Split-C for the New Millennium

Andrew Begel, Philip Buonadonna, David Gay
University of California, Berkeley
{abegel, philipb, dgay}@cs.berkeley.edu

Abstract

We present an implementation of Active Messages and four implementations of the Split-C parallel programming language over the Virtual Interface Architecture user-level networking system running on the Berkeley Millennium cluster. This cluster is composed of 16 2-way SMP Intel 400 MHz PII processors using Myrinet network interface cards. Results from application benchmarks show that the best Split-C implementation is the one with the lowest processor send and receive overheads.

1 Introduction

Split-C is a parallel extension of the C programming language that supports efficient access to a global address space on distributed memory multiprocessors [Dus93]. Programs may access data on other processes via global pointers (essentially a normal C pointer combined with a processor ID). One of Split-C's more unique features is the support for explicit split-phase communication through these global pointers. Split-C supports a split-phase read (called get) and two split-phase write (put and store) operations that allow the programmer to explicitly overlap communication with computation. Completion of communications operations is ensured by a pair of synchronization primitives (one for get and put, another for store). In addition to memory transfer operations, Split-C supports explicit barriers, reductions and scans.

The Virtual Interface Architecture (VIA) defines a set of user-level networking mechanisms designed to minimize communication overhead. The VIA specification [VIA97] was developed by industry leaders as a proposed standard for high performance communication in distributed systems. It combines the principles of Active Messages and many other user level networks [Pak97, Eic95, Pry98, Dub96, Gil96, Dru94] as well as traditional network architectures into a combined software/hardware design.

Split-C has been implemented on many MPPs, as well as the Berkeley NOW (a cluster of 100 UltraSPARC 1s). This paper describes several new implementations of the Split-C communications layer targeted for the Berkeley Millennium cluster (16 2-way Intel PII Xeon SMPs running Linux (kernel version 2.2.1)). The low-level communications architecture on these machines is VIA.

Split-C on the NOW is based on Active Messages. Our first effort was to implement, on top of VIA, a version of Active Messages. This enabled us to easily port a first version of Split-C. We then created several other versions of Split-C's communications layer targeting VIA directly. The VIA specification defines essentially two levels of reliability for message delivery: unreliable (at-most-once) and reliable (exactly-once, in-order). The first Split-C implementation targets reliable VIA, reducing the communications overhead by specializing the communications layer for Split-C. The second targets unreliable VIA and adds mechanisms to the first that ensure reliable message delivery.

Due to limitations in our VIA implementation, we could not support network-based communications between processes running on the same host. A prior effort to support same-host communication [Lum98] developed Multi-Protocol Active Messages – remote communication goes over the network, while communication between processes on the same host (*local processes)* goes through queues in shared memory. While this implementation is effective, it places the same overhead on local communication as remote (same overhead, lower latency). We have developed a more efficient form of local communication by placing the local communication directly in the Split-C communications layer.

In this paper, we describe the implementation of Split-C over these four communication substrates and compare their performance to Split-C on NOW. Our results show that Split-C over unreliable VIA exhibits the best performance of our four implementations. We attribute this to its reduced processor overhead obtained by specializing the communications layer to Split-C. This reduction of overhead was a driving goal of our implementations, and its success confirms the results in [Mar97] which showed that application performance is quite sensitive to overhead but tolerant of latency.

In the remainder of this paper, we describe the structure, implementation and performance characteristics of the various communications architectures that underlie Split-C. In the next section, we explain the concepts behind VIA. In Section 3, we review the Active Messages architecture in preparation for the discussion of the implementation of Active Messages over VIA in Section 4. Section 5 covers our four implementations of Split-C communications layers. The next section presents performance measurements of Active Messages over VIA and our various Split-C implementations. Section 7 discusses some lessons learned during this project. We conclude with future work and present our conclusions.

2 VIA

This section describes the structure of and operation of VIA. The core component of the VIA system, the Virtual Interface, is the primary abstraction for a user's protected, direct channel to the network interface controller (NIC). Communication is achieved through bulk memory-to-memory transfers between a pair of virtual interfaces (VIs). To understand the construction of a VI, we define key terms used in the architecture:

• *Registered Memory* -- A portion of a user's virtual address space that has been pinned into physical memory and made known to a VI NIC. Registered memory functions as the principle communications buffer for network operations. A unique name or Memory Handle is associated with each region and used in conjunction with a user virtual address to access a buffer.

• *Descriptor* -- A data object recognized by the VI NIC that describes a network transfer request to be performed. Descriptors reside in registered memory.

• *Work Queue* -- A FIFO list of Descriptors to be processed by a VI NIC.

• *Doorbell* -- A mechanism for a user process to notify the VI NIC that outstanding descriptors have been posted to an associated work queue. Each doorbell is a protected resource, typically mapped into a user's address space, which is unique to a particular VI/user pair.

As shown in Figure 2-1, a VI consists of send and receive work queues, their associated doorbell resources and the user's registered memory regions. Prior to conducting communication, a connection is established with one, and only one, VI on a remote node. To initiate a network data transfer, the user process builds a descriptor and inserts it into appropriate work queue by placing a token in the queue's associated doorbell. Send descriptors are processed as soon as possible by the VI NIC, while receive descriptors are processed upon arrival of an incoming message. Once the data transfer is completed, the NIC marks the descriptor status as done. The host receives notification of this completion either by polling the descriptor or through an interrupt.

Another key component of VIA is the Completion Queue. The VI Completion Queue allows related VIs to be grouped together and monitored as a single object. At VI creation, each work queue may be optionally associated with a completion queue. When a VI descriptor completes, the VI NIC places an entry into the completion queue that indicates which VI completed an operation. The user process receives this notification through the same mechanisms as for descriptors -- either through polling or via interrupts. A feature of this component is that the send/receive queues of a VI may be associated with the same of different completion queues. For example, one completion queue could be used for all receives while the other is used for sends.

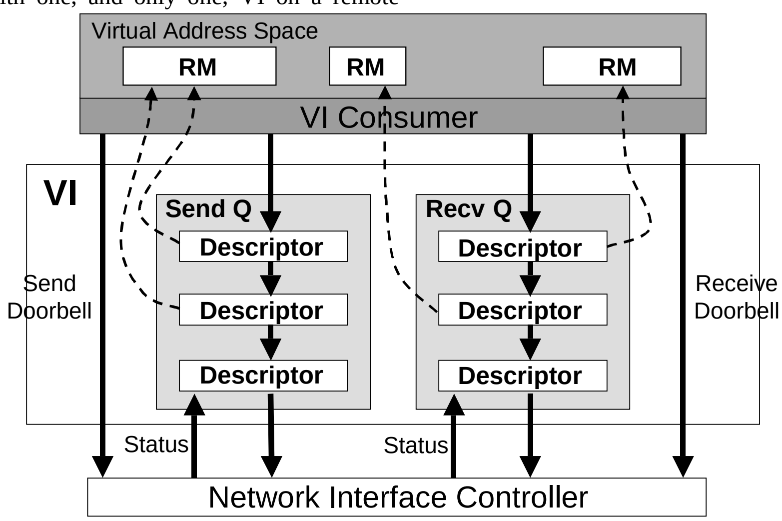

Figure 2-1: An individual Virtual Interface. Both descriptors and data reside in the registered memory regions (denoted by RM).

