VMRPC: A High Efficiency and Light Weight RPC System for Virtual Machines

Hao Chen, Lin Shi and Jianhua Sun
Advanced Internet and Media Lab
School of Computer and Communication
Hunan University, Chang Sha, 410082, China
{haochen,shilin,jhsun}@aimlab.org

***Abstract*—Despite advances in high performance inter-domain communication for virtual machines (VM), data intensive applications developed for VMs based on traditional remote procedure call (RPC) mechanism still suffer from performance degradation due to the inherent inefficiency of data serialization/deserilization operation. This paper presents VMRPC, a light-weight RPC framework specifically designed for VMs that leverages heap and stack sharing to circumvent unnecessary data copying and serialization/deserilization, and achieve high performance. Our evaluation shows that the performance of VMRPC is an order of magnitude better than traditional RPC systems and existing alternative inter-domain communication mechanisms. We adopt VMRPC in a real system, and the experiment results exhibit that the performance of VMRPC is even competitive to native environment.**

I. Introduction

Virtual machine technologies offer a number of benefits in the design and implementation of middleware. These include the ability to make more efficient use of hardware resources and to minimize network overhead by co-locating multiple modules acting on the same data on the same physical machine. Recently, a large category of communication intensive distributed applications and software components have been ported to virtual machine platform, such as high performance storage systems [17], network-router systems [9], and graphics rendering systems [15]. These applications desire for a dedicated communication protocol. Although researchers have developed high performance solutions for these applications, a general purpose RPC system for virtual machines has never been put on the table.

In our former study vCUDA [20], we faced the same problem, the study involved building a virtual CUDA (Compute Unified Device Architecture) system in VMM platform. The task of the virtual CUDA system was to interpret the normal API flow of a CUDA application in a VM and to redirect them to another privileged VM. Redirection was realized by using a traditional RPC system XMLRPC [25] . However, we found XMLRPC caused serious performance degradation in VMM platform, which motivated us to develop a high-throughput inter-domain RPC system for data-intensive applications like vCUDA described above.

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 throughput between VMs in the same VMM. VMRPC combines 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 in a regular RPC operation. Our evaluations show VMRPC's performance is ten folds better than traditional RPC systems in VMMs. For now, we have implemented VMRPC in Xen [1] and the VMWare [22]. VMRPC's interface is small and clean, and there are only 8 APIs exposed to the programmer, which makes VMRPC easy to learn and use. As a case study, we apply VMRPC to the vCUDA project, and experimental results reveal that VMRPC significantly reduces the virtualization overhead.

In this paper, we make the following contributions:

- A low latency and high throughput inter-VM RPC solution, geared towards enterprise appliances that require dedicated high performance RPC in VMMs.
- Well-defined interfaces and abstraction layers, making VMRPC portable across different VMMs.
- Extensive performance evaluations and a real system case study, quantifying the merits of VMRPC.

The rest of this paper is organized as follows. Section II covers background for RPC research and the relevant facility in Xen and VMWare. Section III and section IV discusses the design and implementation of VMRPC respectively. VMRPC interface is introduced in Section V. Next we evaluate VMRPC in Section VI along with a case study. In Section VII, we discuss related work. Finally, in Section VIII we present our conclusions.

II. Background

In the rest of this paper, the term hostOS refers to the administrative OS (or domain0 according to Xen's semantics). The term guestOS refers to a VM (or domainU in Xen).

A. Bottlenecks of Traditional RPC

From our observation, factors that affect the efficiency of traditional RPC systems in VMM environment are as follows:

Problem 1: high latency, by using socket-like communication API.

In VMM, socket-like API has to pass through TCP/IP protocol stacks both in the hostOS and guestOS, which adds extra overhead to the communication path. Although progress has been made to optimize this kind of communication in VMM, it is still less competitive than native asynchronous communication mechanism.

Problem 2: low bandwidth data channel, layered on top of TCP/IP protocol stack.

TCP/IP protocol was originally developed to transfer data over an unreliable network. It performs poorly when used between co-resident VMs due to VMM's virtualization overhead. For example, it was reported that the *page flipping* mechanism in Xen would degrade the performance of network I/O [14], [27].

Problem 3: complex and expensive serialization/deserialization procedure.

Serialization/deserialization is a standard operation of RPC systems. It is expensive because it involves running a large amount of code for looking up data tables, walking the data structure to pack them properly. In a typical RPC, serialization/deserialization procedures commonly occur four times, resulting in enormous computation overhead.

Problem 4: too many system calls involved in each RPC.

Traditional RPC systems have two inherent problems. First, its performance is architecturally limited by the cost of invoking the kernel system calls, data copying between user space and kernel space, and the possible thread rescheduling; Second, in VMs some system calls have to be trapped and handled by the VMM, leading to significant overhead in context switch. In summary system calls in virtual machines are more expensive than in non-virtualization environment.

