An Analysis of VI Architecture Primitives in Support of Parallel and Distributed Communication

17th April 2000

Abstract

We present the results of a detailed study of the Virtual Interface (VI) paradigm as a communication foundation for a distributed computing environment. Using Active Messages and the Split-C global memory model, we analyze the inherent costs of using VI primitives to implement these high-level communication abstractions. We demonstrate a minimum mapping cost (i.e. the host processing required to map one abstraction to a lower abstraction) of 5 μsec for both Active Messages and Split-C using 4-way 550 MHz Pentium III SMPs and the Myrinet network. We break down this cost to use of individual VI primitives in supporting flow control, buffer management and event processing and identify the completion queue as the source of the highest overhead. Bulk transfer performance plateaus at 44 MB/sec for both implementations due to the addition of fragmentation requirements. Based on this analysis, we present the implications for the VI successor, Infiniband.

1 Introduction

In order to enable the widespread deployment of high performance, scalable systems, there has been a concerted effort to develop a standardized cluster communication architecture for system area networks (SAN). This effort yielded the Virtual Interface (VI) Architecture [7] in 1998, and is now focused on the emerging Infiniband architecture [1] which also seeks to encompass network based I/O. The VI Architecture defines a methodology for user-level communication based on direct memory access (DMA) descriptor processing. At its core is a set of design principles for how to implement user-level communication in a manner that virtualizes resources among an arbitrary number of processes. It outlines both a hardware architecture for the network interface controller (NIC) and a software interface upon which communication abstractions are implemented. Infiniband incorporates much of the VI Architecture, with some modifications in terminology and behavior, and represents the intellectual merger of many industry efforts in high performance networked I/O. While it does introduce some new concepts and components, its core is strongly based on the VI Architecture primitives. Thus it is important to evaluate the effectiveness of the core VI Architecture mode of operation in support of established communication APIs.

The VI Architecture exports two fundamental communication operations. One is a matched send-receive model, in which the receiver allocates and registers buffers in anticipation of incoming messages. The other is a Remote Direct Memory Access model, where the sender delivers or reads data directly to a specified region in the target's address space. Several studies and VI Architecture implementors [4, 5, 12, 13, 22, 30] document the performance achieved on the VI primitives. With experience and engineering effort, this aspect is improving.

In this paper, we focus on the cost of mapping useful communications abstractions to the VI primitives. We seek to answer the question of how effectively these primitives can support common usage models.

We consider two distinct models that have been implemented effectively on numerous substrates: active messages and a simple global memory model. The active messages [29] paradigm is centered around lightweight RPC. Communication transactions are based on two-phase request and reply messaging primitives that invoke user level handlers within the receiving application. To explore the global memory model, we use the Split-C parallel language [19]. Here the fundamental primitives are simple memory transactions: synchronous and asynchronous read and write.[1] By mapping these two models to the VI Architecture we can evaluate the relative costs of implementing communication abstractions over the VI primitives. We show the inherent costs of using the VI Architecture (regardless of the speed of the VI interface), the costs that are common to all communication abstractions and those that are unique to particular approaches.

In the next section, we present related work upon which this study builds. Section 3 explains the fundamentals of the VI Architecture and its baseline performance. In Sec-

[1]In Split-C terminology, synchronous read and write, asynchronous, split-phase get and put, and another form of asynchronous, split-phase write, called store.

tion 4, we review the Active Messages and Split-C architectures and, in sections 5 and 6, discuss their implementations on top of the VI Architecture. Section 7 presents our performance measurements of these two high-level communication layers. In Section 8, we discuss the key lessons we learned and the implications for Infiniband.

2 Related Work

Initial studies of native VI primitives [5, 12, 30] have focused on low-level details and the performance of the transport itself. In addition, M-VIA [26] and SCNet [27] demonstrate the ability to layer VI primitives over arbitrary hardware.

High-performance sorting applications (e.g. MillSort [4] and a terabyte sort [13]) were implemented over the VI Architecture to demonstrate the feasibility of the descriptor queue based primitives. In addition, web traffic workload analysis [17] suggested zero-copy VI primitives could assist in reducing server load. However, all of these studies do not analyze the costs associated in the realization of these protocols or applications upon the VI primitives.

There has been extensive work in examining protocol layer costs over other user-level communication abstractions for high-performance systems [6, 18, 20, 31], and from these we draw much of our methodology. Additionally, we use the benchmark techniques developed in [10, 11] to analyze our implementations.

Specific to the VI Architecture, [22] demonstrated the feasibility of layering the distributed component object model (DCOM) protocol (essentially an extension of RPC) over VI primitives. The results indicated that software overheads were several times the underlying transport. In an effort to mitigate this, [14] discussed a series of optimizations to the DCOM protocol to minimize the overhead. However, in that study, the costs associated with implementing the DCOM abstraction at user level dwarfed the impact of mapping to the VI architecture *per se*. Our work extends this by using a much lighter-weight starting point to isolate the characteristics of VI-based communication that result in an unavoidable cost. Several commercial efforts layer protocols over VI primitives, including the Message Passing Interface [2] and TCP sockets [8], but no detailed analysis of the mapping costs has been presented. This paper aims to provide insight into these and future implementations.

In [21], Liu describes a software-based fault injection mechanism for networked systems that was built on top of a commercial VI implementation. While this work investigated fault-tolerance of the architecture, its contribution is largely separate from this effort.

3 VI Architecture / Infiniband

In this section, we outline the VI Architecture and its basic descriptor queue messaging primitives that are carried forward to Infiniband. A baseline performance summary is included for reference in the rest of the paper.

3.1 VI Overview

The Virtual Interface is an abstraction for a protected, direct channel to the network interface controller (NIC). Communication is achieved through memory-to-memory transfers between a pair of connected virtual interfaces (VIs). Key concepts used in the VI architecture include:

- 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 principal communications buffer for network operations. Associated with each region is a Memory Handle (a unique identifier) which is 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 and provide control information and a list of pointers to data buffers.
- 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.

