Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface
Princeton University Technical Report TR-512-96

A shorter version of this report is submitted for publication.

Angelos Bilas and Edward W. Felten
Department of Computer Science, Princeton University, Princeton NJ 08544 USA
{*bilas,felten*}*@cs.princeton.edu*

Abstract

The emergence of new network interface technology is enabling new approaches to the development of communications software. This paper evaluates the SHRIMP virtual memory mapped network interface by using it to build two fast implementations of remote procedure call (RPC).

Our first implementation, called vRPC*, is fully compatible with the* SunRPC *standard. We change the RPC runtime library; the operating system kernel is unchanged, and only a minimal change was needed in the stub generator to create a new protocol identifier. Despite these restrictions, our* vRPC *implementation is several times faster than existing* SunRPC *implementations. A round-trip null RPC with no arguments and results under* vRPC *takes about 33 microseconds.*

Our second implementation, called ShrimpRPC*, is not compatible with* SunRPC *but offers much better performance.* ShrimpRPC *specializes the stub generator and runtime library to take full advantage of SHRIMP's features. The result is a round-trip null RPC latency of 9.5 microseconds, which is about one microsecond above the hardware minimum.*

1 Introduction

Much is known about how to optimize remote procedure call (RPC) mechanisms on traditional workstation networks[2, 14, 17, 21]. The main effort in previous work was to reduce or avoid copying, to make traps and context switches fast, and to take advantage of common-case behavior. The emergence of new multiprocessor network interfaces opens new possibilities for constructing network software. It is not always clear, however, which interface is most appropriate for which task.

The SHRIMP project [5, 6, 9] at Princeton University supports user level communication between processes by mapping memory pages between virtual address spaces. This Virtual Memory Mapped network interface seems to have many advantages including flexible user-level communication, and very low overhead to initiate data transfer.

In this paper we present two fast RPC libraries implemented using virtual memory mapped communication between user processes. The first one (*vRPC*) meets the *SunRPC* interface and achieves a round trip latency of 33 microseconds for a null call with no arguments, without compromising compatibility with the *SunRPC* standard. The server in *vRPC* can service at the same time clients using both the old (UDP- and TCP-based) and the new (SHRIMP-based) protocols. We implement two versions of *vRPC* for SHRIMP by rewriting *SunRPC* and taking advantage of the new features SHRIMP provides.

Our experiments with *vRPC* show that even without changing the stub generator or the kernel, RPC can be made several times faster on the new network interface than it is on conventional networks. *vRPC* outperforms by more than a factor of 4 the best, to our knowledge, reported implementation for fast networks [7], with a round trip time of about 33μs for a null RPC. Even greater gains could be achieved by applying well-known techniques that rely on changes to the stub generator.

The second library, *ShrimpRPC*, is a full-functionality RPC system but is not compatible with any standard. *ShrimpRPC* achieves a round trip latency of 9.5 microseconds, or about one microsecond above the hardware minimum for a round-trip communication. The *ShrimpRPC* library and stub generator are optimized to take advantage of SHRIMP.

In the remainder of the paper, we give some background information about SHRIMP and the new network interface in Sections 2 and 3. Section 4 gives an outline of *SunRPC*. Section 5 describes *vRPC* and presents measurements. Section 6 provides detailed instruction counts for *vRPC*. *ShrimpRPC* is presented in section 7. Finally we compare the different transfer mechanisms provided by the SHRIMP hardware in the context of *vRPC* in Section 8, discuss related work in section 9, and draw conclusions in Section 10.

2 The SHRIMP network interface

The SHRIMP project at Princeton studies how to provide high-performance communication mechanisms to integrate commodity desktop systems such as PCs and workstations into inexpensive, high-performance multicomputers. End-to-end latency and bandwidth available to user processes are the primary performance metrics. The challenge is to provide a low-latency, high-bandwidth communication mechanism whose performance is competitive with or better than that of specially designed multicomputers.

The network interfaces of existing multicomputers and workstations require a significant amount of software overhead to implement message-passing protocols. The main reason for such high overheads is that these multicomputers use network interfaces that require a significant number of instructions at the operating system and user levels to provide protection, buffer management, and message-passing protocols. In these designs, communication is treated as a service of the operating system. This is expensive because it requires several crossings between user level and kernel level for each message, and also because it prevents applications from customizing their use of the communication hardware.

The computing nodes of SHRIMP are Pentium PCs, and the interconnection network is the same one used in the Intel Paragon. The key hardware component is the network interface board, which supports the virtual memory mapped communication (VMMC) model, to provide low-overhead, protected, user-level communication. For more details on the SHRIMP architecture the reader can consult [5, 6, 9]. VMMC is discussed in the next section.

3 Virtual Memory-Mapped Communication

Virtual memory-mapped communication (VMMC) [10] was developed in response to the need for a basic multicomputer communication mechanism with extremely low latency and high bandwidth. These performance goals are achieved by allowing applications to transfer data directly between two virtual memory address spaces over the network. The basic mechanism is designed to efficiently support applications and common communication models such as message passing, shared memory, and client-server.

The VMMC mechanism consists of several calls to support user-level buffer management, various data transfer strategies, and transfer of control.

Figure 1 shows the latency and bandwidth delivered by the SHRIMP VMMC layer [13].

3.1 Import-Export Mappings

In the VMMC model, an *import-export mapping* must be established before communication begins. A receiving process can *export* a region of its address space as a receive buffer together with a set of permissions to define access rights for the buffer. In order to send data to an exported receive buffer, a user process must *import* the buffer with the right permissions.

After successful imports, a sender can transfer data from its virtual memory into imported receive buffers at user-level without further protection checking or protection domain crossings. Communication under this import-export mapping mechanism is protected in two ways. First, a trusted daemon process implements import and export operations. Second, the hardware virtual memory management unit (MMU) on an importing node makes sure that transferred data cannot overwrite memory outside a receive buffer.

The *unexport* and *unimport* primitives can be used to destroy existing import-export mappings. Before completing, these calls wait for all currently pending messages using the mapping to be delivered.

3.2 Transfer Strategies

The VMMC model defines two user-level transfer strategies: *deliberate update* and *automatic update*. Deliberate update is an explicit transfer of data from a sender's memory to a receiver's memory.

In order to use automatic update, a sender *binds* a portion of its address space to an imported receive buffer, creating an *automatic update binding* between the local and remote memory. All writes performed to the local memory are automatically propagated to the remote memory as well, eliminating the need for an explicit send operation.

An important distinction between these two transfer strategies is that under automatic update, local memory is "bound" to a single receive buffer at the time a binding is created, while under deliberate update there is no fixed binding between a region of the sender's memory and a particular receive buffer. Automatic update is optimized for low latency, and deliberate update is designed for flexible import-export mappings and for reducing network traffic.

Automatic update is implemented by having the SHRIMP network interface hardware snoop all writes on the memory bus. If the write is to an address that has an automatic update binding, the hardware builds a packet containing the destination address and the written value, and sends it to the destination node. The hardware can combine writes to consecutive locations into a single packet.

Deliberate update is implemented by having a user-level program execute a sequence of two accesses to addresses which are decoded by the SHRIMP network interface board on the node's expansion bus (the EISA bus). These accesses specify the source address, destination address, and size of a transfer. The ordinary virtual memory protection mechanisms (MMU and page tables) are used to maintain protection [4].

VMMC guarantees the in-order, reliable delivery of all data transfers, provided that the ordinary, blocking version of the deliberate-update send operation is used. The ordering guarantees are a bit more complicated when the non-blocking deliberate-update send operation is used.