B. Local RPC Optimization

Local RPC optimization techniques dedicated to developing a high efficiency RPC system work in a non-virtualized environment, such as LRPC [2], URPC [3], SHRIMP RPC [5], ARPC [26]. LRPC reduced the cost of same-machine communication to nearly the lower bound imposed by hardware. Such impressively high performance derives from two aspects: shared memory and scheduling. Operating systems are able to use the global address space or memory mapping infrastructure, and take advantage of the flexibility of scheduling to speed up RPC operations. Shared memory is helpful for reducing the time of data copy, in some cases like LRPC, it also avoids the serialization/deserialization operation both in the client and server. The optimization of scheduling allows for fast responses, and it also eliminates some context switch overhead.

C. Inter-domain Communication Optimization

Inter-domain optimization techniques, such as Xensocket [27], Xenloop [23] and Xway [14], have mainly focused on strengthening the throughput of data transmission between VMs that are co-resident within a single physical machine. The general approach commonly adopted by most of these techniques is to build a fast inter-domain channel with shared memory. It is evident that the efficient communication channel will improve the performance of RPC systems, but our experiments show that the data presentation in RPC systems primarily leads to the inefficiencies, instead of the communication channel. More concretely, Table I exhibits the effect of acceleration of Xenloop on ICE [11] with three data types of different complexity (defined in Table IV) in a test machine equipped with Intel Celeron D 331 and 2G RAM. Even under the best circumstances (e.g. 'ICE send ByteSeq'), the performance gain is approximately 25%. Once the data type becomes more complex, the acceleration rate is almost negligible (e.g. 'ICE receive FixedSeq'). It helps prove our assumption: the efficiency of data presentation layer is more important than the data transmission layer for RPC systems in VMMs.

TABLE I
THE EFFECT OF ACCELERATION OF XENLOOP ON ICE.

Benchmark	Scale	w/o Xenloop	w/ Xenloop
ICE send ByteSeq	Mbps	1541	1923
ICE receive ByteSeq	Mbps	818	846
ICE send FixedSeq	Mbps	391	401
ICE receive FixedSeq	Mbps	291	292
ICE send VarSeq	Mbps	67	66
ICE receive VarSeq	Mbps	57	57

D. Shared Memory Facilities

We briefly discuss two systems that offer interfaces to manipulate shared memory in Xen [1] and VMWare [22] respectively.

The research community has already developed some VMI (Virtual Machine Introspection) mechanisms [13], [16], [10] to bridge the semantic gap between the VMM and OS. XenAccess [24] is one of such tools developed for Xen hypervisor. The memory introspection library of XenAccess offers applications executing in dom0 the ability to monitor or manage the memory of another virtual machine.

The Virtual Machine Communication Interface [21] is a VMWare specific infrastructure, which consists of two parts: the datagram API and the shared memory API. The datagram API allows processes in different VMs to send messages to each other. The shared memory API allows an application in a virtual machine to share its memory with other applications that reside in host or another virtual machine on the same host.

III. DESIGN

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

- **Non-intrusiveness**: VMRPC should not add extra complexities to system level components, and only depend on the primitives exported by VMMs.
- **High performance**: VMRPC should enable low latency, high throughput RPC with low CPU consumption.
- **Portability**: VMRPC should support different VMM platforms, 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 could be several different approaches to implementing a high performance RPC system in VMM environment. A straightforward way is to modify the VMM to support a new data channel, but it is not a good idea to add extra functionalities to VMMs, which may introduce security vulnerabilities and complicate the implementation of the VMM. The other solution is to develop a customized kernel module in hostOS and/or guestOS to establish a kernel level fast data channel, but that also means VMRPC would be tightly bound to specific kernel version or type of operating system. At last, we decided to implement VMRPC using only the primitives exported by VMM to the user level, without any modifications to Xen/VMWare and any modules/patches to hostOS/guestOS.

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 II-A: *Problem 1* can be resolved by replacing the socket interface with the VMM platform-specific notification mechanism like the event channel in Xen. We solve *Problem 2* by utilizing shared memory mechanism, as previously done by many inter-domain communication tools like Xway or Xenloop. In order to overcome *Problem 3* and *Problem 4*, we move the memory sharing activities to the user level, where the elimination of data serialization/deserialization operation is made possible. 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 optimization and the inter-VM communication optimization strategies to achieve the goal of high performance.

C. Portability

Under the guidance of the portability principle, VMRPC is divided into three subsystems: notification channel, control channel and transport channel, which are shown in Figure 1. Dividing VMRPC into three subsystems makes it possible of separating most functionalities from the underlying VMM implementation, thus facilitating the porting process maximally.