Each VI consists of a send and a receive work queue, their associated doorbell resources, and the user's registered memory regions. Connections between VIs are explicitly one-to-one.

There are two classes of message transactions: send-receive and remote DMA (RDMA). To initiate a network data transfer, the user process constructs a descriptor and posts it into appropriate work queue by placing a token in the queue's associated doorbell. In the send-receive paradigm, the target pre-posts receive descriptors into the receive work queue in order to identify memory regions where incoming data will be placed. The source posts a send descriptor that identifies memory regions of data to send. Each send operation consumes a receive descriptor on the target. The receiver must keep pre-posted descriptors on the receive queue to ensure incoming messages are not dropped. In this scheme, each application manages its own buffer space and neither has explicit information about the peer's registered buffers.

In contrast, with RDMA messages the initiator identifies both the source and destination buffers. Data can be directly written to or read from a remote address space without involving the target process. To conduct an RDMA operation only the sender need prepare and queue a descriptor. However, both processes must exchange information regarding their registered buffers using some out-of-band mechanism (either send-receive or another network). One exception to the one-sided nature of RDMA operations is that a small (4 byte) message, or immediate value, can be be piggybacked on an RDMA write operation. This data word is delivered to the target in a special field of a receive descriptor. Thus, this form of RDMA write with an immediate value consumes a descriptor on the target, while standard RDMA writes do not.

Completions on an individual VI are monitored either through polling or by waiting on a signal from an individual VI. However, in most parallel computing environments, each process communicates with several others using distinct VIs. Managing a group of VIs is simplified through the use of a completion queue. Completions of any of the associated VIs are posted to the completion queue and detected through polling or signals.

3.2 Infiniband

To better understand the implications of the VI architecture for Infiniband, we present a brief overview the Infiniband network architecture [28]. Infiniband is the logical merger of several industry efforts (i.e., Next Generation I/O and Future I/O) in network based I/O architectures. Here, the I/O devices are effectively separated from the host CPU(s) by a switched network fabric (Figure 1). The host channel adapter (HCA) connects directly to the memory controller and is the interface to the network. The target channel adapter (TCA) is the network interface for the individual I/O devices (e.g. disks and WAN adapters). The TCA is similar to the HCA, but can be simplified according to the requirements of the attached device(s). To provide differentiated service and robust network management, data traffic is multiplexed onto multiple independent streams called Virtual Lanes (VLs). Infiniband supports 16 VLs – 15 for data and one for management functions.

The fundamental transport interface supported by the HCA/TCA is the work queue pair (QP) which is equivalent to and exports the same messaging primitives (i.e. descriptor based send-receive, RDMA) as VIs. Data exchange between QPs is still sourced/sinked to registered memory regions established by the application. However, Infiniband provides message-level flow control schemes based on receive credits and NAK's.

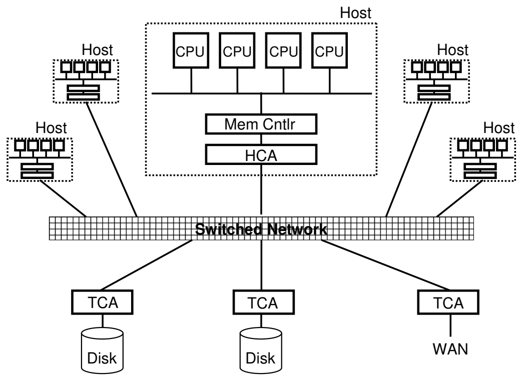

Figure 1: Infiniband network architecture.

	RTT/2 (μsec)	Throughput (MB/s)
LANai 4 VI	33	64
LANai 7 VI	30	68
Giganet cLan [15]	7	130
Compaq ServerNet II [16]	7.4	180

Table 1: Performance base of our VI implementation and comparison with dedicated hardware. ServerNet numbers are based on simulations.

3.3 Performance Baseline

For our first study, we have developed implementations of the VI Architecture for the Myrinet [3] SAN using the LANai 4 and LANai 7 interfaces. These interfaces host an on-board general purpose processor, on-board SRAM (1-2MB) and a set of DMA engines (network send and receive and host-NIC DMA). The LANai 7 interface includes an extra host-DMA engine and hardware-based doorbell support. The VI software base includes a kernel driver, a Virtual Interface Provider user Library (VIPL), and firmware for the NIC that emulates a VI-compliant device. On the LANai 4, the firmware emulates doorbells, and requires the host to wait for a previous token to be processed. The hardware doorbell of the LANai 7 eliminates this synchronization requirement. Supported messaging operations include send-receive and RDMA write with Myrinet hardware-based delivery guarantees.

Table 1 presents a performance summary of these implementations[2] in comparison with commercially available direct hardware implementations. Half round-trip-time (RTT/2) measures the application-to-application latency for a single message. Throughput is the maximum achievable bandwidth of the implementation for a given I/O architecture. The performance of this emulation is less than commercial vendors implementing native VI hardware (e.g. Gi-

[2]LANai 4 performance was measured using 2-Way 400 MHz Pentium-II SMPs with a 33 MHz, 32-bit PCI bus. LANai 7 performance was measured on 4-way, 550 MHz Pentium-III Xeon SMPs with a 33 MHz, 64-bit PCI bus.

ganet cLAN). However, the features of the LANai interface provide adequate performance characteristics with the added benefits of a flexible, instrumentable system. Moreover, our goal is to analyze the cost of mapping common communication down to the VI primitives, not the performance of the primitives themselves. The combination of the LANai 4 and LANai 7 permits an investigation of how useful certain hardware features of a device are.