3 Active Messages

Active Messages (AM) is a simple, extensible paradigm for message-based communication in parallel and distributed computing systems [Eic92, Cul95]. At its core is the concept of integrating communication and computation in a way that minimizes the impact of communication overhead on overall performance. The Active Message mechanism may be viewed as essentially a specialized remote procedure call. Each message contains the name of a user-level handler to invoke on a target node and a data payload to pass in as arguments. The handler function serves the dual purpose of extracting the message from the network and either integrating the data into the computation or sending a response message. A key point under Active Messages is that the network is modeled as a pipeline with minimal buffering for messages. A process may issue a series of messages into the network and continue its computation while the messages propagate. This differs from other communication schemes that use blocking protocols or special send/receive buffers. To prevent network congestion and ensure adequate performance, message handlers must be able to execute quickly and asynchronously. As an additional requirement to prevent deadlock, a handler that generates a reply message must not be prevented from receiving incoming messages, regardless of the state of the outgoing channel. From a programmer's perspective, Active Message handlers are similar to interrupt service routines used in OS kernels and device drivers.

Active Messages on the NOW is implemented on top of a virtual network scheme which supports protected multi-programming communication [Main95]. The design architecture consists of two principal components: endpoints and bundles. The first component, the AM endpoint, is the fundamental abstraction for a process's connection to the network. A collection of endpoints among separate processes is connected to form a protected virtual network. Endpoints implement a two-phase request/reply [Mar94] Active Message scheme in which a request message is directly paired with a subsequent reply message. To provide flexibility for different applications, three different message sizes are supported: shorts (< 32 bytes), mediums (< 4 Kbytes) and bulk transfers (< network MTU). The internals of an endpoint include of a pair of buffer pools (send and receive), a virtual-memory segment, a translation table, a handler table and a protection tag. Endpoints also utilize a credit based flow-control scheme for requests to prevent network congestion and buffer overflow. To initiate a message transfer, a process calls *AM_Request()* or *AM_Reply()* to insert a message into an endpoint send pool for delivery to a remote receive pool. The message contains an integer handler index, a protection tag and the data payload. Upon receipt of a message, the message protection tag is compared against the endpoint protection tag. If they match, the handler index is used to reference the appropriate function in the handler table. For short messages, the arguments in the data payload are passed directly to the function. Medium messages include a pointer to a buffer containing data in addition to the regular arguments. Bulk transfers first copy the data payload to a sender-specified offset in the endpoint's virtual-memory segment and then invoke the handler with the specified arguments. To hide network addressing details, remote endpoints are referenced through an integer index into the translation table that contains the network address of all endpoints in the virtual network. Endpoint addresses are inserted into this table through separate calls to *AM_Map()*. A process can create several endpoints, each of which represents a connection to a separate virtual network.

An important aspect of this Active Messages implementation is that incoming messages are serviced through user-level polling of the endpoints. To simplify operations, a process's endpoints are gathered into disjoint subsets known as AM bundles. Polling of the endpoints in the bundle is done explicitly though a call to *AM_Poll()* and implicitly whenever a process calls *AM_Request()* or *AM_Reply()*. The bundle abstraction permits programmers to group together related endpoints and service them as a single unit. This simplifies programming tasks and permits a form of quality-of-service differentiation for groups of endpoints.

4 Active Messages over VIA

In this section, we present the internal details of the Active Messages over VIA (AMVIA) implementation. A core goal is to maintain the same API as AM in order to allow unmodified use by existing AM applications. As we will show, this goal influences many of the design decisions in AMVIA.

4.1 Components

To preserve API semantics and behavior, AMVIA's implementation retains much of the high level portions of the original AM code base. Low-level details such as operating system and network hardware calls have been replaced with VIA primitive functions. Facilitating the mapping from AM abstractions to VIA abstractions are two meta-structures: the VI Queue (VIQ) and the MAP object (Figure 4-1). The VIQ is essentially a logical channel for AM exchanges of a particular message size. Each VIQ contains a VI, pointers to unique send and receive buffers for descriptors and data, and a request credit counter. The request credit counter has a maximum credit value, *k*, which varies according to the message size assigned to the VIQ. The buffers are sized to support 2*k sends and 2*k + 1 receives (the need for the extra receive is discussed later). Three VIQs, one each for shorts, mediums and bulks, are allocated within a MAP object. The MAP object represents a logical point-to-point connection between endpoints in a virtual network. The MAP object also allocates a single registered memory segment from which the independent VIQ buffers are allocated. This minimizes the number of separate memory registration operations that must be performed. A collection of MAP objects in a user process forms an AM endpoint. Each MAP object in an endpoint is connected to a peer MAP object in every other endpoint on the virtual network. An advantage of the one-to-one connections is that it eliminates the need for protection tags. The connections provide the necessary security from errant messages or spoofing.

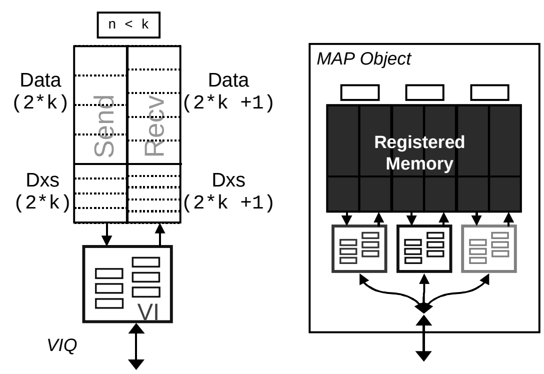

Figure 4-1: The VIQ (left) and the MAP Object (right) meta-structures of AMVIA.

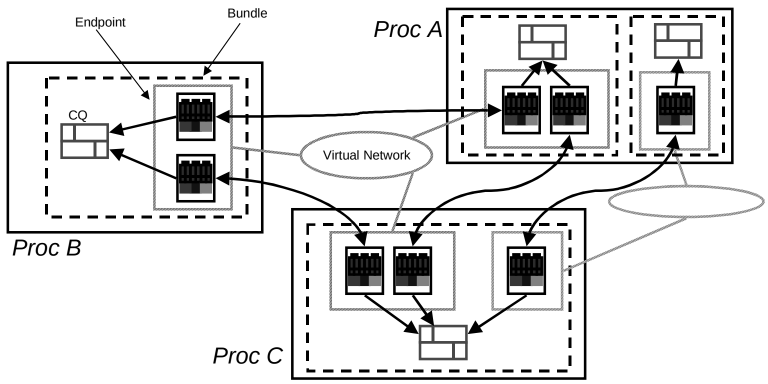

Figure 4-2: The AMVIA component integration architecture.

A side effect of the VIQ strategy is that it requires symmetry of message size between request/reply pairs. For example, if a medium request is sent, a medium reply must be used. Likewise, if a medium reply is expected, a medium request must be issued. The reason for this is a matter of buffering. If request/reply sizes were asymmetric, then each VIQ would have to have additional buffer space equivalent to the largest message size times the sum of allocated credits in the VIQ trio. This defeats one of the design functions of the VIQ to minimize required buffering. We further discuss this limitation later in this paper.