The VMMC model does not include any buffer management since data is transferred directly between user-level address spaces. This gives applications

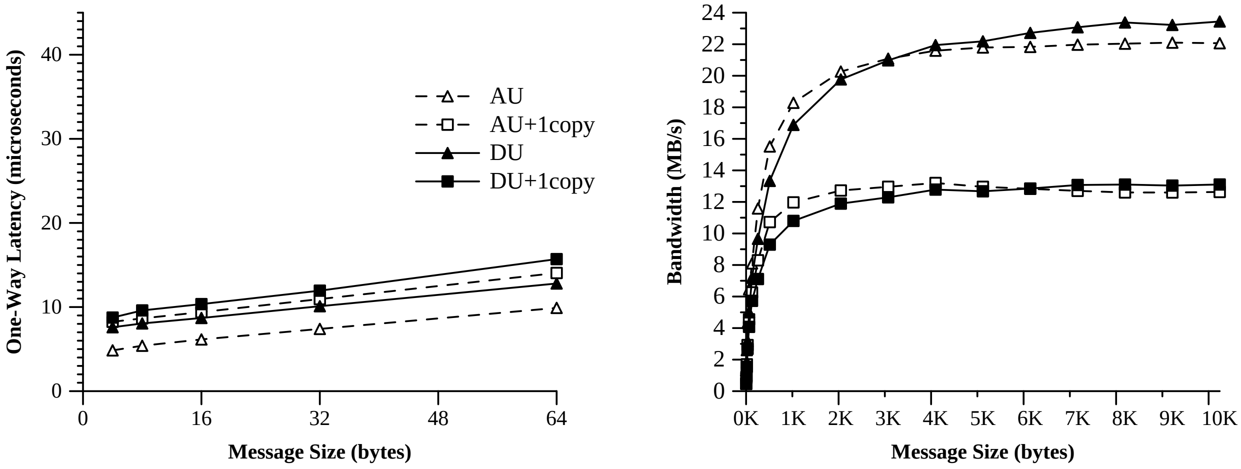

Figure 1: Latency and bandwidth delivered by the SHRIMP VMMC layer using automatic update (AU) and deliberate update (DU).

the freedom to utilize as little buffering and copying as needed. The model directly supports zero-copy protocols when both the send and receive buffers are known at the time of a transfer initiation.

The VMMC model assumes that receive buffer addresses are specified by the sender, and received data is transferred directly to memory. Hence, there is no explicit receive operation. CPU involvement in receiving data can be as little as checking a flag, although a hardware notification mechanism is also supported.

3.3 Notifications

The *notification* mechanism is used to transfer control to a receiving process, or to notify the receiving process about external events. It consists of a message transfer followed by an invocation of a user-specified, user-level handler function. The receiving process can associate a separate handler function with each exported buffer, and notifications only take effect when a handler has been specified.

Notifications are similar to UNIX signals in that they can be blocked and unblocked, they can be accepted or discarded (on a per-buffer basis), and a process can be suspended until a particular notification arrives. Unlike signals, however, notifications are queued when blocked. Currently we have an implementation of notifications on top of UNIX signals, which works correctly but is slow. Soon, we expect to have an implementation similar to active messages [12], and thus to have much better performance than signals in the common case.

VMMC provides a call that allows a process to block until a notification arrives for it. When a process is waiting for a VMMC message to arrive, this call can be used to switch easily between spinning and blocking as appropriate.

4 SunRPC

SunRPC [19] is a widely used remote procedure call interface and specification. *SunRPC* consists of a set of library functions and a stub generator that follows the XDR [18] standard for data representation. *SunRPC* is a single thread implementation.

The general structure of *SunRPC* is shown in Figure 2. It is implemented as a series of layers, each providing services to the layers above.

- The *network* layer implements the `read` and `write` system calls that transfer data across the network.
- The *stream* layer does buffer management. It provides a set of functions to write data (byte strings, integers, etc.) to, or read data from, a stream. The stream layer hides the details of buffer management and network packet size from the higher layers. Its existence is very important to the performance of standard *SunRPC* implementations, since it reduces accesses to the expensive lower layers.
- The *XDR* [18] layer implements the XDR data representation specification, which insulates the higher layers from issues of machine-specific data representation. Data transferred between nodes in a network are translated to XDR format before sending, and translated back from XDR when received. The XDR layer provides a set of functions to send and receive data of a certain set of types: byte streams, longs, strings, unions etc.
- The RPC library implements most of the *SunRPC* protocol, including the management of client-server connections.
- The stub generator RPCGEN produces RPC stubs based on an interface definition supplied by the user.

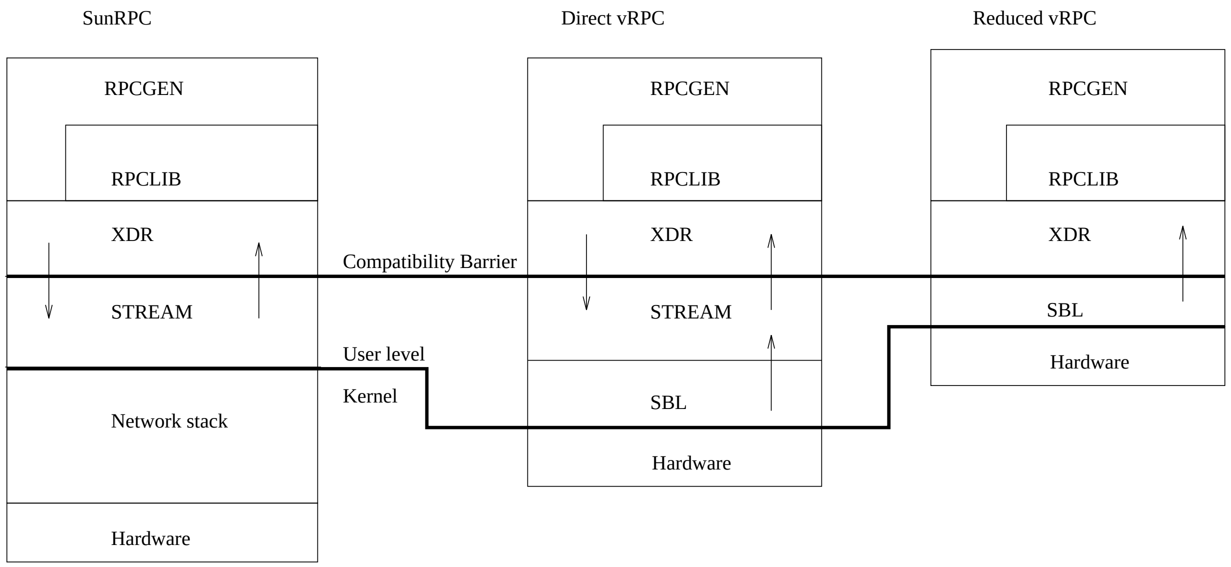

Figure 2: The structure of the different implementations. The arrows indicate the direction and number of copies at user level.

As Figure 2 shows, the interface below the XDR layer separates the machine-dependent code from the machine independent layers.

Figure 2 also shows where copying takes place. Data is copied from the user buffers into the stream buffers and then passed to the operating system functions. In *SunRPC* there are two copies per side per RPC call at user level, plus the copies necessary at protection boundary crossings (one or two), and potentially copies in the kernel between kernel-driver buffers. These amount to a total of at least six copies per RPC call.

Figure 3 shows the common execution path in an RPC. The client sends the header and then the data. When the server replies the client receives first the header of the reply message and then the results. Because of the multiple layers, each call to XDR to send a datum results in a chain of several procedure calls.

5 *vRPC*

Our strategy in implementing *vRPC* on SHRIMP was to change as little as possible, and to remain fully compatible with existing *SunRPC* implementations. In order to meet these goals, we changed only the runtime library; we made minor changes to the stub generator[1] and we did not modify the kernel.

We used two main techniques to speed up the library. First, we re-implemented the network layer directly on top of the SHRIMP network interface.

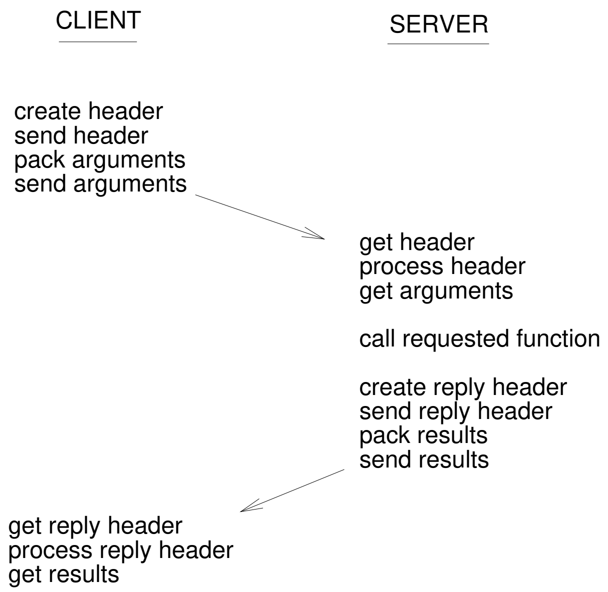

Figure 3: The common execution path in a RPC.

[1] We only added support for the new protocol, so that code for it is automatically generated.

Because the SHRIMP interface is simple and allows direct user-to-user communication, our implementation of the network layer is much faster than the standard one. Our second optimization was to collapse the stream and SBL layers into a new single thin layer that provides the same functionality, thus reducing the number of layers by one.

