. of Durham, 2006.

[12] W. Huang, M. Koop, Q. Gao, and D.K. Panda, "Virtual Machine Aware Communication Libraries for High Performance Computing," *Proc. ACM/IEEE Conf. Supercomputing*, Nov. 2007.

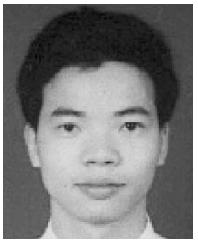
[13] ICE Performance White Paper, http://www.zeroc.com/articles/IcePerformanceWhitePaper.pdf, 2012.

[14] Inter-VM Shared Memory PCI Device, http://lwn.net/Articles/380869/, 2012.
[15] K. Kim, C. Kim, S.I. Jung, H. Shin, and J.S. Kim, "Inter-Domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen," *Proc. ACM Fourth SIGPLAN/SIGOPS Int'l Conf. Virtual Execution Environments (VEE),* 2008.
[16] KVM, http://www.linux-kvm.org, 2012.
[17] H.A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de La-ra, "VMM-Independent Graphics Acceleration," *Proc. Third Int'l Conf. Virtual Execution Environments (VEE '07),* June 2007.
[18] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU Accelerated High Performance Computing in Virtual Machines," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS),* May 2009.
[19] stfufs, http://www.guru-group.fi/too/sw/stfufs/, 2012.
[20] VMCI, http://pubs.vmware.com/vmci-sdk/index.html, 2012.
[21] VMWare, http://www.vmware.com, 2012.
[22] J. Wang, K. Wright, and K. Gopalan, "XenLoop: A Transparent High Performance Inter-VM Network Loopback," *Proc. 17th Int'l Symp. High Performance Distributed Computing,* June 2008.
[23] XMLRPC, http://www.xmlrpc.com, 2012.
[24] X. Zhang, S. McIntosh, P. Rohatgi, and J.L. Griffin, "XenSocket: A High-Throughput Interdomain Transport for Virtual Machines," *Middleware: Proc. ACM/IFIP/USENIX Int'l Conf. Middleware,* 2007.

Hao Chen received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2005. He is now an associate professor at the School of Information Science and Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing and systems security. He published more than 40 papers in top journals. He is a member of the IEEE and the ACM.

Lin Shi is working toward the PhD degree at the School of Information Science and Engineering, Hunan University, China. His research interests include virtual machines and GPGPU computing. He is a student member of the IEEE.

Jianhua Sun received the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2005. She is an associate professor at the School of Information Science and Engineering, Hunan University, China. Her research interests include security and operating systems.



Kenli Li received the BS degree in mathematics from Central South University, China, in 2000, and the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2003. He has been a visiting scholar at University of Illinois at Champaign and Urbana from 2004 to 2005. He is now a professor of the School of Information Science and Engineering at Hunan University. His major research contains parallel computing, grid and cloud computing, and DNA computer. He is a senior member of the CCF.

Ligang He received the PhD degree in computer science from the University of Warwick, United Kingdom, from 2002 to 2005, and then worked as a postdoctor in the University of Cambridge, United Kingdom. In 2006, he joined the Department of Computer Science at the University of Warwick as an Assistant Professor. He is now an associate professor in the Department of Computer Science at the University of Warwick. His research interests include parallel and distributed processing, cluster, grid and cloud computing. He has published more than 40 papers in international conferences and journals, such as *IEEE Transactions on Parallel and Distributed Systems,* IPDPS, CCGrid, MASCOTS. He has been a member of the program committee for many international conferences, and been the reviewer for a number of international journals, including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, etc. He is a member of the IEEE.