D. Simplicity

Being different from Xway or Xenloop, VMRPC is not binary-compatible to legacy systems. A clean and well-defined interface is critical to VMRPC. Traditional RPC frameworks are very invasive, requiring language specific tools and code generators to work, and often lead to huge complexity and intricate code modifications. In contrast, VMRPC needs neither IDL nor code generator, because IDL is replaced by a standard C function calling convention, and code generator is replaced by a convenient C preprocessor macro. By keeping VMRPC interface clean and small, it is possible for developers to start writing production-quality code without having to go through a long learning process.

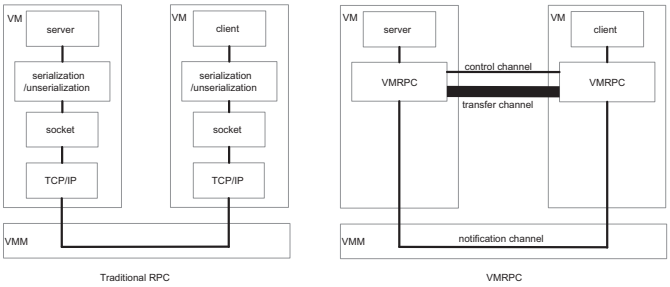


Fig. 1. Architectures of VMRPC and Traditional RPC.

E. Security

VMRPC is specifically tailored towards the needs of an enterprise-class appliance, thus we make some reasonable assumptions as in Fido [8]. First, software components in VMs are assumed to be non-malicious, and granting read-only access to shared memory is acceptable. Second, the possibility of corruptions propagating from a faulty VM to a communicating VM via read-only access of memory is low. Despite of these assumptions, we incorporate several strategies such as managed memory allocation, protection for sharing stack, and control flow verification etc., into VMRPC to promote the whole system security to a significant extent, as detailed in Section IV-C.

IV. Implementation

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

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

Figure 1 also implies the advantages of VMRPC against traditional RPC systems. First, moving communication and control to the user level leaves the kernel (and VMM) only responsible for context switching between the server and

client. Second, VMRPC circumvents the TCP/IP stack, and directly exploits the VMM platform-specific shared memory mechanism to present and transfer data in user space. Meanwhile the expensive serialization/deserialization process is also eliminated. Last, the VMM built-in notification mechanism ensures the minimized latency of 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 copying 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 transfer channel. In Xen, the process is simple: the client allocates a new virtual memory space, then the server maps the corresponding physical page frame in its own virtual address space by using the memory introspection API of Xenaccess: *user_va_map_range*. The case of VMWare is somewhat different, *VMCISharedMem_Create* is launched by server to create a shared memory service, and then the client attaches it 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), Xenaccess consumes only 1.5 seconds while VMCI takes 23 seconds. We observed during the execution of VMCI that the system temporary folder (/tmp directory in Linux) generated a randomly named file whose size is exactly 100M bytes. We speculate that the inefficiency stems from the fact that VMCI is not a shared memory mechanism directly built on page table mapping, based on the file system activities occured in the mapping process. Further optimization to Xenaccess's mapping is possible, but it is beyond the scope of this paper.

Even if creating mappings is expensive (as compared to the runtime overhead), these operations are performed only once at initialization. When the memory mapping is established, all subsequent communication between address spaces will be performed through logical channels that are pair-wise shared between the client and server.

Avoid demand paging: Most modern operating systems implement a demand paging virtual memory architecture. OS allocates a real physical page frame only if an attempt is made to access it. While this strategy works fine in most cases, it is not desirable to our design of memory mapping. In VMRPC, when a mapping operation happens in the server side, we must ensure that there are sufficient physical pages to be mapped, otherwise this operation will fail. This limitation can be resolved by performing a write access to all the pages belonging to that shared memory region, which makes sure there are enough physical memory frames to be attached to each page of the shared virtual memory.

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

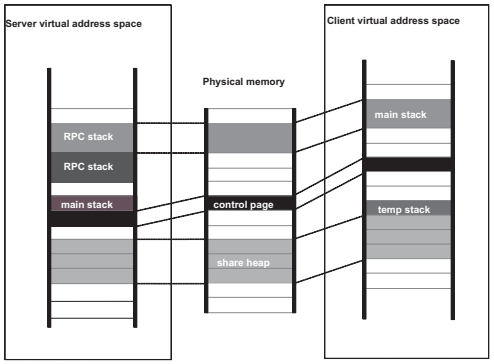


Fig. 2. Virtual address mapping in VMRPC.

