RPC System for Distributed Network Systems

Vijay Masne
Assistant Professor
Department of Computer Science and Engineering
Dr. Babasaheb Ambedkar College of Engineering and Research

Rashmi Phasate
Assistant Professor
Department of Electrical Engineering
G. H. Raisoni Institute of Engineering & Technology for Women

Supriya Thombre
Assistant Professor
Department of Computer Technology
Yeshwantrao Chavan College of Engineering

Nilesh Sambhe
Assistant Professor
Department of Computer Technology
Yeshwantrao Chavan College of Engineering

Abstract

Remote procedure call (RPC) systems have been proven to be a practical basis for building distributed applications. The RPC technique abstracts a typical communication pattern to an ordinary procedure call. Compared with an ordinary procedure call, however, the conventional RPC technique has one evident restriction; pointers (addresses) cannot be passed to remote procedures without the explicit and nontrivial programming effort. This paper presents VMRPC, a light-weight RPC framework specifically designed for VMs that leverages the heap and stack sharing mechanism to circumvent unnecessary data copy and serialization/deserialization. Our evaluation shows that the performance of VMRPC is an order of magnitude better than traditional RPC systems and existing alternative inter-domain communication optimization systems.
Keywords: Remote Procedure Call (RPC), Virtual Machine, Shared Memory, Commutation

I. INTRODUCTION

The virtual machine technology offers a number of benefits in the design and implementation of systems soft- ware. These include the ability of making more efficient use of hardware resources and minimizing the network overhead by co-locating 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 CUDA (Compute Unified Device Architecture) 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 inter-domain RPC system for data- intensive applications like vCUDA.

In this paper, we present the design and implementation of a new RPC system, VMRPC (Virtual Machine Remote Procedure Call). The main goal of VMRPC is to provide extremely low latency and high through- put between VMs in the same VMM. VMRPC com- bines the strengths of the local RPC optimization and inter-domain 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 ten-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 towards applications that require the dedicated high-performance RPC service in VMMs.
- We proposed a well-defined interface, making VM- RPC 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 micro benchmarks 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.

II. 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. Four major factors that affect the performance of traditional RPC systems in virtualized environments are as follows.

A. 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 host OS and guest OS, 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.

B. 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 co-resident 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].

C. 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.

D. 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 non- virtualized environments.

III. DESIGN

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

- Non-intrusiveness: 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 la- tency, high throughput RPCs with low CPU con- sumption.
- 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.

A. Non-Intrusiveness:

There are several different approaches to implementing a high performance RPC system in virtualized environments.
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 host OS and/or guest OS 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 host OS/guest OS.

B. 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 exist- ing 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.

C. Portability:

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

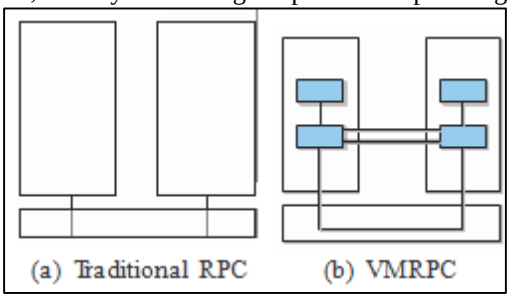

Fig. 1: comparison between VMRPC and Traditional RPC

D. 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 IDLs (Interface Definition Language) 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.

E. Security:

VMRPC is specifically tailored towards the needs of high performance applications, and we make some reasonable assumptions as in Fido [7]. First, we assume that the soft- ware components in VMs are non-malicious, and grant- ing 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.

IV. IMPLEMENTATION

To validate the design goals as discussed above, we have implemented VMRPC in three VMMs: we use it as a representative VMM to describe the implementation details.

Figure 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 pre- allocated 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 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 Figure 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. Mean- while, the expensive serialization/deserialization operations are also eliminated. Last, the VMM's built-in notification mechanism ensures the minimized latency for the RPC operations.

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.

A. Memory Mapping:

As shown in Figure 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 100M 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 100M 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 pair-wise shared between the client and the server.

1) 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.

2) Avoid Page Swapping:

Another issue is that the page swapping strategy adopted by the operating systems.

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.

3) 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.

4) Offset Handling:

None of Xenaccess, VMCI and IVSHMEM provides support for mapping virtual mem- ory 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 distinguish- ing 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.

B. Transfer Channel:

The transfer channel is built on top of RPC heap. The RPC heap is a pre-allocated memory region that is mapped into both the server and client address space, and large volume of data can be directly transferred through it.