To implement a bundle of endpoints, the VI completion queue mechanism is used. When a bundle is allocated, two completion queues are created: one for monitoring sends and the other for receives. VIs are attached to these completion queues when they are created as part of a VIQ. The use of two completion queues permits assigning preferential service priority to receive operations. This minimizes network congestion and helps prevent deadlock. A problematic aspect of the VI specification with respect to completion queues is that there is no mechanism to bind a user-defined context with a particular VI. Thus, there is no direct method of determining the parent VIQ or MAP object of a VI that completes an operation. To solve this problem, a generalized binding table is employed which uses the VI handle value as a key to store an arbitrary pointer.

4.2 Operations

With the exception of events, AMVIA implements all of the API calls available in the native AM implementation. Prior to conducting communication, AM endpoints and bundles are allocated in their normal manner and the virtual network topology created. Upon calling *AM_Map()*, a new MAP object is created with sufficient registered memory space for the three VIQ structures. For each VIQ, a new VI is created and bound to it's VIQ and descriptors pre-posted to the receive queue. The VI is then connected with it's peer on the remote node. VIA uses a discriminator value to match together connections requests between two VI's. For AMVIA, the discriminator is generated by an ordered concatenation of the local and remote endpoint names. Connection establishment is synchronous and will block until both VI's are connected and ready to communicate. Once all three VIQ's have been created and VI connections established the *AM_Map()* function returns.

Sending operations in AMVIA are straightforward and are only slightly modified from the original AM implementation. For an *AM_Request()*, the function attempts to obtain a free send descriptor and a request credit. If both are not available, the function polls until it can proceed. The data payload is then copied into the appropriate message buffer and the send descriptor posted to the send queue. Control then returns to the calling thread without waiting for descriptor completion. *AM_Reply()* functions in an identical manner except that it does not wait for a request credit. Instead, the function checks to ensure the symmetry requirements discussed previously are followed. If not, the reply aborts.

The sequence of operations that take place in an AMVIA receive vary depending on the message size. Short messages are processed by directly invoking the designated handler with the data arguments (the copy here is implicit). For long messages, the data payload is transferred to the VM segment and then the handler function invoked. Processing of medium messages, however, does not involve a copy of the data payload. Instead, the handler is invoked with a direct pointer to the medium message. Thus, incoming medium messages are able to exploit the zero-copy semantics intended by the VI architecture. Once the handler returns, the associated receive descriptor is cleared and re-posted to the VI's receive queue. The fact that the receive descriptor is not recycled until *after* the handler completes requires the receive queue to contain one extra element. This is to ensure that a reply sent by a request handler does not create a new request from which there is no available buffer. Recycling the receive descriptor before invoking the handler would require extra data copies that would degrade performance.

The *AM_Poll()* operation serves a dual purpose in AMVIA. First, the routine checks the receive completion queue in the bundle for incoming messages. For each received message, the routine determines the message type, request or reply, and pushes a pointer to the message into a respective queue. This is repeated until all outstanding receive operations have been segregated into their appropriate queues. Then, depending on a binary argument to the routine, one entry from both the reply and request queues or just the reply queue is processed. This marshalling of incoming requests and replies is necessary for two reasons. First, it provides the means to disable processing of incoming requests that might result in deadlock and, second, it ensures that request handlers are executed atomically. In the native AM implementation, the NIC could distinguish between request/reply types and place them in separate queues, thus negating the need for the host to marshal the two types. The other purpose of the polling routine is to recycle send descriptors. The head of the send completion queue is checked once per call to *AM_Poll()* and the completed send descriptor marked available for reuse.

5 Implementing Split-C

The Split-C compiler is based on a modified version of the gcc C compiler that calls specific functions for each basic Split-C operation: read, write, get, put and store. These operations are implemented in a library (libsplit-c) that also provides the other Split-C functions (e.g., bulk_get, bulk_put, barriers, reductions, etc.). This library also deals with the startup and shutdown of the Split-C program. Thus, all platform-specific code is confined to libsplit-c.

We have implemented a number of different versions of Split-C for the Millennium cluster using the VIA networking abstraction. These four versions are all based on the AMII-based Split-C for the NOW:

1. Split-C over AMVIA: This is essentially unchanged from the NOW version, except for program startup and shutdown issues, which we will not address further.

2. Split-C over AMVIA plus Shared Memory: Off-host communication uses AMVIA, communication between local processes uses shared memory.

3. Split-C over Reliable VIA: This implementation assumes that the VIA layer provides reliable message delivery.

4. Split-C over Unreliable VIA: This implementation does not assume reliable message delivery.

We start with a short overview of the functionality provided in libsplit-c, and then present a description of the implementation of each of these versions of Split-C.

5.1 Overview of libsplit-c

The libsplit-c library implements a number of primitive operations:

- get, put, and store for the basic types (char, short, int, float, double, long long) and bulk objects

- completion detection for gets, puts, and stores (the sync and store_sync operations)

- barriers

All other operations (reductions, scans, reads, writes, etc) are implemented in terms of these primitive operations. There are no practical differences between get, put and store for the different basic types, therefore we will restrict our attention to get, put and store of integers.

5.2 Split-C over AMVIA

Upon startup, Split-C sets up an AM connection between every pair of processes. Split-C's primitive operations (get, put and store) are implemented in a straightforward manner. Each primitive operates on a global pointer and a local address. If the global pointer is local to this process, the operation is carried out using local memory. Otherwise, an active message request is sent to the process specified in the global pointer and the operation carried out remotely. The reply to this active message specifies a handler which alternately writes the result of a get, signals the completion of puts, and does nothing for stores. Bulk get, put and store operations use the AM Medium message format. Each bulk operation is broken into segments corresponding to the maximum medium message size. The implementation is otherwise identical to that used for integers. The sync and store_sync operations busy wait (while calling *AM_Poll()* to drain the network) until the appropriate condition is fulfilled. The barrier uses a two-phase binary tree communication pattern.

5.3 Split-C with Shared Memory

The Millennium cluster is made up of 16 2-way x86 SMPs. In order to use both processors on an SMP, Split-C's communications layer has to support local communication between processes. This may be accomplished at various levels in the communications hierarchy. For instance, we might have each process use a unique physical network endpoint, and use the network to form a loopback connection. As an optimization, we might modify the firmware in the NIC to recognize packets destined for the same host and reroute them directly into its receive buffer. Moving up the hierarchy, we might modify the VIA software layer to recognize a message to another VI on the same host and deliver it directly into the other process's receive buffer. The same optimization might also be performed in the Active Message layer, using shared memory to implement a communications buffer between multiple processes on the same host. This is the implementation choice used by Multi-Protocol Active Messages on a cluster of Solaris SMPs [Lum98].

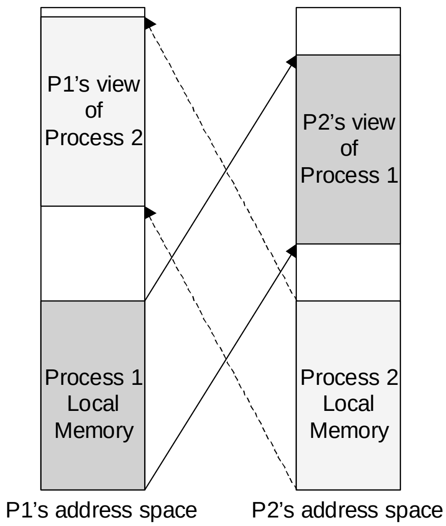

Figure 5-1: Address space mapping on shard-memory Split-C

In furtherance of our goal to remove unnecessary abstractions, we modified the Split-C communications layer to recognize when a message is destined for another process on the same host and use shared memory to copy the data directly into the other process's address space.