Offset handling: Neither Xenaccess nor VMCI supports mapping the virtual memory at a specified address. In Xen, the client's shared region is mapped to an arbitrary address in the server's address space. As a result, the pointer arguments (if any) of function calls on the client side are incomprehensible when used as-is by the corresponding functions on the server side. They need to be translated to appropriate address on the server side in order for the RPC operations to execute properly. Since the length and content of shared memory are identical on both sides, all VMRPC need to do is to add or subtract a constant offset to each pointer argument. VMRPC cannot accomplish this automatically due to its inability of distinguishing pointers from other types of arguments, because there is no explicit type information available in VMRPC. Our choice is to provide another API *VMRPC_offset* that must be invoked to compute the offset between the server and client address space. Details about this API will be given in section V.

B. Transfer Channel

Transfer channel is built on top of *shared heap*. The shared heap is a pre-allocated memory region that is mapped into both of the server and client address space, large volume of data can be directly transferred through it. The shared heap is different to the standard operating system heap, but the two can be used interchangeably by applications. VMRPC provides management APIs to specify the size of the shared heap and set up or destroy a shared heap.

Zero copy: For inter-domain RPCs, exploiting shared memory is a straightforward way to avoid copying from user space to kernel space and vice versa. We ensure data accessibility by mapping the memory in source process's address space to destination process's address space, so that

there are no user layer copies. Kernel layer copies are also avoided by removing the kernel and VMM from the critical data transmission path.

Shared heap size: Since the mapped address of shared region in the server end is totally random, it is very difficult to change the size of shared region dynamically once established. The efficiency of VMRPC depends on the programmer's ability to accurately predict and define the size of a shared heap. Oversized shared heap is wasteful because the corresponding memory pages will be occupied until the end of the RPC procedure. While undersized shared heap may result in allocation failures in the shared region. We are planning to develop dynamic heap management mechanisms to remedy this situation.

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 in the same way as the standard equivalent *free* but in the VMRPC's shared heap. VMRPC also provides APIs such as *VMRPC_heap_setup* and *VMRPC_heap_destroy* to create and destroy user defined shared heaps.

C. Control Channel

Since in VMM the client and server reside in the same hardware (although located in different VMs), it is needless to wrap and express the data in complicated ways.

Control page: Control page serves as the message exchange media. When initializing, VMRPC stores the information about the stack size, heap size and start addresses of stack and heap in it. When the client starts a RPC operation, two types of control information are stored in control page: call index number and ESP value. Call index number is used by the server to find the right servant function in RPC dispatch table. ESP value indicates the stack frame of the current function, since the client's main stack is already mapped to server's address space. Return value is also stored in control page by the server.

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

For each RPC, the client first stores the call index on the top of the current stack into 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 according to the value stored in the control page. When the task finishes, the server switches back to the SMS and writes return value to the control page. Then the client alters the RPC stack with the correct return address by looking for the temporary stack, takes the returned value from the control page and makes this modified RPC stack as the execution stack. By now, a complete two-way RPC operation has been accomplished.

Return address reservation: The control flow information of 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 RPC stack will be overwritten when the callee function is performed on the server side. Thus the client must be able to store the original return address and change the corresponding stack value when the control flow is switched back.

Security issue: Being able to access the stack means the server or client may be able to alter the control flow of the other party, malicious intentions would lead to system crash or exposure of sensitive information. In VMRPC we assume the server is trusted but the client is not. When guestOS launches a RPC operation, the shared stack becomes read-only to guestOS until the server transfers control back to the client. The server will verify the correctness of the return address in stack and clear all sensitive information to guarantee the security of stack sharing. Thus the client has no ability to change the control flow of the server or spy on the data flow of the server.

D. Notification Channel

There are two main reasons that we implemented the notification channel in VMRPC. First, in order to protect the shared stack and heap from concurrent accesses that could produce non-deterministic behavior, we need some kind of synchronization mechanism. Second, the RPC communication semantics requires a way to allow both parties to respond to a remote call or a returned value. VMM-specific asynchronous mechanisms such as the event channel in Xen and VMCI datagram in VMWare, are essential for VMRPC to build the notification channel.

V. VMRPC User Interface

VMRPC provides a simple and clear interface to users, comprising only eight APIs, Table II illustrates a simplified example in which the usage of most of VMRPC APIs is demonstrated in a typical scenario. The server first creates a listening thread by calling *VMRPC_server*. When the client issues a *VMRPC_client* call, they exchange necessary information and build up the control channel and event channel. Then, *VMRPC_server* starts a standard service loop. *VMRPC_heap_setup* is launched in the client to create shared heaps, and it will later be closed by calling *VMRPC_heap_destroy*. The user must use *VMRPC_malloc* and *VMRPC_free* to allocate or release memory blocks from the shared heap.