4 Active Messages and Split-C

In this section, we briefly discuss the Active Messages and Split-C communication models that we have implemented over the VI Architecture as the basis for our study. The emphasis here is the semantic gap between these models and the descriptor queue model of the VI Architecture.

4.1 Active Messages

Active Messages (AM) is a simple, extensible paradigm for message-based communication in parallel and distributed computing systems [32, 9]. The Active Message mechanism may be viewed as essentially a lightweight 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 high-level purpose of extracting the message from the network and either integrating the data into the computation or sending a response message. Under AM, 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, AM handlers are similar to interrupt service routines used in OS kernels and device drivers.

Active Messages has been implemented on a virtual network scheme which supports protected multi-programming communication [23]. The architecture consists of two principal components: endpoints and bundles. The AM *endpoint* is the abstraction for a process' 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 [24] scheme in which a request message is paired with a subsequent reply message. The endpoint logically includes of a pair of buffer pools (send and receive), a virtual-memory segment, a translation table, a handler table and a protection tag, but the internal structure is opaque to the application. Endpoints also use a credit based flow-control scheme for requests to prevent network congestion and buffer overflow.

To provide flexibility for different applications, three different message sizes are supported: shorts ($<$ 32 bytes), mediums ($<$ 4 Kbytes) and bulk transfers ($<$ network MTU). 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. 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 deliver 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 end-points 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.

The AM *bundle* abstraction permits user-level polling of an arbitrary collection of endpoints. The bundle abstraction groups together related endpoints and services them as a single unit. Polling of the bundle is done explicitly through a call to AM_Poll() and implicitly whenever a process issues a request or reply.

While conceptually close to VI, AM exposes none of the detailed descriptor processing and memory registration. Moreover, it establishes a higher level discipline for message reception and processing, with the implementation responsible for achieving the necessary buffer management, flow control and event processing.

4.2 Split-C Global Memory

Split-C is a single program multiple data (SPMD) parallel extension of C. Each process in a Split-C program lives in its own local address space and can refer to data in another process through a *global pointer*, a combination of a process id and a local address. The following operations can be performed on a global pointer:

- Synchronous reads and writes.
- Asynchronous (split-phase) reads and writes (*get* and *put*), with completion detection.
- *Stores* which are asynchronous writes whose completion can only be detected in the process that is the target of the store.

These operations can be performed on the basic C primitive types (`char` through `double`), or as bulk operations of arbitrary size.

The Split-C compiler is based on a modified version of the gcc C compiler that calls specific functions for each of the Split-C memory-memory operations outlined above (read, write, get, put and store). These operations are implemented in a library that provides other Split-C functions (e.g. barrier synchronization, reductions, etc.) and deals with the startup and shutdown of the Split-C program. In the rest of this paper we only discuss the implementation of the memory-memory operations: read, write, get, put, and store. Split-C exposes only the ability to transfer data between arbitrary regions of partitioned global address space in a non-blocking fashion and to detect completion.

AM and Split-C provide significantly higher-level communication abstractions than the VI Architecture, one defining messaging discipline and one global memory transfers. The next sections detail how this semantic gap is bridged and the costs of doing so.

5 AM over VIA

In this section we discuss the internals of the Active Messages over VIA (AMVIA) implementation. The design of AMVIA underwent three major iterations in order to explore major avenues of the mapping. What is presented here is the final architecture (AMVIAv3) and how it differs from the older versions. Later, in Section 8, we highlight important design tradeoffs between the three.

5.1 Components

AMVIA preserves all the API and messaging semantics of Active Messages. Low-level details such as operating system and network hardware calls are replaced with VI Architecture primitive functions. Facilitating the mapping from AM abstractions to VI abstractions is a meta-structure called the MAP object, a name derived from the AM method AM_Map(). The MAP object is essentially a logical channel between two AM endpoints in the virtual network. Each MAP contains a VI, registered send and receive regions for descriptors and data, and a request credit counter initialized to an implementation parameter, k. The buffers are sized to support 2*k sends and 2*k + 1 receives (the need for the extra receive is discussed later). 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 remote endpoint of the virtual network.

The VI completion queue mechanism is used to deal with multiple endpoints and a bundle of endpoints. 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 MAP. The use of two completion queues permits assigning preferential service priority to receive operations.

5.2 Operations

With the exception of wait semaphores, AMVIA implements all of the AM messaging primitives. Prior to conducting communication, AM bundles, endpoints and endpoint handlers are allocated in the normal manner. When establishing the virtual network topology, each call to AM_Map() instantiates a new MAP object including the VI, sufficient registered memory space for the MAP buffers, a set of pre-posted receive descriptors and a small set of state variables. The VI is then connected with its peer endpoint VI on the remote node. The VI connection scheme uses a discriminator value to match connections requests between two VIs.

Sending operations in AMVIA use two separate mechanisms: one for short and medium messages and the other for bulk transfers. For a short or medium request, the function attempts to obtain a free send descriptor and a request credit. If either are not available, the function polls, handling incoming traffic, until it can proceed. The data payload is then copied into the appropriate message buffer and the send descriptor posted to the send queue. For a bulk transfer, two separate VI messages are used: an RDMA write followed by a matched send-receive. The RDMA write operation delivers the data directly from the application's address space to the designated offset in the target VM segment. Achieving zero-copy on the sender is achieved by dynamically registering the necessary address space. A cache of registered regions is maintained so that additional transfers from the same memory page(s) do not cause another expensive registration operation. A send-receive message is then sent in the same manner as short messages to deliver message arguments and notify the target. Replies function in a manner identical to requests, except they do not wait for a request credit. Implicit with the related request was the availability of a buffer slot in which to receive the reply.