In Section 5.1 we describe the communication setup between the client and the server. Section 5.2, 5.3 and 5.4 present *Direct vRPC*, *Reduced vRPC* and *Optimized vRPC* respectively. Section 5.4 the new protocol. In Section 5.4 we present *Reduced vRPC*. Section 5.5 describes a modified version of *Optimized vRPC* that avoids copying on the receive side. In Section 5.6 we summarize the different versions and in Section 5.7 we present measurements.

5.1 Communication setup

For *vRPC* a pair of mappings is established between every client and the server. Although creating mappings is expensive (relative to the very inexpensive communication) because exporting and importing buffers requires system calls, these operations are performed only once, during the initialization phase. In that sense RPC is a typical case of communications software where one can separate the expensive setup phase from the common case. The VMMC interface takes advantage of this.

The client and server use the normal portmapper interface for exporting and binding to services. After the client has found the port where the server is listening, the client establishes an ordinary Internet TCP connection as in *SunRPC*. This connection is used to exchange information about which VMMC buffers will be used. If the TCP connection breaks then each side assumes that the other side has terminated and the VMMC mappings are discarded. The existing TCP mechanism is a very convenient way to set up the communication channel and to detect failures.

After the initialization phase no data is sent over the TCP channel.

5.2 Replacing the Network Layer

One of our goals was to evaluate the performance gain to programs from eliminating kernel involvement in communication. Thus we first replaced the kernel layer with another, built on top of the VMMC interface, with almost no other changes to the code. This resulted to the first version of *vRPC*, *Direct vRPC*.

As mentioned before, the network layer provides a pair of functions (readtcp and writetcp) to read data from, and write data to, the network (via Internet-domain sockets). We replaced these two functions with our own versions (readvmmc, writevmmc) that send and receive data using VMMC between the server and the client.

The following paragraphs describe the basic issues in the implementation of the new layer.

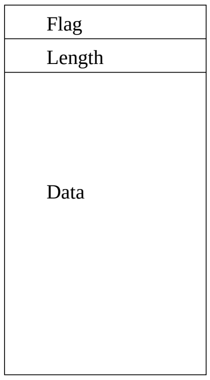

Figure 4: Data structure used by *Direct vRPC* for each stream. Flag and length are reserved locations for the protocol.

5.2.1 The SHRIMP communication layer

The new layer consists of two functions that manage communication buffers. Transfers of data are performed using the VMMC communication mechanisms (*deliberate update and automatic update*). Receiving data does not require any explicit action, because the VMMC model delivers arriving data directly to memory with no CPU involvement. The number of copies on the sending and receiving sides depends on the mode of transfer and will be analyzed later.

The problems that need to be solved for buffer management are *synchronization,* which involves letting the receiver know about data arrival, and *flow control* for the finite shared buffer space.

Note that in the original implementation the network protocol stack took care of the corresponding problems at the lowest level (although at considerable cost).

Conceptually what is needed is a stream between the sender and the receiver. The sender sees only one end of the stream and the receiver the other. Each side sends or receives data without having to worry about the details of the underlying implementation.

This bidirectional VMMC stream consists of two low-level communication functions and the data structure shown in Figure 4. A stream buffer has arbitrary size. The first two words of the buffer are reserved for protocol control information: a synchronization flag (`flag`) and a counter (`length`) of the data valid in the buffer.

When the sender wants to send data, it checks `length` to see whether there is space in the buffer. If the data fits in the buffer, the sender updates the local information, sends the data, and also updates and sends `length.` The receiver checks for new data using `length` and the starting position in the buffer of the new data to arrive.

If the data does not fit in the buffer, the sender fills the buffer and waits for the receiver to set `flag.` This notifies the sender that all the data have been consumed and the buffer can be reused. When the receiver realizes that the buffer is full, after consuming the data, it sets `flag` and adjusts the local information. After this, the sender and the receiver can use the buffer independently.

The protocol shown in Figure 5 is implemented separately in both directions to form a bidirectional channel. The protocol allows the sender and the receiver to send and receive data with no handshaking as long as there is enough space. There is very little handshaking (just one message) when the buffer fills.

In this protocol, the sender will only rarely be delayed. In a client-server application both sides follow a specific protocol. If the sender needs to send information then the receiver needs to receive the same information. If the buffer fills when only part of the information has been sent, then the receiver should be waiting for the rest of the information, so it can immediately detect the situation and unblock the sender. This is especially true for RPC where all the information transfer takes place before and after the computation. The server, for example, is not in the middle of some time-consuming computation while the client is trying to send the rest of the data. Moreover if the server is doing work for another client then this client would have had to wait anyway.

Another way to view the protocol mechanism is as the implementation of a cyclic shared queue; a more direct implementation of a shared queue would be slightly less efficient.

Waiting in the above protocol is implemented by spinning. Timeouts are used to avoid indefinite postponement. More sophisticated techniques for waiting are given in [15].

To summarize, the first version of *vRPC*, *Direct vRPC*, replaces the stream layer with a simple stream communication abstraction implemented directly on top of the VMMC interface. The are still six copies per RPC. Elimination of kernel involvement in the communication path is the only difference from *SunRPC*. Figure 2 shows the overall structure of *vRPC*.

5.3 Collapsing Layers

As noted above, in standard implementations of *SunRPC*, the stream layer is needed to decouple the XDR layer from the network layer, because calls to the network layer are expensive system calls. Now that the communication is done at user level there is no need for the stream layer. In fact, as we described above, the VMMC stream layer of *Direct vRPC* implements a stream. Eliminating the stream layer of *SunRPC* leads to the second version of *vRPC*, *Reduced vRPC*, as shown in Figure 2.

Apart from considerably reducing the layering overhead, this change also eliminates two copies per node per RPC. The only copies left are the transfers of data between library and user buffers on the receiving side. This copy is essential to maintain *SunRPC* semantics.

These two improvements, eliminating the overhead of a whole layer and avoiding copying, are due to

1. the low cost of the user level communication mechanisms which makes the extra stream layer unnecessary, and
2. the direct placement of data in virtual memory.

Moreover, the fact that the client and the server run on nodes of the same multiprocessor leads to two additional improvements. There is no need for translation of the data to and from an agreed transmission format. Also, authentication can be done once, when the SHRIMP connection is set up, rather than on every RPC.

5.4 Tuning *vRPC*

Reduced vRPC is much faster than *SunRPC*, and in fact is faster than any previous RPC implementation that we are aware of. We achieved this performance by merely porting and adapting existing code to the SHRIMP network interface. We can improve the performance further by tuning the RPC library for the SHRIMP and taking advantage of the *SunRPC* semantics and implementation (Figure 5).

In the protocol described above, each send operation consists of two parts: sending the data, and sending the length of the data. Reducing the number of updates of the control information reduces the data transfers but increases the latency of each transfer and reduces the amount of overlap between communication and computation. We modified *Reduced vRPC* to update the control information immediately after every data packet (in the stream protocol) is sent.

As mentioned above VMMC supports two different modes of transfer: deliberate update and automatic update. Performance can be improved by properly choosing when to use each mode. In *Direct vRPC* and *Reduced vRPC* we used synchronous deliberate update for all the transfers as a first approach. The use of synchronous deliberate update everywhere is overkill, since there are cases where it is guaranteed that the data will not be overwritten for some time. For example the header of the call will not be changed until the reply has arrived. The same is true for relatively small transfers. In the current implementation of *SunRPC* the client has to wait for a call to finish before proceeding with the next one. Using asynchronous deliberate update for almost all transfers is safe and increases overlapping of communication and computation.

Automatic update does not require any user code for initiation. The hardware detects accesses to the memory and performs the transfer to the corresponding remote location. This is a low latency mechanism that can be used for small transfers. Using automatic update for protocol control information further reduces the latency. An artifact of using automatic update to send the length is that asynchronous deliberate update cannot be used without extra synchronization, since automatic updates can overtake pending asynchronous deliberate updates, resulting in out of order data transfer.