The most tricky APIs are macro *VMRPC_PROXY* and function *VMRPC_offset*. *VMRPC_PROXY* switches the stack, puts control command and calling index in the control page followed by some stack information, then triggers the notification channel to inform the server. *VMRPC_server* reads the

TABLE II
A SIMPLIFIED EXAMPLE USING VMRPC.

server.c

```
void func1(int i, str* s){
        printf("%d %s\n",i,s);
}
void func2(str* s){
        printf("%s\n",s);
}
vmrpc_call VMRPC_call_list[] = {
        &func1,
        &func2,
}
VMRPC_server();
```

client.c

```
void func1(int, str*);
void func2(str*);
VMRPC_PROXY(func1);
VMRPC_PROXY(func2);
VMRPC_client();
HeapHandle hh=VMRPC_heap_setup(heap_size);
char *str = VMRPC_malloc(13);
strcpy(str, "Hello,world!");
func1(100, VMRPC_offset(str));
func2(VMRPC_offset(str));
VMRPC_free(str);
VMRPC_heap_destroy(hh);
```

TABLE III
THE RESULTS OF LATENCY TEST

	VMRPC	Socket	ICE	XMLRPC
XEN	15μs	40μs	66μs	320μs
VMWare	84μs	90μs	135μs	551μs

TABLE IV
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; //"hello" double d; } Var Var VarSeq[LEN]`	18*LEN

information stored in the control page, and switches to server's RPC service stack to run the locally-defined real function. When finished, *VMRPC_server* records the returned value in the control page, and notifies the client. All the changes are reflected in the RPC stack, shared heap and return value (in control page). Thus after being properly manipulated by *VMRPC_PROXY*, a remote call is actually executed as a local call.

For every RPC call that involves a reference that points to somewhere in the shared stack or heap, *VMRPC_offset* must be invoked to compute the offset between the server and client address space to avoid the inconsistence caused by memory mapping. We perform this calculation in the client where it is easier to obtain the offset information.

VI. EVALUATION

In this section we evaluate the performance of VMRPC by comparing it with two traditional RPC systems (XMLRPC and ICE) and an inter-domain communication optimization system (Xenloop). Three basic performance indicators: latency, throughput and CPU utilization are reported here.

A. Test Setup

Unless otherwise mentioned, all experiments were performed on this machine: Core Duo 6550 2.6GHz CPU with 3GB of memory running Xen 3.1 or VMWare Workstation 6.0 for linux. The hostOS and guestOS in Xen/VMWare are both Fedora 8 (Linux kernel 2.6.21). For each test, the server resided in the Dom0 (or a host in VMWare), the client run in the DomU (or a guest in VMWare). When testing Xenloop, we recompiled the kernel of Fedora 8 and Xen to meet its requirements (Linux kernel 2.6.18 and Xen 3.2).

B. 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 times taken to complete specific computations and data transmissions. For completeness, we also conducted performance measurement about native socket interface by transferring one byte data from the client to server.

As shown in Table III, the numbers indicate the latency in milliseconds averaged across 100,000 null RPC operations. We can find that ICE is a highly efficient RPC system that incurs only a little overhead on top of native socket. But 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 in Xen, but still better than socket and traditional RPCs.

C. Throughput

Simply put, any RPC system's throughput can be calculated as follows: throughput = $\frac{actual\ payload\ per\ RPC}{the\ execution\ time\ of\ a\ RPC}$. The actual payload is the valid information that a servant function actually processes, not including the RPC control information, communication protocol messages and 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 the RPC's total execution time, some of which are relatively stable, while others depend on the characteristics of RPCs. For example, serialization overhead is mainly affected by the size and complexity of the actual payload, the transfer overhead depends on the data volume after serialization.

VMRPC eliminates some kinds of overhead from traditional RPC system, but it also introduces some additional overhead: the pointer argument conversion in *VMRPC_offset*, and the heap management in *VMRPC_malloc* and *VMRPC_free*. To discuss the efficiency of VMRPC in the worst case, in the following tests, the overheads described above are all included in VMRPC's total cost.

Since RPC's throughput is heavily dependent on the type of data transmitted, we define three kinds of data with different complexity: byte sequence, fixed-length structure sequence and variable-length structure sequence, their definitions and actual payloads are summarized in table IV.

The LEN in table IV represents the length of sequences ranging from 100 to 10000000, so we can observe RPC's throughput for payloads ranging from 1KB to about 100MB. We analyze RPC's receive and send operation separately. Figure 3 shows the reported throughput as a function of length of sequence, as compared to that for ICE and XMLRPC. The first three sub-figures depict the throughput of three RPC systems in Xen. The latter three is for VMWare.

The gradual increase of the 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 Figure 3 (a) (d), VMRPC achieves at least up to 10 times the throughput of ICE and XMLRPC in the peak case (the sequence length reaches 10000). The relative discrepancy between VMRPC and ICE/XMLRPC is widening with the increasing complexity of the message, although the absolute value of VMRPC throughput decreases (comparing (a) with (c), and (d) with (f)).