The sequence of operations that take place in an AMVIA receive are essentially identical for all message types and sizes. Messages are processed by directly invoking the designated handler with the data arguments. Processing of medium messages, however, does not involve a copy of the data payload. Instead, the handler is invoked with a pointer to the medium message. Thus, incoming medium messages are able to exploit the zero-copy semantics intended by the VI architecture, as are bulks. Since bulks use RDMA write they are also zero-copy. 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 ensures that a reply sent by a request handler does not create a new request for which there is no available buffer. Recycling the receive descriptor before invoking the handler would require extra data copies that would degrade performance.

Invoking handlers and recycling descriptors is accomplished by the AM_Poll() operation. This method checks the receive completion queues in the bundle for incoming messages. For each received request, the routine places the message onto a queue, while replies invoke the designated handler directly. Requests are processed from the queue only when a boolean argument to the poll routine is true. This demultiplexing of incoming requests and replies and conditional execution of requests 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. 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.

The architecture described incorporates lessons that we learned from our earlier attempts at AMVIA. The first version (AMVIAv1) used three VIs per MAP, each with its own credit counter. This permitted a larger credit allocation for smaller messages while bounding the total buffer space required. One side effect was that reply messages had to be of the same size as the associated request. Also, this version did not make use of RDMA writes for large transfers.

In contrast, the second version (AMVIAv2) used a single VI per MAP, but was based completely on RDMA write transfers. Immediate values were included with each RDMA write in order to notify the target of a pending message. Flow control was achieved through a flexible buffer management scheme that was managed by the sender. The intent here, as before, was to allow more small messages to fill the network pipe. However, the complexity of the scheme, along with other factors, resulted in an unstable implementation. As such, we do not present any performance results for AMVIAv2 in this paper, but rather use it as a point of design comparison. The final version (AMVIAv3) removed the complexity of its predecessors and, perhaps not surprisingly, demonstrated the best performance.

6 Split-C over VIA

The Split-C over VI Architecture must also address connection establishment, request/reply messaging (for get and put), flow control and buffer management, but does so in the context of global memory operations. Our implementation of Split-C over VI primitives assumes a VIA implementation that provides "Reliable Delivery", i.e., messages and RDMA operations are delivered exactly once, in send order. We could not use the RDMA read or write operations directly to implement the Split-C get, put or store primitives since:

- The RDMA read operation in an optional feature according to the specification (and is in fact not available in our VI implementation).
- The actual completion of the RDMA write operation cannot be directly detected with "Reliable Delivery". A workaround involves sending a request after the write, completion of the write is guaranteed when the reply to this request is received. This scheme is used by AMVIAv3, and the resulting performance is no better than implementing 'put' using regular messages (see Section 7). We did not have available an implementation of the VI Architecture providing "Reliable Reception", which does allow detection of the completion of RDMA writes.

We therefore implement 'get', 'put' and 'store' using a credit-based, request/reply messaging protocol similar to AMVIA. The send and receive overhead is smaller for Split-C because no error checking is necessary; only the compiler builds messages, and the dispatch of AM requests and replies through indirect function calls is not required.

Split-C uses one VI per connection to another Split-C process and a simpler message layout than AMVIA. The message size is 24 bytes smaller. This smaller size reduces the number of cache misses when reading and writing messages (all messages are initially un-cached as their memory is also accessed by the NIC).

For bulk puts and gets, we can take advantage of the VI Architecture's support of segmented messages to avoid copying the data to be sent into the message — this was not possible for AM because the semantics of AM allow the data sent in a message to be modified as soon as it has been sent. However, we have to copy the data out of a received message to its destination address, whereas AM can just pass a pointer to the data to the message's handler.

Stores do not need to be acknowledged in Split-C. Thus, except for flow-control purposes, we can omit the reply to stores. After we have received n stores we send a special store-acknowledge reply that acknowledges the last n stores, thus a store only pays $1/n$ of the usual reply cost. Obviously, n must be smaller than the number of credits otherwise the system will deadlock; we pick n to be equal to a quarter of that number.

The version of Split-C over VIA with the LANai 4 has several differences from the LANai 7: it uses two VIs, one for get, put and store of primitive types, and another for bulk get, put and store. The VI for operations on primitive types has more credits than the single VI in our latest implementation.[3] The bulk operations do not use segmented messages and thus incur an extra copy when bulk data is sent.

[3]To avoid artificial differences between Split-C and AM over VIA, we use the same number of credits (8) as AMVIAv3 and the same maximum message size (4K). This artificially increases the overhead of stores as we acknowledge every other store — a production version of Split-C over VIA would increase the number of credits and the maximum message size, but would not affect our analysis.

7 Performance Analysis

Guided by past experiments of network communication architectures, we ran several benchmarking suites to identify fundamental characteristics of the Active Messages and Split-C over VI Architecture primitives.

7.1 Active Messages

Our first set of measurements test the AM over VI implementations and isolate LogP [10] model parameters using the methodology in [11]. These benchmarks illustrate fundamental parameters of high-performance network architectures. Measured parameters include latency, host send and receive overhead, and the gap, which indicates the minimum time between successive messages being sent into the network fabric. We performed LogP benchmarks for AMVIA systems with our old hardware setup (Dual PII-400, LANai 4 Myrinet NIC, VIA2) and with our new hardware setup (Quad PIII-550 Xeon, LANai 7 Myrinet NIC, VIA2). Our results are presented in Table 2.

	RTT/2	Δ	L	O_s	O_r	g
AMVIAv1 (LANai 4)	53	20	45	2.7	5.3	48
AMVIAv1 (LANai 7)	40	9	34	3.2	2.8	34
AMVIAv3 (LANai 7)	36	5	30	3.1	2.9	30

Table 2: LogP measurements for AM and Split-C. The first column shows the impact of increased hardware support. Δ refers to the increase in RTT/2 over the native VI transport. All times are in μsec. Since the minimum AM message size is 16 bytes, we compare with the RTT/2 for a 16 byte VI message (32 μsec) rather than the minimal message in Table 1.