Automatic update can operate in three different modes, with the mode settable on a per-page basis. It can either send each word of data immediately as a network packet (*single-write*), or accumulate writes into a single packet until a non-consecutive location

Receiver

```
while (there is no new data) do nothing;
read data
if (buffer is full) {
  set length 0
  send flag NEXT_PAGE_ENABLE
  set flag NEXT_PAGE_DISABLE
}
```

Sender

```
if (buffer is full){
  wait until flag NEXT_PAGE_ENABLE
  set length 0
  send flag NEXT_PAGE_DISABLE
}
send data
send length
```

Figure 5: The SHRIMP stream protocol.

is written (*block-write*), or accumulate writes until either a timeout occurs or a non-consecutive location is written (*timer-write*). Using block-write whenever more than one consecutive word is transferred in automatic update reduces the latency and increases the bandwidth since one header is used for more data. However, single-write is better for control information since these are single word transfers.

One more possible target for optimization is the function that performs the deliberate update. The ordinary library call to initiate a deliberate update send must check for errors (such as null pointers), and must break transfers whenever they cross page boundaries. Since the *vRPC* library has more knowledge and has control over buffer allocation and data layout, a more lightweight form of the send function could be used.

We tuned *Reduced vRPC* to take advantage of these opportunities. This resulted in the final version of *vRPC*, *Optimized vRPC*, which uses single-write automatic update to transfer control information. For transferring data both synchronous deliberate update and block-write automatic update can be used. We compare these configurations in section 8.

5.5 No-copy *vRPC* implementation

As mentioned above *Optimized vRPC* does one copy per side for a total of two per call (not counting the actual data transfers). It is possible to eliminate copies and have the data appear on the bus only during transfers across the nodes. This is a deviation from the initial constraint, that the stub generator should remain unchanged. Given the current implementation of *SunRPC* it is possible to avoid copying by either slightly modifying the stub generator or having the user modify the code that the generator produces. Only minor changes are required.

We modified *Optimized vRPC* to eliminate copying. In this version the server has to consume the data before the client is able to send more. This is already dealt with since the server has to finish processing the current call before going to the next. Problems could arise if the server keeps data across calls, but this is a highly unlikely case. Moreover this style of programming is not in line with the remote procedure call concept.

Table refNoCopy_AU in Appendix A presents timing measurements for this version of *Optimized vRPC*.

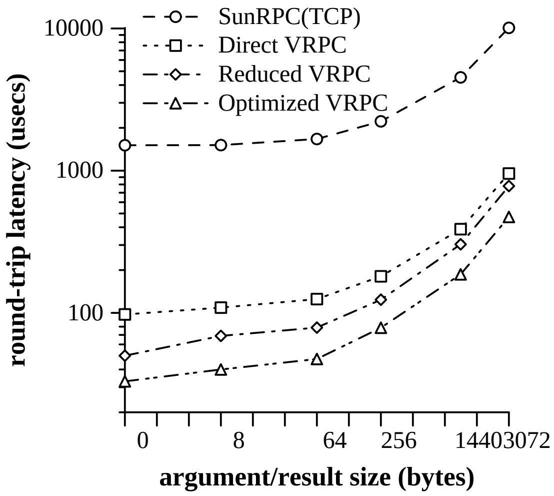

Figure 6: *SunRPC* and *vRPC* latency.

5.6 Summary

To summarize, there are three versions of *vRPC*. In the first version, *Direct vRPC*, only the network layer is replaced, eliminating all system calls from the RPC path. The second version, *Reduced vRPC*, has a simpler structure, and involves considerably less copying. Finally the third version, *Optimized vRPC*, is tuned for the VMMC interface. We also have a modified *Optimized vRPC* that avoids copying at the cost of slightly deviating from the *SunRPC* standard.

5.7 Measurements

In this section we analyze the performance of *SunRPC* and the three versions of *vRPC*. For the measurements we used two SHRIMP nodes. Each node has a Pentium processor running at 60MHz, 32M of memory and an ethernet connection. The operating system is Linux 1.2.8 and the code is compiled with gcc 2.6.3 using the -O3 optimization level. The nodes have the standard Unix configuration in terms of processes, including NFS and yellow pages services.

The two nodes were the only ones active on the SHRIMP network during the experiments, so there was no contention for the network. For the TCP ver-

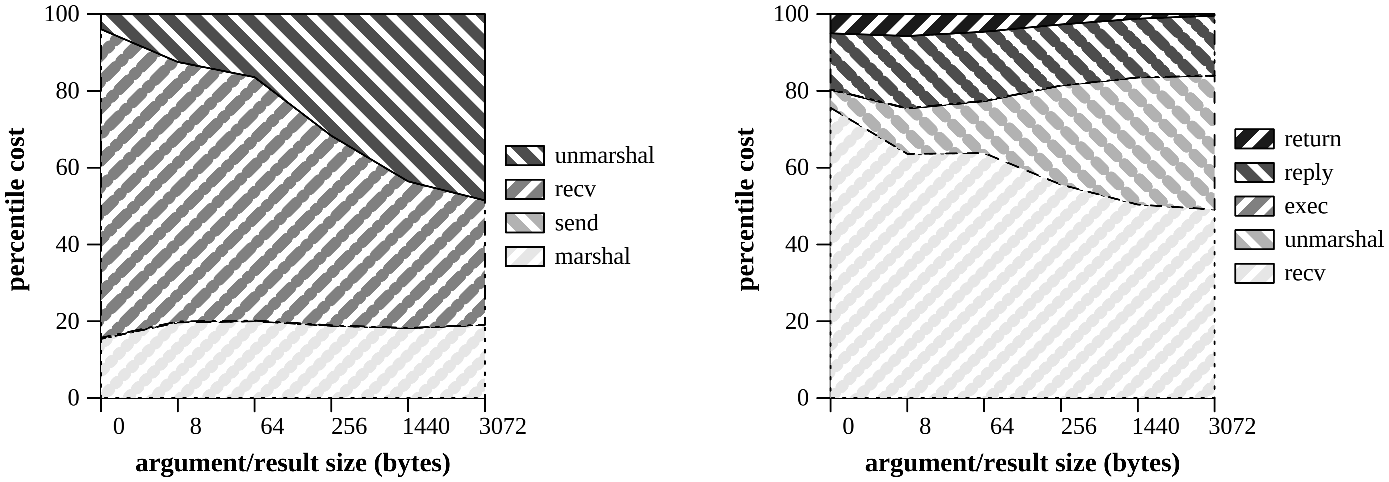

Figure 7: Distribution of the cost in the client (left) and the server (right) for *Optimized vRPC*.

sion, the ethernet segment connecting the two nodes was part of an isolated, private SHRIMP subnetwork.

The time measurements were done by reading, directly from user level, a hardware timer that is part of the CPU. The overhead of reading the timer was measured separately to be 0.5μs (29 cycles); this overhead has already been subtracted from our reported times.

Our experiments compare *SunRPC* and the three versions of *vRPC*. We divide the code into sections and present timing measurements for each section of the code, for null calls with seven argument/result sizes: zero bytes, 8 bytes, 64 bytes, 256 bytes, 1440 bytes and 3K bytes. We measured 1000 calls in each case, eliminated outlying points (due to experiments being interrupted by system daemon activity, etc.) and then averaged over the remaining runs. Full results appear in Tables 2-7 in Appendix A.

Figures 6 and 7 summarize these results. They present the time spent in the client and the server per section of a call, for the above argument/result sizes, as a percentage of the total cost of the call. The *send* and *exec* sections for the client and the server respectively, are almost negligible, as expected. *send* doesn't do much since the transfer is already initiated (automatic update) and the procedure executed in the server is a null procedure; it just sends back a reply of the same size as the arguments to the call.

5.8 Consequences for Performance Tuning

Removing the most expensive component of an execution path creates new bottlenecks; operations which previously were ignored because they were less expensive than the now-removed parts become the largest remaining bottlenecks. Achieving another major reduction in running time will require addressing these new bottlenecks. Sending the header of the call means sending each field separately to the destination using XDR functions. This implies loading each call with the buffer management overhead. We can avoid this cost by copying the header directly to a packet and sending it to the destination in a single message. Although buffer management is relatively simple for our protocol, it still imposes a noticeable cost as instruction counts go down.

The same applies to sending data. If the user defines the arguments to the call to be a contiguous block (opaque data, string etc.), there is only one call to send it across the network. But if the argument is, for instance, an array of integers, each element will be sent separately, incurring a higher overhead. A more sophisticated stub generator can generate much more efficient code for data transfers.

6 Instruction counts

We measured the cost of *vRPC* using a SHRIMP simulator running on a DECstation 5000, generating code using gcc version 2.5.7 with the default optimizations (no switches).