When the message size increases, the performance becomes dominated by the overhead of actually transferring the data. ICE and XMLRPC decrease rapidly due to the serialization, coping, transmission and context switch overhead. In VMRPC, the memory allocation, pointer address conversion and memory copy costs also lead to the decline. A strange phenomenon is that VMRPC (VMWare) sees a large dropoff when workload exceeds 1000000, while VMRPC (Xen) keeps stable on the same load. With further investigations we found that when writing more than 50M bytes data to the shared memory, VMCI's performance is lower than Xen, resulting in higher overhead. We attribute this to the same reason that causes the deficiency of memory mapping.

It is easy to find that 'VMRPC recv' performs better than 'VMRPC send' because of a reduction in data replication operations. On the contrary, 'ICE recv' is worse than 'ICE send' due to an extra copy. At last, we notice the curves of ByteSeq and FixedSeq of VMRPC have a similar shape, because the data of both types is intrinsically contained in a continuous region of virtual address space, resulting in less memory allocations and copy operations. The data of type VarSeq is usually composed by a large number of small and discrete memory blocks, which increases the frequency of allocating and copying operations. As a result, the turning point of curves of type VarSeq comes earlier than the other two types of data.

D. CPU Utilization

We used oprofile [18] to measure the relative CPU consumption for data serialization and deserialization. The same sequences of three structures in the preceding section are also used in this evaluation. Table V presents the results of sending 1,000 sequences containing 10,000 structures from client to server.

TABLE V
THE RESULTS OF CPU UTILIZATION TEST.

	Byte send	Byte recv	Fix send	Fix recv	Var send	Var recv
VMRPC	170	116	218	132	19784	13401
ICE	419	720	203462	231238	219831	318779
XMLRPC	59037	131109	1232753	6127841	1116041	4869226

TABLE VI
HEAP SETUP OVERHEAD.

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

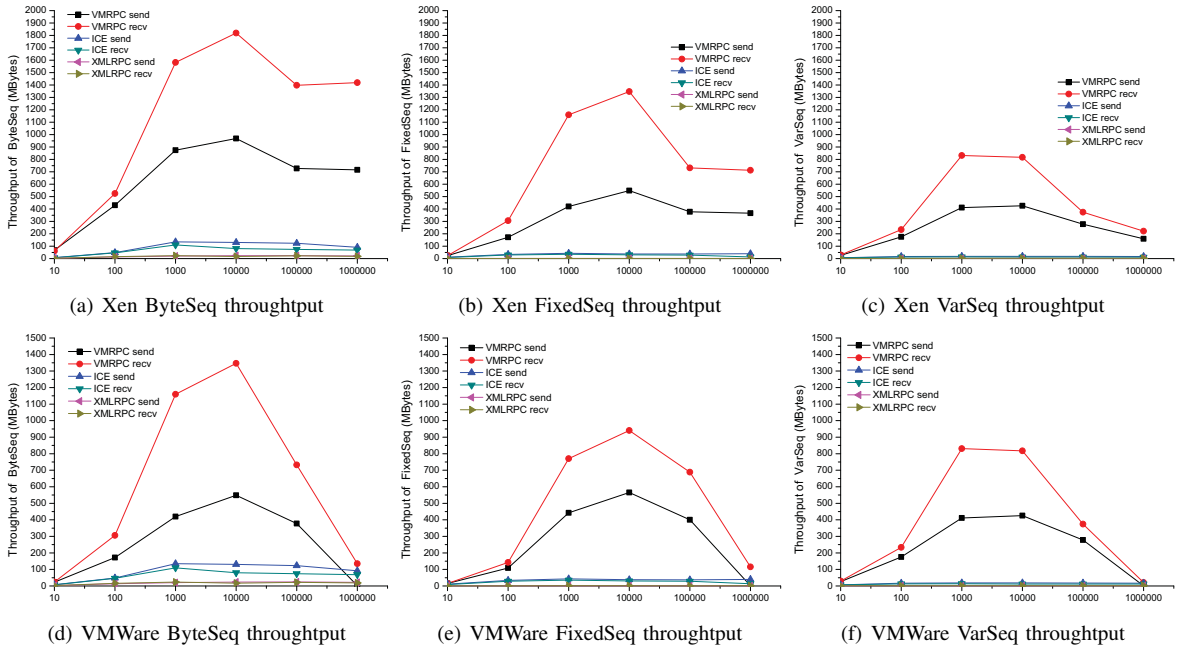
As shown in Table V, XMLRPC is much more CPU intensive than ICE and VMRPC, and it costs even one hundred times more CPU cycles in some cases. For ICE compared with VMRPC, it is about a factor of ten. Also note that recv in XMLRPC uses around four times as many CPU cycles as send. We attribute this to the process of decoding large XML data sources arriving over the wire.