Between the LANai 4 and the LANai 7 versions, there is an improvement of 11-15 μsec in RTT/2, Latency and the gap. This is principally due to the hardware doorbell assist features of the latter interface. The increase in overhead for the LANai 7 results from a slightly more complex access to this hardware assist.

The comparison between AMVIAv1 and AMVIAv3 on the LANai 7 is more interesting. There is a 4 μsec improvement in RTT/2, latency and the gap. This is attributed to the reduction in VI resource utilization in how AM is implemented. The NIC polls each active VI in a round-robin fashion for outstanding sends. In AMVIAv1, the multiple VIs increase this polling overhead, even though two of the VIs have no messages to send. AMVIAv3 uses only one VI and has the smallest VI polling overhead.

The main thrust of our evaluation is reflected in the column labeled Δ which shows the cost of an AM message over and above the raw VI Architecture cost. Hardware support for doorbells and minimizing the number of VIs, at the cost of registered memory utilization, reduce the raw VI cost as well. However, it remains substantial at 5 μsec or 2750 host cycles. While this inflation is perhaps tolerable in our emulation (17%), it represents a serious issue for a full hardware implementation. The mapping cost is contained within the observed send and receive overhead, indicating that the base descriptor processing accounts for (O_s+O_r) - 5 = 1 μsec. We analyze these costs in greater detail below.

7.2 Split-C

To compare the performance of the various Split-C implementations, we run a set of microbenchmarks of the Split-C's memory-memory operations: read, write, get, put, and store, for primitive C types. This is the methodology of [20] which compared Split-C implementations on several different hardware platforms. The results for both the current and old (LANai 4) implementation are presented in Figure 2.

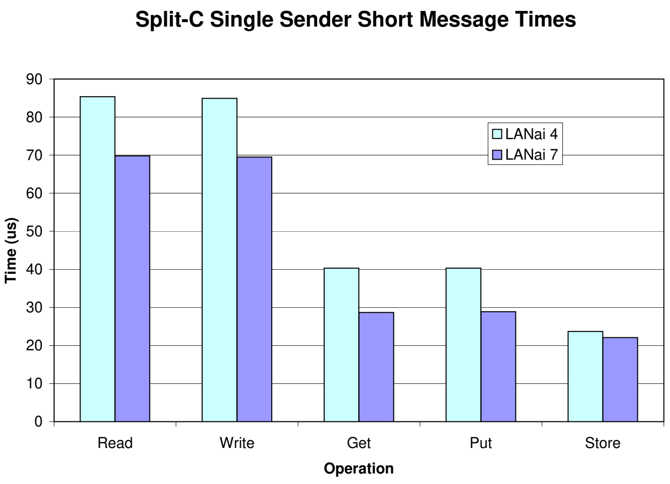

Figure 2: Split-C over VI single-sender short message times.

Between Split-C over the LANai 4 and Split-C over the LANai 7 we see approximately 12μsec of improvement (except for stores). The reasons are the same as for AMVIA. The improvement for stores is lower because our implementation of Split-C over LANai 7 has far fewer credits (8 instead of 64) for operations on primitive types, so it must send far more replies to store messages.

The comparison between Split-C and AMVIA reveals very similar performance despite significant differences in the mapping to VI primitives. The read and write tests effectively measure message round trip time. These results are slightly faster than the respective AMVIA numbers (AMVIAv1 over LANai 4 and AMVIAv3 over LANai 7). The time for repeated get or put operations (which do not wait for completion) reflect the gap. Again, the Split-C results are a slight improvement over the respective AMVIA numbers. Finally, the store results show that we get a substantial improvement when we do not have to acknowledge every store operation.

As with AM, we see a substantial mapping cost revealed in the synchronous operations, 5 μsec total (10 μsec / 2), and little impact on the asynchronous ones.

Category	Operation	% of overhead
Base VI		32.5%
	VipSendDone	7.0%
	VipPostSend	6.1%
	VipRecvDone	7.4%
	VipPostReceive	4.8%
	Recycle send descriptor	1.7%
	Recycle receive descriptor	2.0%
	Build send descriptors	3.5%
Event Notification		44.2%
	VipCQDone (Send)	22.1%
	VipCQDone (Receive)	22.1%
Flow Control		14.4%
	Send bookkeeping	5.7%
	Process received message	8.7%
Semantics		8.9%
	Read data from message	6.3%
	Act on message	2.6%

Table 3: Breakdown of VI operations required in a AMVIA short request-reply operation.

Category	Operation	% of overhead
Base VI		36.1%
	VipSendDone	7.1%
	VipPostSend	9.4%
	VipRecvDone	8.0%
	VipPostReceive	5.6%
	Recycle send descriptor	1.8%
	Recycle receive descriptor	1.8%
	Build send descriptors	2.4&
Event Notification		46.9%
	VipCQDone (Receive)	22.5%
	VipCQDone (Send)	24.4%
Flow Control		11.6%
	Send bookkeeping	4.9%
	Push msg on req/rep queue	6.7%
Semantics		5.4%
	Read data from message	3.6%
	Act on message	1.8%

Table 4: Breakdown of VI operations required in a Split-C get primitive operation. A get call involves a request to be sent by the sender to the target computer, and the matching response by the target containing the data.

7.3 Detailed Breakdown of Map Cost

To better understand the causes of overhead witnessed in the above analysis, we instrumented our AMVIA and Split-C/VI implementations to report a breakdown of host overhead for a request/reply operation. The results are grouped into four categories: Base VI, Event Notification, Flow Control and Semantics. The Base VI category reflects the overhead that occurs with raw VI operations and includes fundamental descriptor manipulation methods. The other groups are the additional costs incurred by usable communication abstractions. Event Notification groups those operations necessary to monitor for message transaction events (i.e. completion). Flow Control are costs associated with managing buffers and descriptors to prevent overruns on lossage. Semantics are overheads specific to the higher abstraction. The results are presented in Tables 3 and 4 and summarized in Figures 3 and 4.