We measured the cost in terms of the number of instructions executed; we determined the instruction counts by running the programs under gdb. The execution path was divided into several parts, corresponding to the phases in a RPC. The results are depicted in Tables 9, 10, and 11 in Appendix B. The tables provide breakdowns for the client and the server for both the *SunRPC* (**TCP**) and *vRPC* (*Direct vRPC*, *Reduced vRPC*).

The measurements were done using a null RPC with varying sizes of arguments and return values. Three cases are presented: a call with 0-byte argument and return value, a call with 1440-byte argument and return value and a call with a 3K argument and return value.

We measured only user-level instructions. Since the SHRIMP code makes no system calls, this count is complete. For the TCP case, though, we did not count any instructions spent in the kernel servicing the read and write system calls.

The tables indicate where system calls are made (R: read, W: write). For the SHRIMP code "S" represents a call to the function that performs the send. This cost (which occurs at user level) varies with the generality of the function. For instance, allowing data misalignment to be handled would induce more checks in the code and thus increase the latency. Instruction counts include the cost of copying; components that include copying are labeled "(C)". The read and write kernel operations may also do one or more in-kernel copies, which we did not count.

Differences in instruction counts between different argument sizes are due to the extra cost of copying larger arguments. For TCP there are four copies per call, plus the copying that takes place in the kernel. *Direct vRPC* has six copies per send and receive, just as the TCP version does, since the goal is to isolate the performance gain from eliminating the system calls. The "extra" copy is actually performed at the first attempt to read the received data; this is the reason it appears at the **process header** section. In that sense the **Receive** section corresponds to receiving the data not into the user-provided buffer but into the export buffer, whereas for TCP it means to actually put the data in the user buffer.

Reduced vRPC has only two copies per call since we wanted the data to be placed in a buffer provided by the user. For *Reduced vRPC* it is also more difficult to measure the instructions at each layer because each call to send data actually sends it, so the first two sections had to be unified.

*Direct vRPC*has about the same instruction counts as the TCP version not including the system calls (given that there is one copy in the kernel), but there are no system calls in the execution path.

Reduced vRPC has considerably less overhead than the TCP implementation.

7 *ShrimpRPC*: A Specialized Implementation

While our *vRPC* library has very good performance, its implementation was limited by the need to remain compatible with the existing *SunRPC* standard. To explore the further performance gains that are possible, we implemented a non-compatible version of remote procedure call.

ShrimpRPC is not compatible with any existing RPC system, but it implements the full RPC functionality, with a stub generator that reads an interface definition file and generates code to marshal and unmarshal complex data types. The stub generator and runtime library were designed with SHRIMP in mind, so we believe they come close to the best possible RPC performance on the SHRIMP hardware.

Buffer Management The design of *ShrimpRPC* is similar to Bershad's URPC [1]; the main difference is that URPC runs on shared-memory machines while *ShrimpRPC* runs on the distributed-memory SHRIMP system. Each RPC binding consists of one receive buffer on each side (client and server) with bidirectional import-export mappings between them. When a call occurs, the client-side stub marshals the arguments into its buffer, and then transmits them into the server's buffer. At the end of the arguments is a flag which tells the server that the arguments are in place. The buffers are laid out so that the flag is in the same place for all calls that use the same binding, and so that the flag is immediately after the data. This allows both data and flag to be sent in a single data transfer.

When the server sees the flag, it calls the procedure that the client requested. At this point the arguments are still in the server's buffer. When the call is done, the server sends return values and a flag back to the sender.

Exploiting Automatic Update The structure of our *ShrimpRPC* works particularly well with automatic update. In this case, the client's buffer and the server's buffer are connected by a bidirectional automatic update binding; whenever one process writes its buffer, the written data is propagated automatically to the other process's buffer. The data layout and the structure of the client stub cause the client to fill memory locations consecutively while marshaling the arguments, so that all of the arguments and the flag can be combined into a single packet by the client-side hardware.

On the server side, return values (`OUT` and `INOUT` parameters) need no explicit marshaling. These variables are passed to the server-side procedure by reference: that is, by passing a pointer into the server's communication buffer. The result is that when the procedure writes any of its `OUT` or `INOUT` parameters, the written values are silently propagated back to the client by the automatic update mechanism. This communication is overlapped with the server's computation, so in many cases it appears to have no cost at all. When the server finishes, it simply writes a flag to tell the client that it is done.

The Stub Generator *ShrimpRPC* uses a specialized stub generator which is designed to make low-overhead stubs that cause marshaled data to be laid out as described above. Rather than describe the algorithm in laborious detail, we will give an example[2] that should illustrate how *ShrimpRPC* stubs work.

Assume that the *ShrimpRPC* stub generator is given an interface file `foo.x` with the following contents:

[2]In the example code, we omit some necessary checking for errors and other unusual conditions, in order to simplify the exposition. We have also shortened some of the identifiers; the real library uses long, obscure identifier names to avoid name collisions.

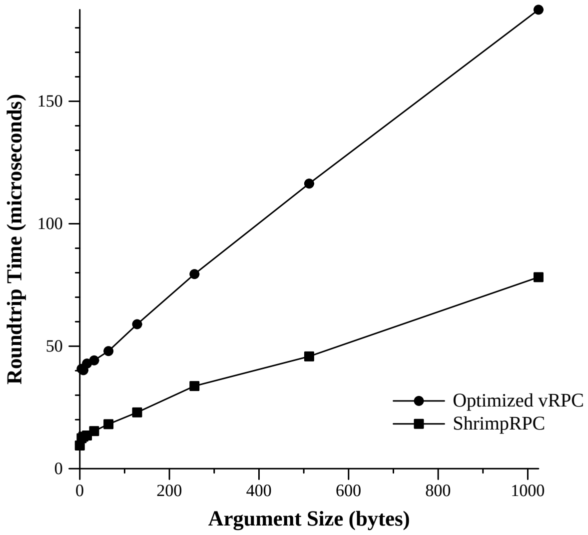

Figure 8: *Round-trip time for null RPC, with a single INOUT argument of varying size.*

```
STRUCT thingy { int a;  float b[17]; }
PROCEDURE add(IN int x, IN int y, OUT int sum)
PROCEDURE myproc(IN thingy q)
```

The stub generator creates an include-file that contains the data structure definition shown in Figure 9.

The client-side stub is shown in Figure 10. The stub first copies the arguments into the RPC buffer, and sets a flag in the RPC buffer. Since there is an automatic-update binding from the client's RPC buffer to the server's RPC buffer, this has the effect of sending the arguments and the flag value to the server. The client then simply spins until the flag value is changed back to zero. When this has occurred, the client knows that the server has sent back the return values, so the client extracts the return values from the RPC buffer and returns.

The server-side stub code is shown in Figure 11. The main server stub checks to see whether there is a call pending through the binding, and if there is one, it executes the call. The server detects a call by noticing that the flag in its RPC buffer is nonzero. The flag value tells the server which procedure is being invoked. The nonzero flag value tells the server that the client has already written the arguments into the RPC buffer. Rather than copying the arguments, the server simply passes them by reference into the remotely-invoked procedure. As the procedure runs, certain OUT and INOUT variables may be written; these written values are transparently propagated back to the client's RPC buffer by the automatic update mechanism. When the server is finished, it sets the flag in its RPC buffer to zero, thus informing the client that the call is finished.

As shown here, the main server stub uses polling to detect remote calls. The *ShrimpRPC* library switches from polling to blocking if a sufficient period of time has elapsed without any RPC requests arriving. This

```
struct thingy { int a; float b[17]; }

enum _foo_rpc_procedure {
  _foo_rpc_add_proc = 1,
  _foo_rpc_myproc_proc = 2;
}

struct _add_args {
  char _padding[60];
  int  sum;
  int  x;
  int  y;
  volatile int  flag;
};

struct _myproc_args {
  struct thingy q;
  volatile int  flag;
}

union {
  _add_args add_args;
  _myproc_args myproc_args;
} _foo_rpc_args
```

Figure 9: Data structure definitions emitted by the *ShrimpRPC* stub generator for the example interface.

blocking behavior is implemented in a *ShrimpRPC* library procedure that is not shown here.

The strategy described so far works whenever the size of RPC arguments is known to the stub generator. Variable-size arguments are handled using a separate bidirectional stream between the client and server. This is essentially the same stream mechanism used by *vRPC*. When some arguments have variable size, the constant size arguments are sent via the *ShrimpRPC* path described above; then the variable-size arguments are passed via the stream. Arguments of type `String` use a small fixed-size buffer, overflowing into the stream only if necessary.

