

Efficient Communication Mechanisms for Cluster Based Parallel Computing*

Al Davis, Mark Swanson, Mike Parker

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA
ald@cs.utah.edu, swanson@cs.utah.edu, map@cs.utah.edu

Abstract. The key to crafting an effective scalable parallel computing system lies in minimizing the delays imposed by the system. Of particular importance are communications delays, since parallel algorithms must communicate frequently. The communication delay is a system-imposed *latency*. The existence of relatively inexpensive high performance workstations and emerging high performance interconnect options provide compelling economic motivation to investigate NOW/COW (network/cluster of workstation) architectures. However, these commercial components have been designed for generality. Cluster nodes are connected by longer physical wire paths than found in special-purpose supercomputer systems. Both effects tend to impose intractable latencies on communication. Even larger system-imposed delays result from the overhead of sending and receiving messages. This overhead can come in several forms, including CPU occupancy by protocol and device code as well as interference with CPU access to various levels of the memory hierarchy. Access contention becomes even more onerous when the nodes in the system are themselves symmetric multiprocessors. Additional delays are incurred if the communication mechanism requires processes to run concurrently in order to communicate with acceptable efficiency. This paper presents the approach taken by the Utah Avalanche project which spans user level code, operating system support, and network interface hardware. The result minimizes the constraining effects of latency, overhead, and loosely coupled scheduling that are common characteristics in NOW-based architectures.

1 Introduction

Minimizing communication latency is a critical design component of parallel processing systems since parallel programs communicate frequently. Reductions in communications latency permit increased levels of parallelism between finer grained components. The result is that communication latency is the key to achieving high levels of aggregate system performance and reasonable scalability. MPP (massively parallel processor) supercomputers achieve low communication latencies and are capable of high sustained bandwidths. MPP systems

* This work was supported by SPAWAR contract #N00039-94-C-0018 and ARPA order #B990.

achieve enhanced performance by providing support for synchronization between communicating programs. Even though today's MPP systems are all based on commodity microprocessors, the logic and packaging to support the communication and synchronization mechanisms are specialized for the low-volume supercomputer market. The result is that MPP systems achieve very high levels of performance but are too expensive.

The existence of relatively inexpensive high performance workstations and the emerging availability of high performance interconnect options provide compelling economic motivation to investigate cluster based multiprocessor architectures. If performance scalability can be achieved, then a wide range of performance options can be selected to suit the customer's needs. Unfortunately, several aspects of commercial workstations and networks are significant handicaps from the standpoint of low-latency communication in a multiprocessor cluster configuration. Workstation I/O architectures and networks have been designed for generality rather than performance. These problems span the entire system framework from code path lengths in applications, libraries, operating systems, protocol stacks and device drivers to physical delays in the processor, memory system, I/O controllers, network interface, and within the interconnect fabric. It is clear that these latency problems must either be drastically reduced or effectively hidden if clusters are to achieve acceptable levels of performance. The approach described here is an interesting hybrid in that it provides both significant latency reduction as well as a mechanism for hiding some of the remaining delays.

In clusters of workstations (COWs), the largest latency component is the software code path between the application code and the physical interconnect fabric on both the send and receive sides. The biggest leverage can be therefore be obtained from improved software message passing protocols and by reduced operating system overhead. Further reduction can be achieved by improved network interfaces that accelerate these protocols and tightly integrate the interconnection fabric with the processor and its memory system. Additional improvement can be achieved by improving the *percolation effect* caused by the significant compulsory miss penalty that the CPU incurs as the receiving application starts to access the communication data.

This paper describes the communications latency reduction approach taken by the Utah Avalanche project. The Avalanche machine supports both distributed shared memory as well as message passing based programming models. However, the focus in this paper is restricted to the message passing aspects of the system. The approach concentrates on latency reduction across the whole system and provides improvement in each of the possible areas, namely:

- A new set of efficient message passing protocols and the associated applications programming interface.
- A streamlined lightweight system call interface which reduces operating system overhead while retaining traditional levels of security.
- An improved network interface which is tightly coupled with the memory system, provides hardware support for the new protocols, and acts as a

communication data cache to reduce the percolation effect.