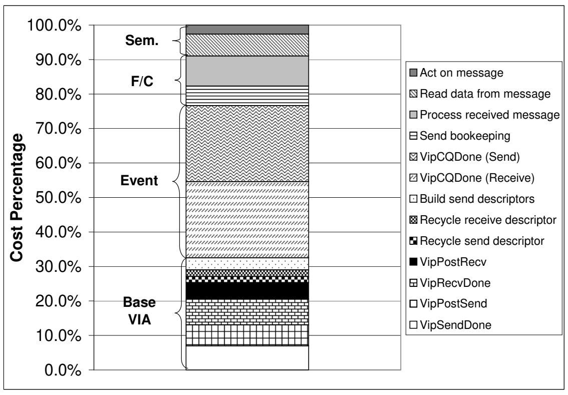

Figure 3: AMVIA timing breakdown.

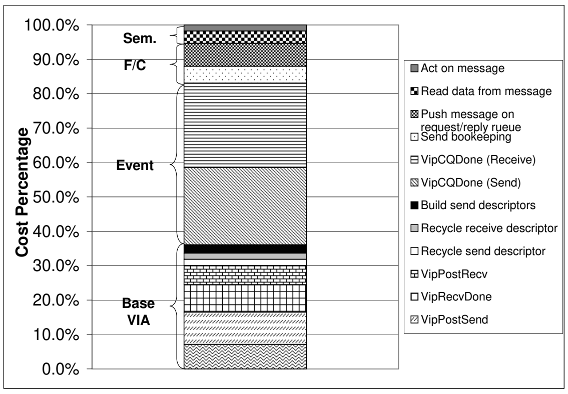

Figure 4: Split-C timing breakdown.

The largest cost is event notification associated with the routine, VipCQDone(), for both send and receives. This method executes a programmed I/O read operation to the NIC-hosted Completion queues, and if a completion has occurred, a programmed I/O write to clear it. We elaborate on the design decision to place the completion queues on the NIC in the next section. This event notification cost occurs in any real usage of the VI Architecture, but is generally not present in the published raw VI performance results because all that is required is completion of a series of one-to-one messages.

The next two major cost components are flow control associated with the VipSendDone() and VipRecvDone() methods. After an event notification, the application uses these functions to retrieve the completed descriptor off the respective work queue. These operations both involve un-cached reads to check descriptor fields updated by the interface. The methods VipPostSend() and VipPostRecv() do a programmed I/O write to post the doorbell tokens. For the Flow Control category, there is a slight increase in the time to process a message for AMVIA over Split-C due to the generalized nature of the abstraction.

Finally, we see that the cost of implementing AM semantics (get the packet, dispatching the handler) are indeed 1.5 times Split-C (get address, service read) but that these costs are dwarfed by the generic needs of event notification, flow control and buffer management.

7.4 Bulk Transfers

As a final measurement, we examine the bulk message throughput attained by the VI implementation, AMVIAv3 and Split-C/VI for various messages sizes from 4 bytes to 32 kilobytes. The results are presented in Figure 5.

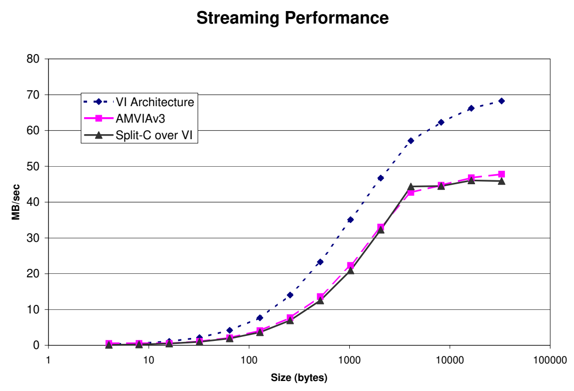

Figure 5: Bandwidth (in MB/s) attained by VIA, AMVIAv3 and Split-C for a bulk send operation.

The VI native test attains the highest throughput because it is a one-sided streaming benchmark — there is no acknowledgment of messages from the receiver. This represents the theoretical maximum bandwidth achievable by this interface.

Below 4 kilobytes, the AM and Split-C implementations achieve nearly identical throughput due to similarities in their underlying operations. In AM, send requests must be acknowledged by a reply and a copy is performed on the sender for each message up to 4K in size. Split-C also requires a response from the target in order to maintain Split-C 'put' semantics in which the sender is notified when the target has received the message. Split-C also has a memory copy on receipt since the data may be delivered to an arbitrary location in the target's address space.

To better understand the impact of this behavior on bandwidth, we perform a timing breakdown for a 1 KB message. The average difference in time per message between AM or Split-C and the native VI benchmark is 18 μsec. A breakdown of this difference is presented in Table 5.

Component	Time (μsec)
Memory copy	4
Short network receive	14.5
TOTAL	18.5

Table 5: Breakdown of per-message time difference as observed in the bulk performance test. Memory copy bandwidth was measured using the Pentium cycle counters.

The short network receive component is the necessary cost required for the interface to process a short incoming message from the network. On the LANai-based VI implementation, the interface requires a finite amount of time to process a receive. During this time, it is unable to process the next send transaction. In a steady state, the benchmark is receiving one short message for every message sent. To estimate this time, we use the single-sided time of the VI-native benchmark for a 32 byte message, 2.2 MB/s. Assuming a receive occupies the same interface time as a send, this yields a per-message time of 14.5 μsec. A dedicated hardware interface that could alleviate this cost would yield improved throughput.