Split-C's global address space assumes a separate address space for each process running in a Split-C program. In the tradition of C, Split-C allows a programmer to create a global pointer that points to *any* part of memory in any process. Split-C also guarantees that some local addresses are the same in all processes, namely the addresses of global variables and the result of all_spread_malloc. The programmer can use this invariant to create a valid global pointer to another process's address space from knowledge of where a pointer is in its own address space.

To maintain compatibility with Split-C programs that exploit this invariant (essentially all Split-C programs) we should map the stack, data segment and heap of each process on a host into the address space of every other process on the same host (Figure 5-1). We decided to not support the stack because it is dynamically allocated and were it to be extended, every other process would have to be notified to map the new chunk of memory. We give each process a 256 MB *virtually allocated* heap which we map into the address space of every other process at program startup.

Mapping the data segment of each process was a little more difficult. We first tried to copy the data segment into the first part of the shared memory heap, for if the code were compiled to be position independent (`-fPIC`) then every access to memory in the data segment would be indirected through a register. At the start of main(), we could just modify the register to point to the copy of data segment residing in the shared memory heap. Unfortunately, we also found that the Split-CC compiler (a modification of gcc) inserts code to compute the PC-relative offset to the global offset table (GOT) in every function prologue. This meant we would have had to reset the GOT register at the start of every procedure (even those in library functions). We made an attempt to modify Split-CC to not insert the GOT calculation, but this broke gdb which relied on knowing the length of the function prologue. In addition, we would have had to rewrite the binary for each library that was not compiled with Split-CC.

Instead, we take advantage of Linux 2.2.1's new `/proc` file system to mmap the data segment. Each process has a virtual directory in `/proc` which contains a virtual file called "mem." This file points to the virtual memory of each process. In order to successfully attach to each process's data segment, we had to modify the Linux kernel to turn off conservative error checking clauses that prevented any process from mmapping another process if it was not its parent.

With these modifications in place, we modify the communications primitives in the Split-C communications layer. First, we add a clause to each read, write, get, put and store operation to identify global pointers to memory on the same host and perform a read or write directly in the shared memory region belonging to the process specified in the global pointer. We also rewrite the barrier operation to perform an array-based barrier among processes local to each host. Then, one process on each box participates in a two-phase tree-based barrier using active messages. Most of Split-C's other global operations are written using get, put and store, and require no changes.

Atomic operations, however, are another tricky area. Split-C takes advantage of the atomicity semantics of Active Message handlers to implement atomic remote procedures. Unfortunately, a process isn't able to run arbitrary user functions on behalf of another process on the same host even if it has access to its memory. Support for these atomic operations has not yet been completed, however, it will likely consist of an explicit producer/consumer queue containing functions for each process to execute.

5.4 Split-C over Reliable VIA

The implementation of Split-C assumes that reliable message delivery is provided by VIA. On a reliable VIA connection, reception of an unexpected message for which there is no descriptor posted, or for which the posted descriptor is not big enough causes the connection to be terminated [VIA97]. Therefore, it is essential to run a credit scheme to prevent this from occurring. As the existing implementation of AMVIA implements such a scheme (and also assumes a reliable VIA connection), we chose to implement this version of Split-C by merging libsplit-c with the AMVIA library. This opens up a number of opportunities for optimization of the send and receive overhead:

We remove support for the active message "bulk" format (Split-C bulk operations use the "Medium" format), thus reducing the number of VIs per connection to a remote process from three to two. Dynamic dispatch is not required on sends, nor on receives. Messages are still identified by a small unique integer (which we still call the "handler"), but the code for each handler is placed inside a switch statement in the poll routine. Finally, we reduce the

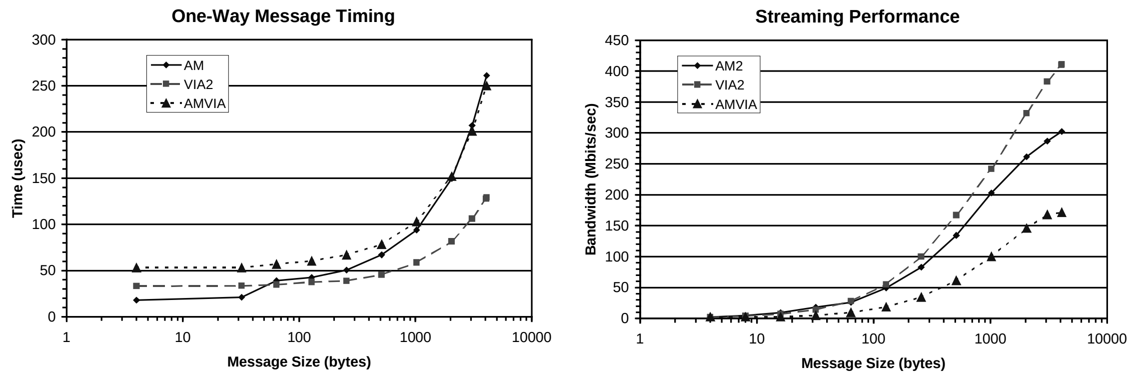

Figure 6-1: One-way messsage timing and streaming performance for AMVIA. The underlying VIA and the native AM implementation results are provided for comparison.

message size by removing unnecessary fields in the message format. The smallest possible message is reduced to 4 bytes from an original specified minimum size of 28 (in fact, the AMVIA implementation has a 64 byte minimum message size).

Split-C over reliable VIA also brings a small change to the AM request/reply discipline (and hence to the AMVIA request credit management scheme): some requests need not send replies. This is useful for the Split-C store operations which do not deliver any completion notification to the sender. Obviously, if store requests were never acknowledged, a processor sending such requests would soon run out of credits. To get around this problem, after a processor P_1 receives n requests from a processor P_2 for which it does not reply, it sends a special acknowledgement message to P_2 that restores n request credits to it. We chose n to be a quarter of the maximum number of credits. If n is chosen too small, there is little benefit for reply-free requests. If n is too large, the general credit scheme will become ineffective.

5.5 Split-C over Unreliable VIA

A large part of the Split-C over unreliable VIA implementation is identical to the reliable version. Split-C's primitive operations are still implemented assuming a reliable request/reply mechanism -- the only difference is in the implementation of the mechanism itself.

Our goal for this implementation is to provide reliability with very low processor overhead on both send and receive sides. We assume that message failures are extremely rare, so that it is not necessary to perform well in their presence. In the absence of losses, the credit scheme assuming unreliable VIA should behave the same as for reliable VIA. No messages will be lost because the target process was not ready to accept them. While we have implemented all the bookkeeping necessary to provide reliability, it is important to note that our current system does not implement detection and retransmission of lost messages.

We provide a reliable request/reply mechanism over each VIA connection by merging a conventional sliding window protocol with the credit scheme used by AMVIA. There are six important aspects to our implementation:

- We number requests and replies uniquely and separately, rather than number all messages in a single sequence.

- Each request or reply exchanged includes an acknowledgement of the received requests and replies that have been fully *processed* (see the discussion of reply-free requests below).

- Requests and replies are processed in the order they were sent; while this is not necessary to satisfy Split-C semantics, it is required as the sliding window protocol acknowledgements all previous message with a single acknowledgement.