E. Heap setup overhead

Table VI shows the time spent on mapping shared regions with varied size in Xen and VMWare. Although the setup overhead for large heap is not ignorable in cases where the heap size is larger (in the VMCI case, we need approximately 22 seconds to set up a 100M heap), we regard this as the initialization overhead. And it will not have any negative impacts on system performance at runtime.

F. VMRPC vs Xenloop

Having discussed the design, implementation and evaluation of VMRPC, one may question that if the performance of VMPRC is superior to a traditional RPC system enhanced by an inter-domain communication optimization system. In order to show the comparative advantage gained by VMRPC over the inter-domain communication optimization systems, we compare VMRPC with Xenloop [23] due to its desired features such as simplicity, transparency and high performance. Unfortunately, VMRPC's current implementation does not support communication between DomUs in Xen, while Xenloop only offers communication optimization between two co-located DomUs. Thus we have to analyze the relative speedup of throughput in their respective testing environments: the throughputs of ICE between DomUs are presented in the second column of TableVII, we fill 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. Based on the results, we can see the speedup from VMRPC is far higher than those of Xenloop. In addition, the advantage of VMRPC becomes clearer as the complexity of payload grows.



(a) Xen ByteSeq throughtput (b) Xen FixedSeq throughtput (c) Xen VarSeq throughtput

(d) VMWare ByteSeq throughtput (e) VMWare FixedSeq throughtput (f) VMWare VarSeq throughtput

Fig. 3. The results of throughtput test.

TABLE VII
THE THROUGHPUT COMPARISON BETWEEN XENLOOP AND VMRPC.

Mbps	ICE DomU	ICE DomU Xenloop	Xenloop Relative Speedup	ICE Dom0	VMRPC Dom0	VMRPC Relative 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%

TABLE VIII
STATISTICS OF BENCHMARK APPLICATIONS.

	API calls	GPU RAM	Data Volume
AlignedTypes (AT)	1990	94.00MB	611.00MB
BinomialOptions (BO)	31	0.01MB	0.01MB
BlackScholes (BS)	5143	61.03MB	76.29MB
ConvolutionSeparable (CS)	48	108.00MB	72.00MB
FastWalshTransform (FWT)	144	128.00MB	128.00MB
MersenneTwister (MT)	24	91.56MB	91.56MB
MonteCarlo (MC)	53	187.13MB	0.00MB
ScanLargeArray (SLA)	6890	7.64MB	11.44MB

VMRPC performs surprisingly about 122 times better than ICE in case of 'Var recv' as shown in the last row of TableVII.

G. Case Study: vCUDA

vCUDA is a GPGPU high performance computing solution for virtual machines. vCUDA allows applications executing within virtual machines (VMs) to leverage hardware acceleration, which can be beneficial to the performance of a class of high performance computing (HPC) applications. vCUDA works by hijacking CUDA dynamic libraries in user level to redirect CUDA APIs to a remote server residing in Dom0, which fulfills real computational tasks in GPU. The XMLRPC was used to implement this redirection. In our former work [20], we used some CUDA SDK samples to validate the feasibility of the vCUDA architecture, but the poor performance of XMLRPC has enormous negative impact on the overall system performance. By applying VMRPC to vCUDA project, we expect a decent improvement in performance.

Table VIII shows the statistical characteristics of the benchmarks, such as the quantity of API calls, the device memory size they consume and the data volume transferred from or to GPU device. The benchmarks range from simple data management to more complex Walsh-Transform computation and MonteCarlo simulation.

Figure 4 exhibits the performance comparison between new vCUDA (with VMRPC) and old vCUDA (with XMLRPC) in Xen. As shown in Figure 4, the performance boosting for the benchmarks (such as AT, FWT, and MT) that involve large data transfer is more obvious than those that transmit less data. This is mainly due to the local RPC and shared memory optimizations. The VMRPC enhanced vCUDA system is competitive to the native execution in most cases. From the figure, we can see that the low latency characteristic of VMRPC also helps improve the performance. For example, the SLA benchmark involves little data flow, but it gains reasonable performance improvement in spite of the 6890 RPC calls.

VII. RELATED WORK

We present related work by organizing the literature into local RPC optimization and inter-domain communication optimization.

LRPC [2] addressed how local RPC can be implemented with minimal overhead. It emulates the native local procedure

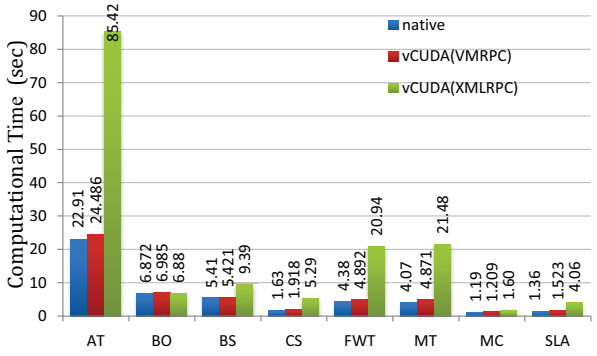


Fig. 4. vCUDA performance results with XMLRPC, VMRPC and native execution.

call model, and no extra message-passing but the original procedure-call convention is needed. By using client's thread to execute the requested service in server's address space, LRPC sets up a simple control transfer model. [7] extended LRPC to 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 out of the kernel and supporting them at the user level within each address space. Nevertheless URPC is still an intra-OS RPC, more precisely, an IPC tool. SHRIMP RPC [5] actually is another version of URPC in a distributed memory architecture.