Above 4 KB, both implementations shift to sending multiple messages, although for different reasons. AMVIAv3 uses zero-copy RDMA write operations, but must follow up the RDMA write with a send-receive transaction in order to deliver the message meta-data. Split-C sends multiple messages due to fragmentation above 4 KB. The effect of the transition produces the knee in the curve at approximately 44 MB/sec.

8 Discussion

The development and analysis of AMVIA and Split-C over VI primitives yield several insights. In this section, we evaluate the design tradeoffs in AMVIA and Split-C, and show how these are impacted by subtle differences in the underlying layers. We also present implications for Infiniband.

8.1 Retrospective

The design iterations of both AMVIA and Split-C/VI were intended to explore the mapping down to VI primitives from several angles. The results of this effort yielded two invariants.

The first invariant was the need for a flow-control mechanism to prevent dropped VI messages (due to buffer overruns and/or lack of available receive descriptors). In AMVIAv1, AMVIAv3, and both versions of Split-C/VI, the flow-control was based on a credit scheme. AMVIAv2 used a specialized buffer allocation system tailored for RDMA writes. In both AMVIAv1 and the first Split-C/VI, the objective was to permit more small messages into the network (approximately 64) with the belief that this would improve small message performance. In reality, the simple unified credit scheme with a credit allocation that balanced performance with required buffering proved to work the best.[4] It used less VI resources and actually exhibited better small message performance as evidenced by the LogP benchmark.

The second invariant was the need to use the completion mechanisms of the VI library for incoming messages. According to VI semantics, a host process is notified of an incoming message only when a descriptor is consumed. Due to this, we found that the send-receive model of VI communication was the best fit. RDMA writes generate notification only on the delivery of a 4-byte immediate value. For Split-C and AMVIA, this immediate is too small to include the necessary meta-data for the protocol layers. Also, the meta-data could not be appended to the message, since this could have interfered with application data structures. For bulk transfers in AMVIA, we used RDMA writes for the data, but followed it with a short VI message to carry the meta-data. Split-C/VI used a copy on the receiving end of the bulk transfer.

A key implication of the completion invariant is the requirement to use a completion queue. Any application, especially arbitrary communication protocols, that can expect to create several VI based connection will necessarily use the completion queue. Attempting to individually poll or wait on many VIs is not an efficient mechanism. This differs from simple native VI benchmarks that use only a few VIs and thus don't need a completion queue. As shown, the event notification has the highest overhead cost, principally due to the multiple programmed I/O operations. In our VI implementation, we chose to make the completion queues NIC-based because of the size of the event token (4 bytes). Our experience shows that a null DMA transaction requires 2-3 μsec with the LANai hardware. Thus a host-based queue requiring a NIC-host DMA was not a better alternative. There is also a degree of duplication in operations when using the completion queue. The VipCQDone() method only notifies an application that an event has occurred on a particular VI. The application must then invoke the appropriate follow-on mechanism to actually pop a descriptor off the queue to service the event.

[4]We arbitrarily chose a credit allocation of 8 for the later implementations.

8.2 Implications for Infiniband

The points discussed above have several implications for Infiniband and future high-performance network architectures. We separate these both in terms of implementation and semantics.

The performance breakdowns presented illustrate the cost associated with cache misses and I/O operations in communication overhead on present hardware architectures. Programmed I/O and un-cached memory transactions are expensive relative to other software mechanisms. The Infiniband HCA concept may alleviate some of this expense by interfacing directly with the memory bus and avoiding complex I/O bus interactions. Still, there is an issue of cache coherency between the HCA and the processors. Previous work with coherent network interfaces [25] that enable I/O to be cached illustrate performance gains by allowing direct reads and writes of network interface registers/memory to be cached. For operations such as completion queue checking, allowing cached reads could significantly reduce overhead, especially in the case that the event queue is empty. Alternatively, if the Infiniband mechanism uses DMA, the hardware engines must be able to provide comparable performance to memory operations, even for small transfers.

The flow-control mechanisms of Infiniband offer some promise to alleviate software based end-to-end buffer management costs. The combination of credits and receiver-not-ready NAK could eliminate the requirement for flow-control at the upper layers. Additionally, Infiniband-compliant hardware would have the ability to fragment large messages, thus preventing upper layers from having to adapt to network transmission units. We believe our implementations could benefit from both these features provided the cost of using them did not adversely affect latency or gap.

The effect of the virtual lanes in Infiniband is somewhat less clear. While the independent channels could prevent head of line blocking (e.g. between short and medium messages), a limit of 15 lanes may not be able to fulfill the service demands of all applications.

Semantically, the descriptor-based queues of Infiniband may still impose a cost to higher-level protocols because the host must format and decode the descriptors. One aspect where this is true is in small message performance. Descriptors that are as large as or larger than a small message will impose overheads both to build and manipulate. Although using the Immediate data semantic of the VI Architecture could help, it is not clear that maintaining this to a 4byte value is adequate. We suggest that, as a minimum, the immediate be able to support the precision of a pointer (typically 64-bit for future systems).

Another semantic issue with Infiniband regards memory registration. In the VI Architecture, registered memory is pinned by the user application. In one sense, this retention of physical resource by the applications results in a "not-so-virtual" interface. The Infiniband architecture retains this same semantic of pinning physical memory. As yet, the impacts of this on the large scale are unknown. Many hosts and processes could potentially result in several VI's, each requiring adequate buffer space for transactions. The flow-control and datagram features of Infiniband may alleviate this somewhat, but scalability may still be adversely impacted.