Performance Figure 8 compares the performance of two versions of RPC: the *SunRPC*-compatible VRPC, and the non-compatible *ShrimpRPC*. We show the fastest version of each library, which uses automatic update in both cases.

The difference in round-trip time is more than a factor of three for small argument sizes: 9.5 μs for the non-compatible system, and 33 μs for the *SunRPC*-compatible system. The difference arises because the *SunRPC* standard requires a nontrivial header to be sent for every RPC, while the non-compatible *ShrimpRPC* system sends just the data plus a one-word flag, all of which can be combined by the hardware into a single packet. For large transfers, the difference is roughly a factor of two. This occurs because *ShrimpRPC* does not need to explicitly send the INOUT and OUT arguments from the server back to the client; these arguments are implicitly sent, in

```
/* data structure definitions */

/* code for RPC binding omitted */

SrpcStatus remote_add(
  Binding* binding,
  int* x, int* y, int* sum)
{
  struct _foo_rpc_args * buf = binding->buf;
  buf->add_args.x = *x;
  buf->add_args.y = *y;
  buf->add_args.flag = _foo_rpc_add_proc;
  while(buf->add_args.flag != 0){  /* spin */  }
  *sum = buf->add_args.sum;
  return SrpcOK;
}

/* code for remote_myproc omitted */
```

Figure 10: Client stub code for the example interface.

the background, via automatic update as the server writes them.

Since the round-trip latency of a *ShrimpRPC* call is within one microsecond of the hardware minimum, we do not provide a detailed breakdown of how *ShrimpRPC* spends its time.

8 Implementation Tradeoffs

Operation	Bandwidth (MB/s)
bcopy, 1LC to DRAM	36
bcopy, 2LC to DRAM	20
bcopy, DRAM to clean 2LC	36
bcopy, DRAM to dirty 2LC	29
various bcopys	NA
DMA from DRAM	23
DMA from 2LC	10

Table 1: Bandwidth of `bcopy` and DMA in MB/s. 1LC and 2LC stand for first and second level cache respectively.

In addition to the algorithmic choices described so far, the performance of our RPC libraries also depends on the details of how we use the SHRIMP hardware. Two main decisions were required: whether to use deliberate update or automatic update, and how to configure the caches of the PC nodes. The two decisions are actually bound together, since the performance of deliberate update and automatic update transfers depend on caching.

Recall that deliberate update and automatic update use very different mechanisms to send data. Deliberate update performs a DMA transfer from memory across

```
/* data structure definitions */

/* code for RPC binding omitted */

int _add_poll(Binding* binding) {
  struct _foo_rpc_args * buf = binding->buf;
  switch(buf->add_args.flag){
    case _foo_rpc_add_proc:
      add(&(buf->add_args.x),
          &(buf->add_args.y),
          &(buf->add_args.sum));
      return SrpcOK;
    case _foo_rpc_myproc_proc:
      myproc(&(buf->add_args.q));
      return SrpcOK;
    default:
      return SrpcNoCall;
  }
}
```

Figure 11: Server stub code for the example interface.

the I/O bus to the network interface board, while automatic update simply snoops values that are written on the memory bus.

Detailed micro-benchmarking of the SHRIMP system shows that, in most cases, deliberate update and automatic update achieve the same asymptotic bandwidth: about 23 MB/sec. The limiting factor in both cases is the rate at which packets can be transferred across the EISA I/O bus on the receiving side. However, there are some cases in which performance is worse.

To understand how this occurs, we must consider the cache architecture of the PCs. Each PC has an 8 kbyte, two-way associative, write-through first level cache, and a 256 kbyte, two-way associative second level cache. The second level cache uses write-through or write-back, depending on per-page state bits. Both caches use a line size of 32 bytes. The inclusion property is always maintained.

The hardware keeps the cache contents coherent with DMA transfers to and from memory. If DMA asks to read a memory location, and that location is dirty in the second-level cache, the cache writes the line back to memory before the DMA read can proceed. If DMA asks to write a memory location, and that location is in the second-level cache, then the line is written back to memory if dirty, and in any case is invalidated in the caches, before the DMA write may proceed.

Because deliberate update does DMA reads from memory, its performance suffers when the data is dirty in the cache. At the beginning of each cache line in the transfer, the DMA engine is stalled while the line is written back from the cache to memory. When this occurs, deliberate update bandwidth degrades to 10 MB/sec.

One way to avoid this problem is to use write-through caching for the affected pages; write-through

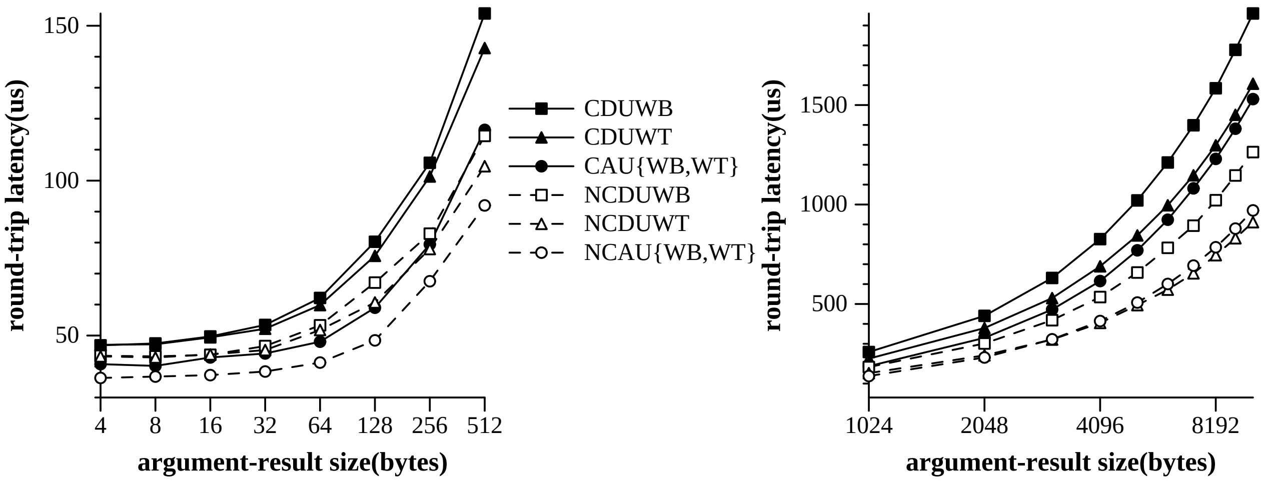

Figure 12: Latency of *vRPC* for small and large argument sizes

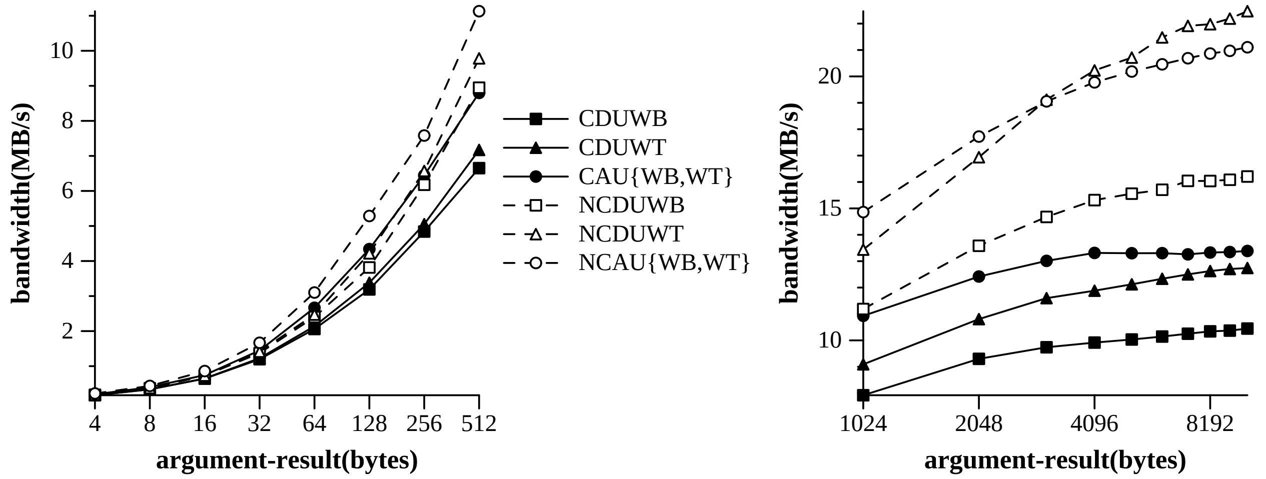

Figure 13: Bandwidth of *vRPC* for small and large argument sizes

pages cannot be dirty in the cache. Unfortunately, if the application also uses these pages for computation, this will lead to increased memory bus traffic, with lowers overall performance.