These features are performed in a system setting which maximizes the use of commercial off-the-shelf (COTS) technology. The new network interface couples Hewlett-Packard's multiprocessor workstation technology to Myricom's interconnect fabric to achieve user to user message latencies that are less than 4 microseconds. A standard Unix operating system is used with only minor modifications to maintain traditional levels of security and safety. An additional benefit is that the protocol permits asynchronous message passing applications to be self synchronizing to alleviate the need for overhead intensive gang scheduling operations.

2 Message Latency

Quantifying message latency has been a non-standard process which varies with the focus of the particular experiment. The concern here is with total system performance and with the latency seen by communicating application programs. Latency is defined here to be the time between a sending application process initiating a send to the time the receiving application process can actually issue instructions which use the information contained in the message. Message latency is therefore the combined influence of several system components:

- **Interconnect Performance:** This includes the fall-through delay, delays imposed by connection set-up and tear-down, propagation delay, and the transit delay imposed by components in the transmission path.
- **Interconnect Behavior:** This includes overheads associated with maintaining the quality of service for aspects such as reliability, flow control, access control, packet size, and contention delays.
- **Network Interface:** Delays are influenced by the amount of software aid required to transport data in and out of the interface.
- **Memory System Effects:** In NOW architectures, the message data on the send side tends to reside in the cache while on the receive side the data is only useful when it is either in the cache or the processor registers. There is often a considerable delay incurred in moving the data between the cache and the network fabric.
- **System Overhead:** Protocol generality translates proportionately into software overhead. This includes a number of potentially hidden delays such as context switching, memory bus contention, and cache miss penalties in addition to the protocol code path lengths.

Numerous design decisions exist for each of these five areas. For NOW/COW systems utilizing commodity fabrics and processors, many of the choices are beyond the control of the system architect. The restriction is a choice based on cost and integration capability. The challenge is to maximize performance while minimizing cost by using commodity components. The use of shared media

networks results in intractable latencies under heavy loads. A high capacity point to point interconnect fabric is currently the best option.

The Avalanche approach is to use Hewlett-Packard's symmetric multiprocessor (SMP) workstations and Myricom's Myrinet-II interconnect fabric coupled by a custom network interface called the *Widget*. Hewlett-Packard provides one to four processor workstation platforms based on their HP-7200 or PA-8000 microprocessors [7, 15]. While these processors vary significantly in their internal architecture, they both use the Runway [5] main memory bus. The Runway bus is a 64-bit, split transaction bus which supports cache coherence for SMP configurations. For HP7200 based systems, the clock speed is 120 MHz with a maximum sustainable bandwidth of 768 megabytes/second in 4-way systems. The results presented in this paper are based on the HP7200.

The Myrinet-II [4] fabric runs at 160 MHz and uses bidirectional, byte-wide, data paths in each link to provide a source routed, wormhole, point to point interconnect fabric. While the Runway bandwidth is 4.8 times that of the Myrinet, the system is reasonably well balanced since the Runway must support cache to cache, and cache to memory transactions as well as I/O. Myrinet switching is done using 8 or 16 link crossbar switches with a fall-through time of 110 ns. The source routing choice provides topology independence although the particular topology will affect latency. Cascaded switches have negligible impact on bandwidth but do increase fall-through delays. Finally, cost effective cascaded switch topologies for large systems will have blocking properties. The blocking probability and associated overhead will vary with aggregate load. The performance numbers reported in this paper are based on a single switch configuration and therefore model a non-blocking cluster of modest size.

The Myrinet provides reliable in-order message delivery, as well as internal error and flow control. The Avalanche protocols depend on this reliability and, to a lesser extent, on ordering guarantees. The tacit assumption is that the Myrinet is a totally reliable network. This assumption has proven reasonable in practice for clusters that are not geographically widely distributed. Increased inter-node distances will degrade performance, and a higher level error protocol is probably necessary. A version of TCP/IP for more distributed clusters is under development. The main defect of the Myrinet technology for NOW/COW systems is the network interface card, LaNai, which incurs several microseconds of latency on each side. Additional overheads are caused by the LaNai connecting an I/O controller rather than directly to the memory bus. The LaNai choice is reasonable for LAN applications but not for tightly coupled cluster organizations such as Avalanche. The LaNai interface also precludes coherent fabric transactions and does not efficiently support the improved protocols described in this paper.