- For each connection, we keep track of the number of request credits available. This information is derived from the request acknowledgements in the messages we receive.

- Reply-free requests do not normally need any extra acknowledgements as long as both sides are exchanging messages (a request acknowledgement takes the place of a reply, hence the need to acknowledge requests once they are processed, rather than acknowledging them as soon as they are received). A special "no-op" reply is sent in the case where n reply-free requests are received from a process without any message being sent back to it. We have arbitrarily set n to a quarter of the maximum number of request credits.

- Timeouts are detected by recording the time at which each message is sent. In the poll routine, the current time is compared with those of unacknowledged messages which cause them to be resent if a (fairly high) timeout has expired. However, we have not yet implemented this important part of our reliability scheme.

6 Performance

This section discusses the performance measurements of AMVIA and the various implementations of Split-C described above, as well as Split-C on the NOW. For AMVIA, we measure one way message time, streaming bulk performance and LogP micro-benchmarks. For Split-C, we measure raw communication primitive performance and benchmark five applications.

6.1 AMVIA

The performance of AMVIA is evaluated from three benchmarks: One-way message time, streaming performance and the LogP micro-benchmark [Cul93, Cul96]. One way message time is a measure of the average time it takes for a message of a given size to be transmitted from a source node and received by the destination node. It is measured through a series of ping-pong tests in which a message is sent to a destination node which reflects it back to the sender. The resulting round-trip time (RTT) is divided by two to yield the one-way time. The streaming benchmark measures the throughput when messages are sent successively from a source to a sink with no pause in between. The results of these two benchmarks are presented in Figure 6-1. Performance results for the native VI Architecture an AM implementations are included for comparison.

The AMVIA one-way message time of 53 µsec adds 15 µsec to the underlying VIA one-way time for the same message size. It is three times worse than the AM implementation on the NOW. The extra time is attributable to the complex polling routine and handler dispatching on the host. The polling routine must travel through multiple layers of indirection (Completion Queue to VI to Descriptor to Data) to retrieve the message and lookup the VI to VIQ binding. It must then marshal the message by request/type and then dispatch the handler. For messages larger than 32 bytes, additional overhead is incurred due to buffer copying. In AMVIA, this cost is amortized by other factors whereas it is pronounced in AM.

The factors that affect one-way time also extend to streaming performance. AMVIA achieves a throughput of 172 Mbits/sec for 4KB messages. This is less than half of either VIA or AM on the NOW. Some of the throughput limitations are due to the credit-based flow control scheme in Active Messages which enforces positive acknowledgement for every message sent. However, the majority of the impact comes again from the polling routine. The problem is exacerbated because outgoing messages may stall while incoming messages are being processed.

A better understanding of the low-level performance of VIA is achieved through a LogP analysis. The results of the LogP benchmarks for AMVIA and AM on the NOW are presented in Figure 6-2 and summarized in Table 6-1. For $\Delta < 15\mu sec$, the time per message increases rapidly to approximately 15 µsec. This time reflects the mean interval a process must wait for the VI NIC to service a particular doorbell register given a fixed number of VIs and network load. Above a burst size of 64 messages, the flow-control scheme takes effect and the time per message increases to the gap value of 50 µsec. For $\Delta \geq 15$ µsec, there is sufficient delay between messages to prevent waiting for the doorbell register and the initial time per message increase is caused by the servicing of incoming replies. It is tempting to label the 15 µsec plateau as send overhead (o_s), but this would result in a negative receive overhead (o_r). This is because the VI NIC continuously processes network traffic in between servicing doorbells. Thus, the waiting period actually overlaps with the transport latency (L). With our VI implementation, this overlap cannot be hidden by useful computation. An additional latency component is incurred by the polling routine. The request/reply marshalling and send descriptor recycling add an additional factor that is not directly part of a send or receive stream.

Parameter	Value (µsec)
Latency (L)	45
Send Overhead (o_s)	3
Receive Overhead (o_r)	5
Gap (g)	50

Table 6-1: AMVIA LogP microbenchmark summary

The performance of AMVIA is somewhat bipolar. On one side, the implementation is robust and works for all of the benchmarks and Split-C applications we tested. Yet, its raw performance parameters (i.e. bandwidth, RTT) are somewhat substandard. This paper has examined the major sources of these degradations in AMVIA. However, root causes of the poor performance can be

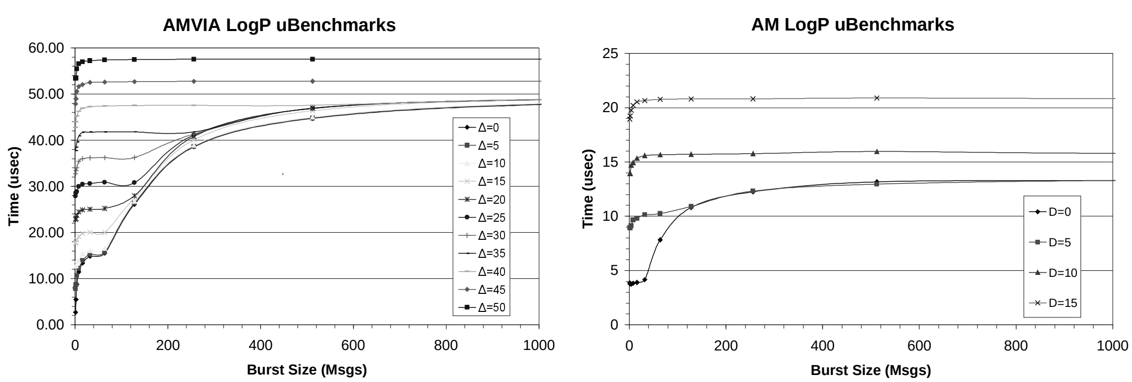

Figure 6-2: LogP Microbenchmark signatures for AMVIA and native AM.

traced back to VIA. For one, VIA is principally a bulk data transport and does not perform well for small messages [Buo98]. In contrast, in AM small message performance is paramount. Another lesson of AMVIA's performance is that both VIA and AM benefit from direct hardware support of their architectures. For VIA, this principally includes doorbell management [Buo98] and virtual address translation support [Buo99]. AM benefits from support for small messages and hardware separation of requests and replies. Although not absolute, the benefits of hardware assist make porting AM over a generalized transport such as VIA more difficult.