Automatic update also suffers slightly when the source data is not in the caches. Our libraries implement bulk automatic update by doing a memory-to-memory copy from the source data location to an RPC buffer. If the source data is only in memory, then the copying code will take repeated cache misses, lowering the bandwidth to 20 MB/sec. Fortunately, this case appears to be rare.

These factors lead us to favor automatic update as a bulk transfer mechanism for our RPC libraries on the current SHRIMP system.

On future machines, however, the conclusion may be different. The ratio of memory-bus speed to I/O bus speed may change. Future machines may not allow write-through caching, so automatic update pages will be forced to forgo caching altogether. Future systems will have more aggressive memory bus designs, so it will be harder to interface to the memory bus on a stock motherboard. Finally, different memory bus protocols may reduce or eliminate the DMA coherence penalty. We cannot predict which transfer mechanism will be better on future architectures.

Detailed Performance Tradeoffs. Table 1 shows the cost of deliberate and automatic update for different cache configurations. Figures 12 and 13 show the performance of *Optimized vRPC* with and without copy for diferrent cache configurations. One way to view the search space depicted in those Figures is by considering three different variables; *copying, mode of transfer and cache policy.* There are eight cases:

Copy(**C**)

Automatic update(**CAU**)

Write back(**CAUWB**)

Write through(**CAUWT**)

Deliberate update(**CDU**)

Write back(**CDUWB**)

Write through(**CDUWT**)

NoCopy(**NC**)

Automatic update(**NCAU**)

Write back(**NCAUWB**)

Write through(**NCAUWT**)

Deliberate update(**NCDU**)

Write back(**NCDUWB**)

Write through(**NCDUWT**)

9 Related Work

Our approach is similar in some ways to URPC [1], since both exploit user-level communication. URPC is built on top of a shared memory architecture while we use the distributed-memory SHRIMP architecture.

Bershad's LRPC [2] tries to optimize the kernel path for same-machine RPC calls. Since we have eliminated the kernel entirely, LRPC does not apply to our situation.

Thekkath and Levy [7] investigated the impact of recent improvements in network technology on communication software. They point out that both high throughput and low latency are required by modern distributed systems and that newer networks strive only for high throughput. They develop techniques to achieve low latency in communication software, using RPC as a case study. Their goal along with demonstrating the new techniques is to provide design guidelines for network controllers that will facilitate writing low latency communication software in traditional architectures.

Active messages [12] are a restricted form of RPC, in which the server-side procedure may not perform any actions that might block. Active messages achieve performance similar to *ShrimpRPC* on high-performance hardware, but without allowing general handlers to be invoked. The Optimistic Active Messages [22] approach allows an arbitrary handler to be invoked, using a fast-path implementation but switching to a slower path if the handler blocks. Neither of these systems provides full RPC services, such as automatic stub generation or binding between untrusting parties.

Several papers (e.g. [14, 17]) describe optimizations that dramatically improve the performance of RPC in traditional systems. This is generally done by avoiding copying, and reducing context switching overhead and network and RPC protocol overhead.

10 Discussion and conclusions

Network interfaces can have a great impact on communication performance and ease of programming. User level communication and virtual memory mapped network interfaces embody many of the optimizations done in other systems at extra work. Simply modifying an existing communication library can give results close to or better than the most highly optimized version in traditional interfaces. Copying can be limited to a minimum, and there are no interrupts on packet receipt and no kernel intervention.

The VMMC interface reduces the user to user latency. Using low latency mechanisms and providing support for user level communication leads to high performance communication software.

This effort shows that new architectures can open new horizons to distributed programming by providing high performance at low implementation cost.

Acknowledgments

We are grateful to Cezary Dubnicki for his expert advice on the SHRIMP simulator [11], and for his implementation of the SHRIMP system software. We would also like to thank Liviu Iftode for his implementation of an early mini version of the system software. We thank the members of the SHRIMP team for many useful discussions during the course of this work.

This project is sponsored in part by ARPA under contract N00014-91-1-1144, by NSF under grant MIP-9420653, by Digital Equipment Corporation, and by Intel Corporation. Felten is supported in part by an NSF National Young Investigator Award.

References

[1] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. User-level interprocess communication for shared memory multiprocessors. ACM Transactions on Computer Systems 9, 2 (May 1991), 175-198.

[2] B. Bershad, T. Anderson, E. Lazowska, H. Levy. Lightweight remote procedure call. ACM Transactions on Computer Systems 8, 1 (Feb. 1990), 37-55.

[3] A. Birrell, and B. Nelson. Implementing remote procedure calls. ACM Transactions on Computer Systems 2, 1 (Feb. 1984), 39-59.

[4] M.A. Blumrich, C. Dubnicki, E.W. Felten, and Kai Li. Protected, User-level DMA for the SHRIMP Network Interface. Proceedings of 2nd International Symposium on High-Performance Computer Architecture, February, 1996, pages 154–165.

[5] M.A. Blumrich, K. Li, R.D. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. Proceedings of 17th Annual Symposium on Computer Architecture, June 1990, pages 142–153.

[6] M.A. Blumrich, C. Dubnicki, E.W. Felten, K. Li, and M. Mesarina. Virtual memory mapped network interfaces. IEEE Micro 15(1):21–28, February 1995.

[7] C. A. Thekkath, and H.M. Levy. Limits to low-latency communication on high-speed networks. ACM Transactions on Computer Systems 11, 2 (May 1993), 179-203.

[8] D. Cheriton. The V kernel: A software base for distributed systems. IEEE Software 1(2):19–42, April 1984.

[9] C. Dubnicki, K. Li, and M. Mesarina. Network interface support for user-level buffer management. Workshop on Parallel Computer Routing and Communications, Springer-Verlag, 1994.

[10] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li. Software support for virtual memory-mapped communication. Proceedings of 10th International Parallel Processing Symposium, April 1996.

[11] C. Dubnicki. SHRIMP Basic Library and Its Simulator. Dept. of Computer Science, Princeton Univ., 1995.

[12] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser Active Messages: A Mechanism for Integrated Communication and Computation. Proceedings of 19th International Symposium on Computer Architecture, May 1992, pages 256–266.

[13] E.W. Felten, R.D. Alpert, A. Bilas, M.A. Blumrich, D.W. Clark, S.N. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. Proceedings of 23rd International Symposium on Computer Architecture, May 1996, pages 296–307.

[14] D. Johnson, and W. Zwaenepoel, The Peregrine high performance RPC system. Tech. Rep. COMP TR91-152, Dept. of Computer Science, Rice Univ., 1991.

[15] A. Karlin, K. Li, M. Menasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In Proceedings of 13th Symposium on Operating Systems Principles, Oct. 1991, pages 41–55.

[16] D. Ritchie and K. Thompson. The Unix time-sharing system. Communications of the ACM, 17(7):365–375, July 1974.

[17] M. Schroeder, and M. Burrows. Performance of Firefly RPC. ACM Transactions on Computer Systems 8(1):1–17, Feb. 1990.

[18] Sun Microsystems, Inc. XDR: external data representation standard. Internet Request For Comments RFC 1014, Internet Network Working Group, June 1987.

[19] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification Version 2. Internet Request For Comments RFC 1057, Internet Network Working Group, June 1988.

[20] C. Thacker, L. Stewart, and E. Satterthwaite, JR. Firefly: A multiprocessor workstation. IEEE Transactions on Computers, 37(8):909–920, Aug. 1988.

[21] R. Van Renesse, H. Van Staveren, and A. Tanenbaum. The performance of the Amoeba distributed operating system. Software Practice and Experience, 19(3):223–234 (Mar. 1989).

[22] D.A. Wallach, W.C. Hsieh, K. Johnson, M..F. Kaashoek, and W.E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. Proceedings of 5th ACM Symposium on Principles and Practice of Parallel Programming, July 1995.

[23] A. Wolman, G. Voelker, and C.A. Thekkath. Latency Analysis of TCP on an ATM Network. Technical Report 93-03-03, Dept. of Computer Science, Univ. of Washington.