The design of the fabric interface can have a dramatic effect on both latency and overhead[20]. The overhead incurred by the CPU interactions to send or receive a message are usually difficult or impossible to eliminate from the basic latency. The Avalanche choice has been to create a custom network interface capable of directly coupling the Myrinet to the Runway bus. The important difference between the Widget functionality and that of other memory to memory

interconnect capabilities is that the Widget permits improved memory locality by providing cache coherent message handling, direct hardware support for the Avalanche protocols, and acts as a communication cache that is integrated into the receiver's memory hierarchy. From the perspective of the other processors and the memory controller in the SMP node, the Widget looks just like a processor. The downside of this approach is that the node loses a processor slot in order to achieve efficient communications. This does not need to be the case in general, but the current decision is imposed by the existing packaging options.

3 Communications and Scheduling Overhead

Overhead also results from many factors, including protocol complexity, fabric interface design and capabilities, and protocol implementation. For parallel applications, the largest overhead contributions come from data movement and from polling. Protocol design and implementation dictate the number of copy operations, with a practical minimum of two copies to move the data into and out of the communications fabric. Interface design also impacts the copy overhead. For example, DMA-capable interfaces eliminate the instruction execution overhead of these copies. Further integration of the interface into the memory system can reduce the cache, bus, and main memory overheads. Additionally, an interface which requires polling of device registers in order to check for message arrival is more latency intensive than an interface which permits arrival notifications to be posted as a cacheable memory object.

Operating system overhead is often a significant fraction of the communications overhead. System calls, context switch times, interrupt and polling overheads are all culprits. The Avalanche approach has been to provide minor modifications to a standard UNIX operating system kernel (BSD and HP-UX in this case). This provides lightweight system call and interrupt capabilities in order to retain TCP/IP levels of security without incurring the intractable latencies of traditional implementations. The result is increased efficiency without a corresponding reduction in safety.

Scheduling can impact parallel application performance in a number of ways. Delays in scheduling tasks can delay all tasks as they wait for synchronization. In a communications model where receiving tasks are required to extract messages from the fabric, delay in scheduling a receiving task can delay the sender of an otherwise non-synchronized communication. These effects have naturally led to concern that clusters of workstations running independent operating systems will exhibit poor parallel performance. Results from our own simulations² and from work done by Paikin, et. al. at the University of Illinois at Urbana show that it may be unnecessary to modify the kernel to perform coordinated scheduling.

Under high loads and with appropriate communication mechanisms, distributed parallel applications show a tendency to *self-schedule*. This improves performance significantly in terms of time-to-completion for single parallel ap-

² Space limitations prohibit presenting these measurements in this paper.

plications and in terms of throughput for multiple competing applications. An appropriate communication mechanism should provide a blocking message reception primitive for cases when the message is not yet available. It should also provide an inexpensive primitive for the case where the message is available. This form of co-scheduling is imprecise since send and receive processes may not be running concurrently. The communication mechanism must efficiently address the case where the receive process is not running. This will remove the synchronization requirement for the receiving process, and support asynchronous communications. Finally, this capability must be provided in a fashion that does not incur additional scheduling overhead elsewhere in the system.

4 Message Passing in Avalanche

Message passing in Avalanche takes a complete system approach to minimizing latency and overhead, and reduces the importance of coordinated scheduling. Lightweight, sender-based protocols[22] are used to reduce the software components of latency and overhead, and to allow simple hardware acceleration of common operations. Operating system involvement is retained in connection establishment and in message transmission. This kernel mediation on the send side permits protected use of the fabric by multiple processes. Operating system involvement in message reception is optional, at the discretion of the application. A custom network interface is used to lower overhead and to reduce processor involvement in data transfer between the fabric and the memory for both transmission and reception. Additional latency and overhead improvement is achieved by connecting the network interface directly to the memory bus rather than via an I/O adapter.