6.2 Split-C Microbenchmarks

To test the performance of the Split-C primitive operations – read, write, get, put and store – on basic types, we use a simple microbenchmark that repeats each operation 10,000 times and reports the average time per iteration. We run these benchmarks twice, once with one process sending to an idle one (one-way), and another where both processes are simultaneously sending to each other (two-way). The results are presented in Figure 6-3. These results include timings on Split-C over AM on the NOW for reference. The times for Split-C over shared memory are so small (less than 400 ns) that they are practically invisible. We see that the NOW is much faster than all of the VIA-based implementations, which is consistent with the results above. We also see that the stores are twice as fast as get and put operations in the Split-C over both reliable and unreliable VIA due to the reply-free requests. The two-way message benchmarks shows that reads and writes take essentially the same about on time as with one-way messages, but the gets, puts and stores are twice as slow. Reads

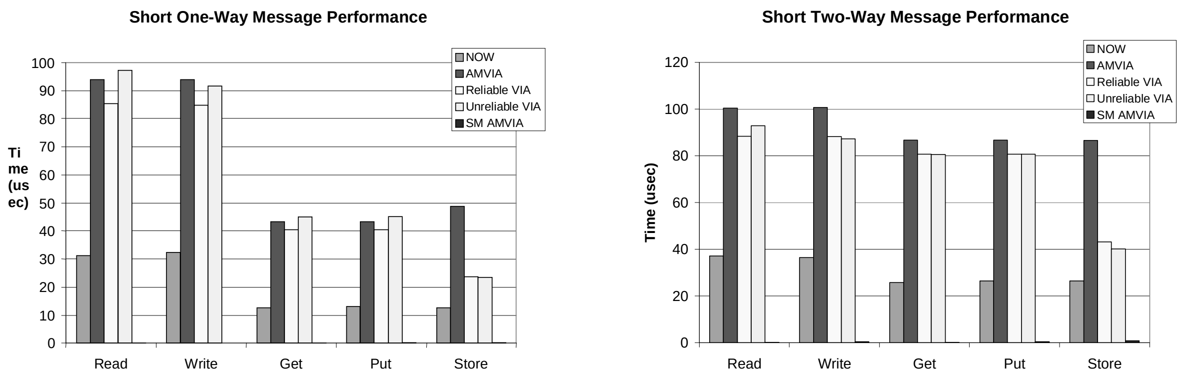

Figure 6-3: Split-C microbenchmarks for primitive operations on integers (smaller is better).

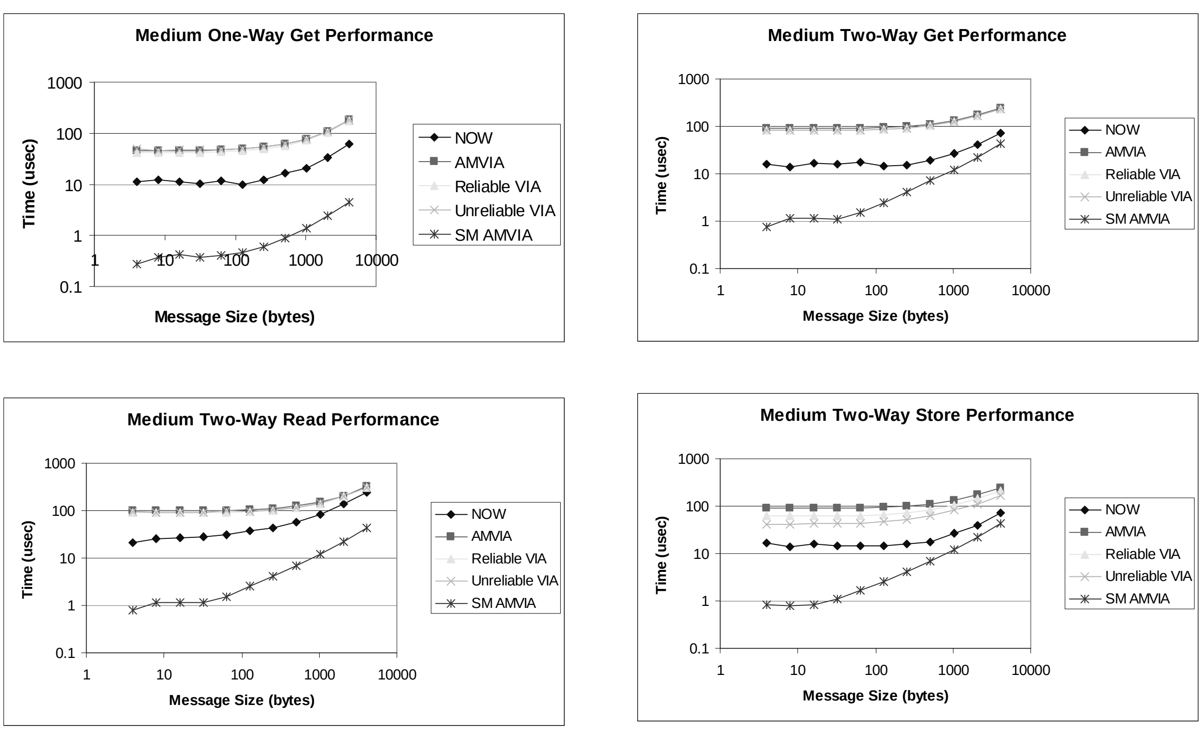

Figure 6-4: Performance of selected Split-C bulk operations (smaller is better).

and writes do not slow down because they can process incoming requests in the time they are waiting for their replies. Gets, puts and stores slow down because the processor and the NIC must process incoming messages when they would otherwise be sending the next request.

Next, we measure the performance of the Split-C bulk operations with message sizes from 4 bytes to 4 kilobytes in the one-way and two-way cases. In Figure 6-4, we present representative results for one-way bulk gets, two-way bulk reads, gets, and stores. For small messages, two-way gets are twice as slow as one-way gets for the same reasons stated above. For large messages, the performance difference between the two is smaller because message processing overlaps with message transfer. One interesting feature to notice is the slope of the shared memory performance between one-way and two-way messages. This is due to true-sharing cache misses in our simple-minded benchmark.

6.3 Split-C Applications

We benchmark five applications (Figure 6-5):

- 3D FFT: we compute a 128x128x128 3D Fast Fourier Transform.
- Conjugate gradient solver: solving on a 4096x4096 sparse matrix.
- Cholesky decomposition: Cholesky decomposition of a 1536x1536 matrix (decomposed in (12x8) x (8x12) x (16x16) blocks).
- EM3D: An electro-magnetic particle simulation in three dimensions. This benchmark uses scaled scaling with 500 particles per node with an average of 30% remote nodes.
- pico-Ray parallel ray tracer [Mar95]: ray trace a teapot.We show their performance on the five Split-C implementations on 1 through 16 processors. The shared memory implementation is still incomplete and only has results for 3D FFT, and 1 and 2 processors on Raytrace and Cholesky.

On 3D FFT, we see that the NOW fails to scale past 8 processors. This may be due to a lack of bisection bandwidth needed for the all-to-all communication pattern of the parallel transpose step of 3D FFT. The shared memory version is slower than the non-shared memory versions because two processors on a host are competing for access to the single network card.

Conjugate gradient scales very poorly. It uses stores extensively, hence the improved performance of Split-C over reliable or unreliable VIA as compared to Split-C over AMVIA. We do not know the reason for the terrible performance of Split-C on the NOW for more than 2 nodes.

The raytracer shows the best speedup of all the applications. The results include an oddity: the raytracer compiled with Split-C over AMVIA is 54% faster on one processor than the raytracer compiled with Split-C over reliable or unreliable VIA. This performance anomaly disappears from 2 processors onwards.

The scaling of EM3D is poor (and abysmal when compared to the single processor time which we did not include in the graph). As with conjugate gradient, we see an advantage to Split-C over reliable or unreliable VIA due to EM3D's use of store operations.

On Cholesky, the scaling is reasonable past two processors. This is also the only application where the NOW performs better than the Millennium for large numbers of processors. Shared memory Split-C does well, scaling linearly with two processes.