XenSocket [27] is a one-way communication channel between two VMs based on shared memory. It defines a new socket type, with associated connection establishment and read-write system calls that provide the interface to developer by leveraging the underlying inter-VM shared memory mechanism. IVC [12] is an user level communication library intended for message passing HPC applications. It provides a socket-style API. Xway [14] intercepts TCP/IP stack beneath the socket layer, and provides transparent inter-domain communication with extensive modifications to network protocol stack in the operating system. Xenloop [23] is a fully transparent inter-domain network channel, which exploits the netfilter hooks in Linux to intercept outgoing network packets and shepherds the packets destined to co-resident VMs through a inter-domain shared memory channel. One main drawback of Xenloop is that it does not support the communication between Dom0 and DomU. Fido [8] is a high performance inter-domain communication mechanism tailored for enterprise appliances. Fido is perhaps the closest equivalent to VMRPC based on the techniques implemented in these two systems, but it is not designed specifically as a RPC system.

VIII. Conclusions and Future Work

In this paper, we have described the design, implementation and performance evaluation of VMRPC. VMRPC sacrifices transparency for extremely efficient communication, and provides fast responsiveness and high throughput when deployed for collaborative components in virtualization environments. It is specifically designed for applications where high volume data transmission between VMs is desired. Our evaluation shows that the performance of VMRPC is an order of magnitude better than traditional RPC systems and existing alternative inter-domain communication mechanisms. As our future work, we plan to develop new features such as dynamic heap management, non-blocking RPC, and add support for communication between domUs.

Acknowledgments We thank the anonymous reviewers for their helpful feedback. This research was supported in part by the National Basic Research Program of China under grant 2007CB310900, the National Science Foundation of China under grants 60803130 and 60703096, and the Fundamental Research Funds for the Central Universities of China.

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] B. Bershad, R. Draves, and A. Forin. "Using Microbenchmarks to Evaluate System Performance." Third Workshop on Workstation Operating Systems, April 1992.
[5] 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.
[6] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. "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.
[7] 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.
[8] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson. "Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances". In Proc. USENIX, June 2009.
[9] Cisco Systems. http://www.cisco.com/products.
[10] T. Garnkel and M. Rosenblum. "A virtual machine introspection based architecture for intrusion detection". In Proc. the Network and Distributed Systems Security Symposium, February 2003.
[11] M. Henning. "A new approach to object-oriented middleware". IEEE Internet Computing 8 (2004), pp. 66-75.
[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] A. Joshi, S. King, G. Dunlap, and P. Chen. "Detecting past and present intrusions through vulnerability-specific predicates". In Proc. ACM Symposium on Operating Systems Principles (SOSP), pages 1-15, Oct 2005.
[14] 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.
[15] 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.
[16] M. Laureano, C. Maziero, and E. Jamnhour. "Intrusion detection in virtual machine environments". In Proc. 30th EUROMICRO, pp.520-525, 2004.
[17] NetApp Storage Systems. http://www.netapp.com/products. .
[18] OProfile. http://oprofile.sourceforge.net/news/
[19] M. Schroeder and M. Burrows. "Performance of Firefly RPC". In Proc. 12th ACM symposium on Operating systems principles, pp.83-90, November 1989.
[20] L. Shi, H. Chen, and J. Sun. "vCUDA: GPU Accelerated High Performance Computing in Virtual Machines". IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rome, Italy, May, 2009.
[21] VMCI. http://pubs.vmware.com/vmci-sdk/index.html.
[22] VMWare. http://www.vmware.com.
[23] J. Wang, K. Wright, and K. Gopalan. "XenLoop: A Transparent High Performance Inter-VM Network Loopback". In Proc. 17th international symposium on High performance distributed computing, Boston(HPDC), MA, USA, June 2008.
[24] XenAccess. http://xenaccess.sourceforge.net.
[25] XMLRPC. http://www.xmlrpc.com.
[26] C. Yarvin, R. Bukowski, and T. Anderson. "Anonymous RPC: Low Latency Protection in a 64-Bit Address Space". In Proc. USENIX, June 1993.
[27] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. "Xensocket: A high-throughput interdomain transport for virtual machines". In Proc. Middleware, 2007.