4.1 Sender-Based Protocols

The key concept of sender-based protocols[19] (SBPs) is a connection based mechanism that enables the sender to manage a reserved receive buffer within the receiving process' address space that is obtained when the connection is established. The sender directs placement of messages within that buffer via an offset within each packet header. The sender has no knowledge of the actual physical or virtual location of the buffer at the receiving end. A pointer to the base of the buffer is contained in the receive endpoint data structure established as part of the connection. This endpoint is identified by a field within each packet header. The sender can rely on an arriving message being placed within the buffer at the offset the sender has specified, as long as the offset lies within the buffer and the message would not extend past the end of the buffer. Given a reliable network, a message that is successfully sent will be received, since neither the network or the receiving processor/interface will drop the message. The protocol API provides a mechanism to permit the sender to know when the message has been sent on the send side. Sender and receiver cooperate to manage the buffer space. The receiver must inform the sender when receive buffer space

can be reused, but this responsibility is left to a higher level protocol. For some communication patterns, standard techniques such as piggy-backed ACKs might be used. For others, buffer state is implicit in synchronization. For example, an RPC ACK implies that the request has been consumed.

The Avalanche implementation of SBPs is called *Direct Deposit*[18] (DD); it provides system call-based connection establishment and message transmission. Much of the security and safety overhead is isolated to connection setup time and is provided by going through the kernel. The use of a system call allows implementation of safe and atomic shared access to the communication interface for message transmission. It also complements the security of OS-mediated connections by allowing the kernel to limit a process' access to its own connections. While mechanisms to allow secure user-level transmission via a shared interface have been implemented, they often rely on some combination of added interface complexity, limited sharing of the interface, OS scheduling support and virtual memory mapping. Retaining the system call interface for sends provides safety and flexibility at overhead and latency costs that are modest (see Section 5.2).

DD does support user mode message reception. On message arrival, the Widget writes a *notification object*, or note, into a circular queue specified by the connection. This queue can be in kernel or user memory. Each note contains a valid flag, which the Widget sets when a message arrives. A user level receive consists simply of checking the valid flag of the next note. When it is set, the user extracts the message address, size, and connection identifier from the note. Several connections within a single process can share a single notification queue, at the discretion of the user. Message buffers can likewise be in kernel or user memory, to yield a completely user-mode receive capability. A system call is provided to allow a process to sleep and wait for the arrival of a message notification on a specified queue. When the Widget posts a note to a queue, for which a waiting process is sleeping, it generates an interrupt to cause the kernel to alert the sleeping process.

A key to the performance of DD and the Widget is that receive buffers and notification queues must reside in *wired* memory. Thus, the Widget can write to them when a message arrives. They need not be mapped by the TLB, as the Widget performs memory operations using physical addresses. The use of wired memory may be viewed as non-scalable. However, the cost of the extra DRAM required by this approach is a minor fraction of the system cost. It is a small price to pay for a significant increase in performance.

4.2 The Network Interface

As seen by the CPU, the Widget provides a simple interface supporting both DIO (processor mediated) and DMA transmissions. The CPU builds a transmission request block (TRB) in a memory queue and then issues a single write to a device register to initiate transmission by incrementing the count of outstanding TRBs. The TRB contains destination information (node, endpoint identifier, offset) and source information (message address and length). A TRB contains both physical and virtual source address information for a single page of a message. A long

message utilizes a sequence of TRBs. Formation of the subsequent TRBs is overlapped with transmission of data described by earlier ones. DIO differs from DMA in that the source address specified is within a region of the Widget's on-board SRAM, to which the CPU would have copied the message data. Each DIO TRB is limited to describing only 128 bytes of data (this is the allocation unit within the Widget's on-board memory). CPU operations to initialize a TRB should hit in the cache after an initial miss. The Widget then reads the entire TRB in one bus transaction.

Completion of transmission of each TRB is signaled via a status bit in the TRB. Given a reasonable number of TRBs, the Widget will be finished with a TRB before the software needs to reuse the TRB. The software only incurs a single cache miss to determine that the TRB has been processed. A lightweight system call is provided to the user to query the status of transmission on a given connection.