Looking at the applications as a whole, we see that Split-C over unreliable VIA has the best overall performance, slightly ahead of Split-C over reliable VIA.[1] Both of these implementations of Split-C have reduced processor overhead on both send and receive. This reduced overhead comes from the reply-free requests used for stores and from specialization of the request/reply mechanism for Split-C. This confirm results presented in [Mar99] which show that application performance is most sensitive to processor overhead and tolerant of latency. Thus, our simple microbenchmarks showing sustained message rate and round-trip times are not directly indicative of application performance.

7 Discussion

7.1 Design Tradeoffs and Evaluation.

The development and analysis of AMVIA and its integration into Split-C yield several insights into both AMVIA and VIA. In this section we present an evaluation of the design tradeoffs in AMVIA and how these are impacted by subtle differences in VIA and AM.

7.2 Logical Channels

The use of a single VI and independent buffers as a logical channel for a particular AM message provides an interesting balance of resource scaling and reliability. By using a separate VI and queue for short, medium and bulk messages a finer grained credit scheme can be used that reduces the amount of buffering necessary. The amount of buffering required for a VIQ in the AMVIA implementation is given by the equation:

$[4 * k + 1] * M$
(k = Credit Allotment, M = Message Size)

[1] It is not clear why unreliable VIA should be faster than reliable VIA, as the reliability mechanism should only add overhead and the reliable VIA implementation is actually using an unreliable VI.

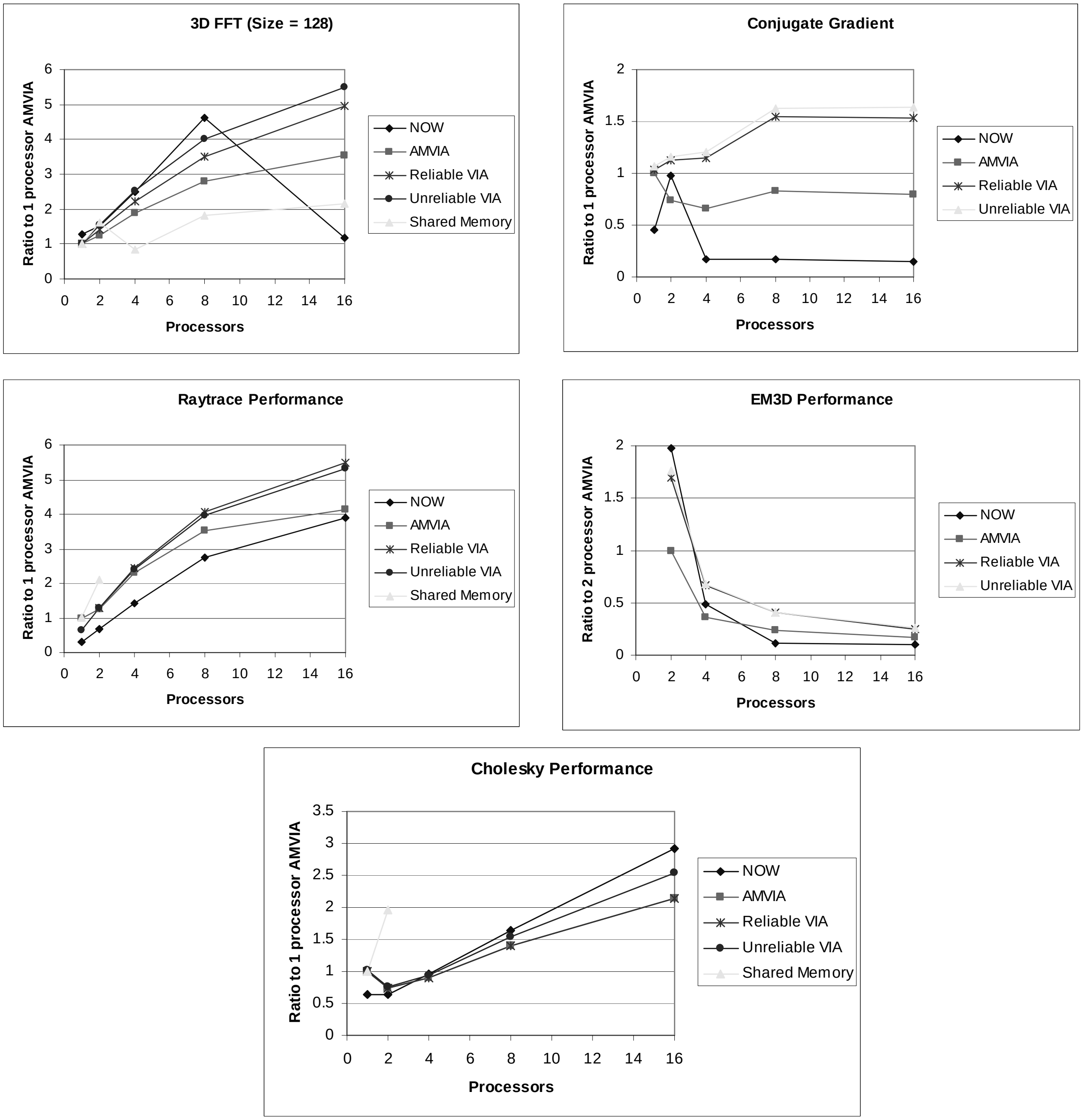

Figure 6-5: Split-C application performance (bigger is better)

If a single VI were used for all messages sizes, assuming a credit allotment of 32 and a network MTU of 80KB, the buffering requirement exceeds 10 MBytes. In the three VI scheme, different credit allotments can be assigned to each message size. For AMVIA, performance tests suggest that a credit allotment of 64, 8 and 1 for shorts, mediums and longs respectively is adequate for many applications. The resulting buffering requirement for this allotment is approximately 540KB. One drawback of this design is that it requires the programmer provide advance knowledge of an expected reply message size for a given request. If the implementation were more general, then the communication layer would necessarily be required to have sufficient buffering to handle any size of reply. This defeats the original purpose of the design. Presently the AM API does not provide the means to notify the communication layer of the expected reply size. Thus, AMVIA assumes and enforces symmetric request/reply message sizes. Another drawback is the need to marshal request/reply messages on the host. The alternative to this is to implement separate channels for requests and replies by creating separate VIs for each. This would double the number of VI resources needed by the an

AM virtual network. We believe the 3 VI design represents the best balance of performance and resource scalability possible.

7.3 Data Management

The data management schemes in AMVIA are constrained by the AM API and the development time that was available. For send operations, copying data into the communication buffer permitted us to quickly and robustly implement AMVIA without having to create sophisticated memory management schemes. While this defeats the mechanisms of the VI Architecture designed for zero-copy sends, it is required for short and medium messages since AM semantics allow modification of source data once the sending function returns. Blocking until the send operation completes would negatively impact performance. Receive operations in AMVIA implement zero-copy only for the medium message payload. Handler arguments are implicitly copied via the stack for all message sizes. Bulk receives require data copies since destination buffers are identified by an offset into a VM segment and not an actual address, thus forcing a level of interpretation not available in the VI layer. The zero copy of medium payload and implicit copy of arguments require an extra receive slot for the reasons discussed previously.