9 Conclusion

The emergence of the VI Architecture and Infiniband as SAN communication standards provides an exciting opportunity for widespread development of large-scale clustered systems. Indeed, the network-based I/O concept in Infiniband represents a significant architectural revolution for today's systems. However, their establishment as the *de facto* standard requires a deep understanding of their performance and processing cost. In this paper we have detailed the inherent cost of mapping the descriptor queue based model of these standards to two well-known communication models – Active Messages and the global memory model used in Split-C. Using these models, we analyze the necessary host processor time required to map these abstractions to the VI primitives. The results show a 5 μsec mapping cost on current hardware, regardless of the higher-level abstraction. Detailed analysis of this cost shows that the event notification mechanism of the VI completion queue to have the highest overhead at 2 μsec. In addition, we demonstrate the sensitivity of bulk message performance to key hardware capabilities.

While 5 μsec may seem small, as processors move to 64-bit architectures with sub-nanosecond cycle times, these costs will become less tolerable. As well, Programmed I/O and cache misses are unlikely to significantly improve in relation to processor performance. The implications of this for Infiniband could possibly be severe. Our discussion of these implications highlights the areas that Infiniband improves over its predecessors and where it can still make progress.

References

[1] InfiniBand Trade Association. Infiniband trade association home page. http://www.infinibandta.org.

[2] MPI Software Association. Mpi software association home page. http://www.mpi-softtech.com.

[3] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L Seitz, J.N. Seizovic, and Wen-King Su. Myrinet: a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[4] P. Buonadonna, J. Coates, S. Low, and D.E. Culler. Millennium sort: a cluster-based application for windows nt using dcom, river primitives and the virtual interface architecture. In *In Proceedings of the 3rd USENIX Windows NT Symposium.*, pages 83–92, Berkeley, CA, July 1999.

[5] Philip Buonadonna, Andrew Geweke, and David E. Culler. Implementation and analysis of the virtual interface architecture. In *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference*, Orlando, Florida, November 1998. ACM Press and IEEE Computer Society Press.

[6] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? *ACM SIGPLAN Notices*, 29(11):61–73, November 1994.

[7] Intel Compaq and Microsoft Corporations. Virtual interface architecture specification. version 1.0, December 1997. Available at http://www.viarch.org.

[8] Microsoft Corp. Winsock direct path. http://www.microsoft.com.

[9] 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, February 1995. Available at http://now.cs.berkeley.edu/Papers/Papers/gam-spec.ps.

[10] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):1–12, July 1993.

[11] David E. Culler, Lok T. Liu, Richard P. Martin, and Chad Yoshikawa. LogP performance assessment of fast network interfaces. *IEEE Micro*, February 1996.

[12] D. Dunning and et al. The virtual interface architecture. *IEEE Micro*, 18(2):66–75, March 1998.

[13] S.A. Fineberg and P. Mehra. The record-breaking terabyte sort on a compaq cluster. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 73–82, Berkeley, CA, July 1999.

[14] A. Forin, G. Hunt, Li Li, and Yi-Min Wang. High-performance distributed objects over system area networks. In *In Proceedings of the 3rd USENIX Windows NT Symposium.*, pages 21–30, Berkeley, CA, July 1999.

[15] Giganet. Giganet home page. http://www.giganet.com.

[16] Alan Heirich, David Garcia, Michael Knowles, and Robert Horst. ServerNet-II: a reliable interconnect for scalabe high performance cluster computing, September 1998. http://www.servernet.com/flat/public/brfs_wps/snetii/snetii.pdf.

[17] K. Kant and Y. Won. Server capacity planning for web traffic workload. *IEEE Transactions on Knowledge and Data Engineering*, 11(5):731–747, September 1999.

[18] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: where does the time go? *ACM SIGPLAN Notices*, 29(11):51–60, November 1994.

[19] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of the Supercomputing '93 Conference*, pages 262–273, Portland, OR, November 1993.

[20] Arvind Krishnamurthy, Klaus E. Schauser, Chris J. Scheiman, Randolph Y. Wang, David E. Culler, and Katherine Yelick. Evaluation of architectural support for global address-based

communication in large-scale parallel machines. *Proceedings of ASPLOS '96*, 31(9):37–48, September 1996.

[21] Ting Liu, Z. Kalbarczyk, and R.K. Iyer. A software multilevel fault injection mechanism: case study evaluating the virtual interface architecture. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 306–307, Los Alamitos, CA, 1999.

[22] Rajesh S. Madukkarumukumana, Calton Pu, and Hemal V. Shah. Harnessing user-level networking architectures for distributed object computing over high-speed networks. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 127–136, Berkeley, August 1998.

[23] Alan M. Mainwaring and David E. Culler. Design challenges of virtual networks: Fast, general-purpose communication. In *Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practise of Parallel Programming (PPoPP'99)*, volume 34.8 of *ACM Sigplan Notices*, pages 119–130, A.Y., May 1999.

[24] Richard P. Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Proceedings of Hot Interconnects*, Stanford, CA, August 1994.

[25] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 247–259, New York, May 22–24 1996.

[26] NERSC. M-via: A high performance modular via for linux. http://www.nersc.gov/research/FTG/via.

[27] N. Ogawa, T. Kurosawa, N. Tachino, A. Savva, et al. Smart cluster network (SCnet): design of high performance communication system for san. In *IEEE Computer Society International Workshop on Cluster Computing*, pages 71–80, Melbourne, Victoria, December 1999.

[28] Various. Infiniband tutorials. In *Proceedings of the I/O Technology Forum and Expo and Server I/O 2000*, Monterey, CA, February 2000. http://www.sresearch.com.

[29] T. von Eicken, D. E. Culler, S. C. Goldstein, and K.E. Schauser. Active messages: a mechanism for inte-grated communication and computation. In *Proceedings of the 19th Annual Inter-national Symposium on Computer Architecture*, pages 256–266, Gold Coast, Qld., Australia, May 1992.

[30] T. von Eicken and W. Vogels. Evolution of the virtual interface architecture. *Computer*, 31(11):61–68, 1998.

[31] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: a user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29, pages 303–316, 1995.

[32] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. Technical Report CSD-92-675, University of California, Berkeley, March 1992.