The Widget and CPU generally do not interact at all on message reception. As described earlier, the Widget writes a note into memory when a message arrives. In some cases the receiver will block, waiting for message arrival notification. A status bit in the note indicates that the Widget should interrupt the CPU in this case. This is the only time in normal reception that the Widget interacts directly with the CPU.

4.3 Integrating the Network Interface into the Memory Hierarchy

The Avalanche Widget is a system memory bus resident interface. It operates as a peer with the CPU(s) in maintaining memory coherence. Both incoming and outgoing message data are moved into and out of the memory system using coherent bus transactions. The software need not flush or purge cache lines to ensure data consistency. This reduces both overhead and latency in the CPU, cache, bus, and memory system.

Incorporated into the Widget is a 256K SRAM buffer called the Shared Buffer (SB). The SB is used to stage outgoing messages and acts as a second level (1024 line, direct-mapped, physically indexed, 128 byte line) data cache for incoming message data. Outgoing messages are pipelined through the SB in units of 32 bytes (1 cache line) at a time. For long messages, the transmission of data can be overlapped while subsequent data is being fetched from the memory system. Variations in the time to acquire data are smoothed by this mechanism, providing a steady stream of data to the Myrinet.

For incoming messages, data is stored in the SB, and the Widget asserts ownership of the affected cache lines. If the message occupies a portion of a given cache line, the Widget merges the new data into the old value after gaining ownership of the line. The Widget then serves as a fast source for subsequent CPU accesses to these lines. Another benefit of buffering in the SB is that it allows data to be moved out of the Myrinet without waiting for the main memory to absorb the data. Once again, bus and memory bandwidth consumption are reduced, as is cache miss latency as seen by the CPU to access the data. Some

messages are too long to be stored in the SB and must be placed in main memory. In this case, the SB allows the transaction to be pipelined.

5 Performance Results

The fundamental performance of communication primitives of an Avalanche system based on 120 MHz HP 7200 processors and a 160 MHz Myrinet are examined here. The results were obtained from execution-driven simulations using the Paint simulator[17]. Paint simulates the HP PA RISC 1.1[14] architecture and includes an instruction set interpreter and detailed cycle-level models of a first level cache, system bus, and memory system similar to those found in HP J-class systems. In addition, it contains a detailed cycle-level model of the Widget and a simple Myrinet simulator that models contention only at node inputs. For these particular experiments, a simple processor is modeled at 120 MHz, with non-blocking loads and stores. Most instructions (except floating point) take one cycle. The cache is direct mapped, virtually indexed, 128 KB with 32 byte lines. It provides single-cycle loads and stores, a one-cycle load-use penalty, and data streaming. The Runway bus is also modeled at 120 MHz. The memory system contains 4 banks and a 17 entry write buffer. The Myrinet is modeled at 160 MHz, giving nearly 160 MB/second bandwidth; fall through time for the single switch is 14 processor cycles (approximately 116 ns.), propagation delays are 1 cycle on each side of the switch. In the experiments, cache misses to main memory had an average latency of 30 cycles, while those serviced by the Widget's cache capability had an average latency of 16 cycles. The simulation environment includes a small BSD-based kernel, used to provide device driver and system calls which initialize the Widget, set up connections, and perform message send operations.

5.1 Latency

Latency is measured by starting programs on two nodes. A receiver program establishes a communication endpoint and enters a loop polling for incoming messages. The sender program connects to that endpoint, waits to ensure that the receiver will be polling, and then sends a number of messages. Figure 1 presents the latency in 120 MHz cycles for message sizes from 4 to 2K bytes in length. Time is measured from before the send system call to when the receiver has discovered the message's arrival and touched a word from each cache line comprising the message, causing those cache lines to be brought into first level cache. The latency columns are labeled either DIO or DMA and *cache* (meaning the Widget's caching capability is used), *nocache* (meaning that arriving data is moved to main memory), or *notouch* (where the receiver does not perform the touch of the incoming data to bring it into the first level cache).