7.4 Completion Queues

A third area of design tradeoffs is evident in the use of completion queues as the AM bundle abstraction. The completion queue is perhaps the most natural projection of the AM bundle into the VI architecture and presents a simpler management model than using completion queues at the endpoint level. However, like VIs, completion queues are a limited resource of the VI provider and may not scale (i.e. they may overflow) in the face of a large number of VIs and heavy communication. Additionally, using the completion queues at the bundle level prevents migration of endpoints between bundle objects. Completion queues also expose a shortcoming of VIA in that there is no mechanism in the specification to map a user context with a particular VI. This makes it difficult to implement protocols on top of VIA since special data-structures may need to be bound to VIs. In AMVIA, this problem is resolved through the use of a separate hash table that uses the VI handle value as an index to a user-defined pointer.

8 Future Work

There are several avenues for improvements to AMVIA. The first is to improve performance. If the AM API semantics are modified to forbid modification of message contents during transit then AMVIA could provide zero-copy medium messages by first registering the user's memory as a VIA buffer and then sending the message directly from the user's memory. Bulk transfers could be implemented with VIA remote DMA write operations. This would not only reduce communication overhead, but would also greatly reduce the amount of communications buffering needed.

It would be a good idea for AMVIA to support a multi-protocol capability similar to that of Lumetta's Multi-Protocol AM [Lum98]. Another alternative would be to add a similar multi-protocol capability to VIA. Here, the primary challenge is a naming scheme that would enable VIA to distinguish loopback connections from connections between local processes. VIA connections are based on a discriminator value which is matched between two VI's to establish the connection. The VI provider cannot directly determine if a connection request sent to its own node is for the originating process or for another one. One way to fix this is to create a logical host address for each VI consumer on the system.

Two VIA features would be useful for implementing Split-C over reliable VIA, however we could not use them since our VIA does not support them:

- The VIA remote DMA read and write operations allow reads and writes directly into a target process's address space. These operations could be used to implement the Split-C get and put operations without requiring the intervention of the target process. Remote DMA writes could also be used to implement store, though a message signaling the arrival of the data would also have to be sent to the target process. Using remote DMA reads and writes in this fashion would provide 0-copy bulk gets, puts and stores.

- VIA supports a gather operation on the sending side. The messages representing bulk gets, puts and stores have two parts, a header followed by data. Using the gather facility, we could send such messages without the need to copy the data and header into a contiguous area of memory. This is analogous to the optimization for AMVIA discussed at the beginning of this section.

The use of remote DMA reads and writes is not applicable to an implementation of Split-C over unreliable VIA because remote DMA reads are not available, and the completion of remote DMA writes cannot be detected. However the gather facility could be used.

9 Conclusion

Implementing Split-C over AMVIA was relatively simple because AMVIA presents essentially the same interface to Split-C as AMII on the NOW. This implementation of AMVIA was also a good base for producing our specialized versions of Split-C on reliable and unreliable VIA. AMVIA's lack of support for local processor communication pushed us to explore providing shared memory communication in the Split-C layer.

Results from application benchmarks show that the best Split-C implementation, Split-C over unreliable VIA, is the one with the lowest processor send and receive overheads.

10 References

[Bod95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and Wen-King Su, "Myrinet: A Gigabit-per-Second Local Area Network.", *IEEE Micro*, vol. 15, (no. 1), Feb 1995, pp. 29-36

[Buo98] P. Buonadonna, A. Geweke, D. E. Culler. "An Implementation and Analysis of the Virtual Interface Architecture", *Proc. of Supercomputing '98*, Orlando, FL, 7-13 November 1998.

[Buo99] P. Buonadonna, J. Coates, S. Low, "Millennium Sort: A Cluster-based application for NT using River Primitive, VIA and DCOM." to appear in *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, 12-17 July 1999.

[Cul93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation.", *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1993. pp.1-12.

[Cul95] D. Culler, K. Keeton, L. Krumbein, L.T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. "The Generic Active Message Interface Specification.", *University of California, Berkeley*, Feb. 1995. Available at http://now.cs.berkeley.edu/Papers/Papers/gam-spec.ps.

[Cul96] D. E. Culler, L. Tin Liu, R. P. Martin and C. Yoshikawa, "Assessing Fast Network Interfaces.", *IEEE Micro*, vol. 16, (no. 1), Feb 1996, pp. 35-43

[Dru94] P. Druschel, L. L. Peterson, B. S. Davie, "Experiences with a High-speed Network Adaptor. A software Perspective," *Proceedings of the SIGCOMM '94 Symposium*, August 1994, pp. 2-13

[Dub96] C. Dubnicki, L. Iftode, E. W. Felten, Kai Li. "Software support for virtual memory-mapped communication.", *Proceedings of IPPS '96, the 10th International Parallel Processing Symposium, Proceedings of International Conference on Parallel Processing.* Honolulu, HI, USA, 15-19 April 1996. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, p.372-381.

[Dun98] D. Dunning et al., "The Virtual Interface Architecture.", *IEEE Micro*, vol. 18, (no. 2), Mar/Apr 1998, pp. 66-75

[Dus93] D. E. Culler and A. Dusseau and S. C. Goldstein and A. Krishnamurthy and S. Lumetta and T. von Eicken and K. Yelick, "Introduction to Split-C" *University of California, Berkeley*, 1993.

[Eic92] T. von Eicken, D. E. Culler, S. C. Goldstein, K.E. Schauser. "Active messages: a Mechanism for Integrated Communication and Computation." *Computer Architecture News*, vol.20, (no.2), (19th Annual International Symposium on Computer Architecture, Gold Coast, Qld., Australia, May 1992. pp.256-266

[Eic95] T. von Eicken, Anindya Basu, Vineet Buch and Werner Vogels, "U-Net: A User-level Network Interface of parallel and Distributed Computing.", *Proc. of the 15th ACM Symposium of Operating Systems Principles*, vol. 29, (no.5), Dec 1995, pp. 40-53

[Gil96] R. B. Gillett, "Memory Channel Network for PCI," *IEEE Micro*, vol. 16, February 1996, pp. 12-18

[Lum98] S. L. Lumetta, "Design and Evaluation of Multi-Protocol Communication on a Cluster of SMP's", Ph.D. Thesis, *University of California, Berkeley*, Berkeley, CA, 1998.

[Main95] A. Mainwaring and D. Culler. "Active Message Applications Programming Interface and Communication Subsystem Organization.", *University of California, Berkeley*, Dec. 1995. Available at http://now.cs.berkeley.edu/Papers/Papers/am-spec.ps.

[Mar94] R.P. Martin, "HPAM: An Active Message Layer for a Network of HP Workstations", *Proceedings of Hot Interconnects '94*, Stanford, CA, August 1994.

[Mar95] R. Martin, L. Liu, C. Yoshikawa, "Architectural Character of pico-Ray" CS-258 *University of California, Berkeley*, 1995

[Mar97] R. Martin, A. Vahdat, D. Culler, T. Anderson. "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture." *International Symposium on Computer Architecture* , Denver, CO. June 1997

[Pak97] S. Pakin, V. Karamcheti, A. A. Chien. "Fast messages: efficient, portable communication for workstation clusters and MPPs." *IEEE Concurrency*, vol.5, (no.2), April-June 1997. p.60-72.

[Pry98] L. Prylli and B. Tourancheau. "BIP: a New Protocol Designed for High Performance Networking on Myrinet." *Workshop PC-NOW, IPPS/SPDP98*, Orlando, USA, 1998.

[VIA97] "Virtual Interface Architecture Specification. Version 1.0", *Compaq, Intel and Microsoft Corporations*, Dec 16, 1997, available at **http://www.viarch.org**