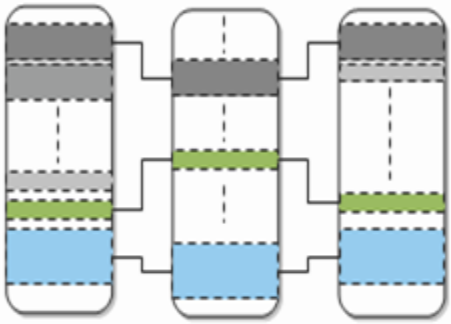

Fig. 2: Virtual address mapping in VMRPC

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.

1) Zero Copy:

For inter-domain 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.

2) 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 allocate to other applications until the end of a RPC operation. While undersized RPC heap may cause allocation failure in the shared region.

3) 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 vm- rpc heap destroy to create and destroy user-defined RPC heaps.

C. Control Channel:

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

1) Control page:

The control page serves as a message exchange media. At startup, VMRPC stores the meta- data, 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.

2) 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.

'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.

3) 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.

4) 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.

5) 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 guest OS issues a RPC operation, the shared stack is configured as read-only to the guest OS.Until the server transfers the control back to the client. The server will validate the integrity of the return ad- dress 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.

D. 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 non-deterministic 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.

E. 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 Section 2 and Section 3 of the supplementary file with an illustrative example.

V. 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), 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 sftufs [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 non-trivial 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. 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 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.

A. 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 1.5GB RAM, single VCPU and bridged network in all VMMs (Xen, VMWare and KVM). We used File Bench [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, File Bench per-forms 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 16KB 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 Web pages. All the threads append 16KB 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 128KB and the read-write ratio is 1:2.

VI. RELATED WORK

We present the related work by organizing the literature into local RPC and inter-domain 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. [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. Fast RPC [11] is designed for splitting monolithic programs into multiple co-operating 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 an 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 inter-domain communication channel. The difference be- tween them is that Xway intercepts the TCP/IP stack be- neath the socket layer, and Xenloop exploits the netfilter hooks in Linux to intercept the outgoing network packets routed to another VM. Xway needs extensive modifi- cations 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 inter-domain communication mechanism tailored for enterprise appli- ances. 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.

VII. 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 VM- RPC is an order of magnitude better than the traditional RPC systems and the existing inter-domain communication mechanisms in processing data-intensive applications, and VMRPC also improves the performance of a networked file system.

REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, and S. Hand. ”Xen and the art of virtualization”. In Proc. ACM Symposium on Operating Systems Principles (SOSP), Oct. 2003.
[2] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. ”Lightweight remote procedure call”. ACM Transactions on Computer Systems (TOCS), v.8 n.1, p.37-55, Feb. 1990.
[3] B. Bershad, T. Anderson, E Lazowska, and H. Levy. ”User-level interprocess communication for shared memory multiprocessors”. ACM Transactions on Computer Systems (TOCS), v.9 n.2, p.175-198, May 1991.
[4] A. Bilas and E. Felten. ”Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface”. Journal of Parallel and Distributed Computing. 40(1) pp.138-146, 1997.
[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sand- berg. ”Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer”. In Proc. 21st Annual International Symposium on Computer Architecture (ISCA), Chicago, pp. 142-153, April 1994.
[6] V. Bourassa and J. Zahorjan. ”Implementing lightweight remote procedure calls in the Mach 3 operation system”. Technical Report TR-95-02-01, University of Washington, Department of Computer Science and Engineering, Feb 1995.
[7] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasun- daram, K. Voruganti, and G. R. Goodson. ”Fido: Fast Inter-Virtual- Machine Communication for Enterprise Appliances”. In Proc. USENIX, June 2009.
[8] H. Chen, L. Shi, and J. H. Sun. ”VMRPC: A High Efficiency and Light Weight RPC System for Virtual Machines”. In Proc. IWQoS, June 2010.
[9] FileBench. http://www.fsl.cs.sunysb.edu/∼vass/filebench/.
[10] Filesystem in Userspace. http://http://fuse.sourceforge.net/.
[11] M. Hearn. ”Security-oriented fast local RPC”. Technical Report in the Department of Computer Science, University of Durham. 2006.
[12] W. Huang, M. Koop, Q. Gao, and D. K. Panda. ”Virtual machine aware communication libraries for high performance computing”.In Proc. SuperComputing, Reno, NV, Nov. 2007.
[13] ICE performance white pater. http://www.zeroc.com/articles/ IcePerformanceWhitePaper.pdf.
[14] Inter-VM shared memory PCI device. http://lwn.net/Articles/380869/.
[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”. In Proc. VEE, ACM Press, 2008.
[16] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de La- ra. ”VMM-independent Graphics Acceleration”. In Proc. VEE 2007. ACM Press, June 2007.