11 Appendix A: Detailed *vRPC* Performance Data

Below are the analytic data for the measurements we made. The cost mentioned in section *Call and marshal* (*Reply*) is the time to send the argument (result) to the peer process. Depending on the mode of transfer used, this section might also include the transfer initiation for *vRPC*. Thus it is not easy to separate the cost of the transfer from the software overhead. Section *Send* accounts for the data transfer in the cases where there is an explicit send operation. Section *Receive and process header* is the time spent waiting for the peer process to reply and to process the header of the reply. This cost includes also the time spent processing in the peer process. Section *Unmarshal and return* shows the time spent in unmarshaling the result. Finally sections *Execute null procedure* and *Return from upcall* in the server show the time to execute the remote, null, procedure in the server and the time taken to return from the previous call and get ready for the next. The server always services all the outstanding requests from all the clients before going to sleep.

As mentioned above it is not always easy to separate the cost of the data transfers from the software overhead because the VMMC model overlaps communication with computation. For the 0-byte argument/result case, for instance, this cost is about 20 μs for the transfer of 2 headers (6 words each) and 2 words of control information. The rest is software overhead.

Client	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Call and marshal	12.0	7.3	10.6	5.1
Send	140.3	16.2	5.7	0.1
Receive and process header	1351.9	71.9	32.1	26.6
Unmarshal and return	4.9	2.0	1.6	1.3
Client Total	1509.1	97.5	50.0	33.0
Server	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Receive and process header	88.9	68.5	60.4	75.6
Upcall and unmarshal	0.5	4.1	3.2	4.7
Execute null procedure	0.0	0.0	0.1	0.1
Reply	10.2	25.3	33.0	14.6
Return from upcall	0.4	2.1	3.3	5.0
Server Total	100.0	100.0	100.0	100.0

Table 2: Time analysis of a 0-byte argument/result null RPC.

Client	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Call and marshal	12.5	10.0	17.4	7.9
Send	137.6	17.1	5.5	0.1
Receive and process header	1353.9	76.8	42.0	27.1
Unmarshal and return	8.9	4.9	3.9	5.0
Client Total	1512.8	108.9	68.9	40.0
Server	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Receive and process header	88.4	65.6	56.9	63.6
Upcall and unmarshal	0.9	5.7	5.7	11.8
Execute null procedure	0.0	0.1	0.0	0.1
Reply	10.3	25.8	33.6	18.8
Return from upcall	0.4	2.7	3.8	5.7
Server Total	100.0	100.0	100.0	100.0

Table 3: Time analysis of a 8-byte argument/result null *vRPC*.

Client	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Call and marshal	12.7	10.5	20.2	9.5
Send	154.7	20.9	5.7	0.1
Receive and process header	1492.3	87.7	47.2	30.1
Unmarshal and return	10.0	5.8	5.7	7.8
Client Total	1669.8	125.0	78.8	47.4
Server	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Receive and process header	88.4	66.9	56.1	63.8
Upcall and unmarshal	0.9	5.0	7.2	13.5
Execute null procedure	0.0	0.1	0.0	0.1
Reply	10.4	25.8	33.4	18.0
Return from upcall	0.3	2.2	3.2	4.6
Server Total	100.0	100.0	100.0	100.0

Table 4: Time analysis of a 64-byte argument/result null RPC.

Client	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Call and marshal	16.6	14.0	35.2	14.8
Send	208.9	40.2	5.6	0.1
Receive and process header	1979.4	116.7	69.9	38.8
Unmarshal and return	17.6	9.7	13.1	24.9
Client Total	2222.5	180.7	123.8	78.6
Server	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Receive and process header	88.8	66.1	57.3	55.6
Upcall and unmarshal	0.8	4.7	10.3	25.8
Execute null procedure	0.0	0.1	0.0	0.0
Reply	10.2	27.6	30.5	15.9
Return from upcall	0.2	1.6	1.9	2.7
Server Total	100.0	100.0	100.0	100.0

Table 5: Time analysis of a 256-byte argument/result null RPC.

Client	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Call and marshal	26.3	27.6	101.8	34.0
Send	425.6	111.6	5.5	0.1
Receive and process header	4049.7	210.5	141.2	71.2
Unmarshal and return	26.5	38.0	55.0	81.3
Client Total	4528.1	387.7	303.5	186.6
Server	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Receive and process header	87.8	61.8	53.2	50.4
Upcall and unmarshal	0.8	6.4	15.3	33.1
Execute null procedure	0.1	0.0	0.0	0.0
Reply	11.2	31.1	30.8	15.3
Return from upcall	0.1	0.7	0.7	1.2
Server Total	100.0	100.0	100.0	100.0

Table 6: Time analysis of a 1440-byte argument/result null RPC.

Client	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Call and marshal	63.9	68.7	286.0	90.3
Send	1144.8	303.5	5.8	0.1
Receive and process header	6728.4	473.6	221.3	153.0
Unmarshal and return	2158.6	108.8	266.5	229.3
Client Total	10095.7	954.6	779.6	472.8
Server	TCP	*Direct vRPC*	*Reduced vRPC*	*Optimized vRPC*
Receive and process header	66.3	51.3	43.5	49.1
Upcall and unmarshal	21.4	14.4	23.0	34.9
Execute null procedure	0.0	0.0	0.0	0.0
Reply	12.2	34.0	33.2	15.6
Return from upcall	0.1	0.3	0.3	0.4
Server Total	100.0	100.0	100.0	100.0

Table 7: Time analysis of a 3K-byte argument/result null RPC.

Optimized vRPC without copy			
Client	1440 bytes	3K	6K
Call and marshal	44.0	87.8	169.7
Send	0.1	0.1	0.1
Receive and process header	48.5	70.5	112.8
Unmarshal and return	78.8	162.4	317.4
Client Total	171.4	320.9	599.9
Server	1440 bytes	3K	6K
Receive and process header	16.5	9.8	5.1
Upcall and unmarshal	36.2	39.6	41.9
Execute null procedure	0.0	0.0	0.0
Reply	46.5	50.2	52.8
Return from upcall	0.8	0.4	0.3
Server Total	100.0	100.0	100.0

Table 8: Time analysis of *Optimized vRPC* without any copy. Automatic update is used for all the transfers. Numbers are in microseconds.

12 Appendix B: Detailed *vRPC* Instruction Counts

Client	TCP	*Direct vRPC*	*Reduced vRPC*
Call and marshal	342(C)	342(C)	0
Send	114+W	141+2S	237+2S
Receive	10+R	0	0
Process header	1534	1110(C)	248
Unmarshal and return	73(C)	73(C)	55(C)
Client Total	2073+R+W	1630+2S	540+2S
Server			
Receive and process header	1905+R	1467(C)	220
Upcall and unmarshal	111(C)	168(C)	111
Execute null procedure	10	10	10
Reply	843+W(C)	869+2S(C)	260+2S(C)
Return from upcall	97	97	104
Server Total	2966+R+W	2611+2S	705+2S
Client+Server Total	5039+2R+2W	4241+4S	1245+4S

Table 9: Instruction counts for a null RPC.

Client	TCP	*Direct vRPC*	*Reduced vRPC*
Call and marshal	1267(C)	1261(C)	0
Send	114+W	140+2S	353+3S
Receive	10+R	0	0
Process header	1534	1922(C)	248
Unmarshal and return	1026(C)	1026(C)	1001(C)
Client Total	3951+R+W	4349+2S	1602+3S
Server			
Receive and process header	1905+R	2277(C)	220
Upcall and unmarshal	1467(C)	1524(C)	1460(C)
Execute null procedure	11	11	11
Reply	1767+W(C)	1787+2S(C)	375+3S
Return from upcall	132	135	139
Server Total	5282+R+W	5734+2S	2205+3S
Client+Server Total	9233+2R+2W	10083+4S	3807+6S

Table 10: Instruction counts for a 1440–byte argument and 1440–byte result RPC.

Client	TCP	*Direct vRPC*	*Reduced vRPC*
Call and marshal	2185(C)	2179(C)	0
Send	114+W	141+2S	353+3S
Receive	10+R	0	0
Process header	1534	2840(C)	248
Unmarshal and return	1944(C)	1944(C)	1919(C)
Client Total	5787+R+W	7122+2S	2520+3S
Server			
Receive and process header	1905+R	3195(C)	220
Upcall and unmarshal	2844(C)	2901(C)	2837(C)
Execute null procedure	11	11	11
Reply	2685+W(C)	2705+2S(C)	375+3S
Return from upcall	132	135	139
Server Total	7577+R+W	8947+2S	3582+3S
Client+Server Total	13364+2R+2W	16069+4S	6102+6S

Table 11: Instruction counts for a 3K–byte argument and 3K–byte result RPC.