Several interesting trends should be noted in these numbers:

- DIO is competitive with DMA for messages up to 256 bytes in length.

Fig. 1. Latency vs. Message Size

Message Size	DIO		DMA	
	cache	no cache	cache	no touch
4	382	390	400	401
16	399	403	411	412
32	413	422	422	423
64	466	501	508	484
128	574	663	587	524
256	786	923	777	635
512	1210	1443	1156	855
1024	2064	2496	1915	1291
2048	3796	4568	3446	2181

Fig. 2. Overhead vs. Message Size

Message Size	Receive			DMA Send			DIO Send		
	low	high	avg	low	high	avg	low	high	avg
4	65	129	71	146	148	146	117	147	141
16	65	129	71	146	148	146	119	151	143
32	65	65	65	146	148	146	124	156	148
64	65	129	84	146	148	147	134	166	158
128	65	152	78	146	152	147	154	186	178
256	65	150	82	146	168	153	250	350	311
512	65	150	82	145	154	147	503	573	566
1024	65	89	69	145	148	146	1089	1089	1089
4096	65	89	71	145	148	147	4278	4278	4278
8192	65	65	65	243	252	248	8436	8508	8453

- The efficacy of the Widget’s cache capability becomes evident for medium sized (4 cache lines) and larger messages, resulting in latency reductions of 13% to 17%.
- Getting data into the first level cache is a significant portion of the actual latency. Comparing the DMA/cache latency with DMA notouch, the latter understates real latency by as much as 37%.

5.2 Overhead

Overhead results are reported in Figure 2. The receiver delays long enough for all messages to arrive, eliminating wait time within the library call. Timing starts before the receive library call and ends when it returns and the first cache line of message data has been referenced. This is an exact measurement of first-order message reception overhead. Second order effects such as bus/cache contention incurred as the messages arrive at the node are not measured by

this experiment. Receive overhead is independent of message size and has a small range, averaging 65 to 84 cycles. This cost is very nearly optimal, and is dominated by fundamental references to three cache lines: the notification queue descriptor, the note at the head of the queue, and the first cache line of message data.

Sender overhead is the time spent in the send system call. Timing commences before each send call and ends when it returns. Send overhead for DMA mode is nearly constant up to 4K bytes. Then it increases by approximately 100 cycles. As explained in Section 4.2, each TRB can specify the transmission of up to 4K bytes of data. DMA sender overhead increases linearly at a rate of 100 cycles per 4K increment in message size. This is the time required to allocate and initialize a TRB and increment the Widget's count of outstanding TRBs. It is independent of whether the send interface is implemented within a system call or not. Subtracting this 100 cycles from the base time of 150 cycles gives an upper bound of 50 cycles on the cost imposed by using a system call. The 50 cycles includes the library call sequence and some bookkeeping which would exist even in a user-level IO approach.

In contrast, DIO sender overhead increases linearly at about 1 cycle, 8.33 nanoseconds, per byte increment in message size. As the 160 MHz Myrinet can optimally absorb one byte every 6.25 nanoseconds, one concludes that DIO cannot saturate the fabric. Data movement from the CPU over the system bus is the limiting factor for DIO transfers.

Another performance-critical operation is polling time. It is independent of message size and send mode. A non-blocking receive serves as the message arrival polling primitive on a specified notification queue. It simply returns a status value indicating there are no messages. A simple poll on a queue with no unconsumed messages takes 53 cycles if it causes a cache miss on the note structure, which is likely on the first poll. 19 cycles are required for subsequent polls as long the note remains in cache. The posting of a notification by the Widget will purge the stale note from the cache, resulting in the previously reported time for a receive. This poll is not minimal, since it uses the receive library call. More aggressive approaches, such as compiler-generated code, could reduce this to a couple of memory references and a conditional. Polling to determine if a send has completed is done via a system call and takes 49 cycles.

5.3 An Example Program

This section reports performance data for an example program, illustrating the effectiveness of the Widget's cache and the high volume of communication sustainable while still achieving speedup. The program used is an asynchronous version of a successive over-relaxation problem. It is *asynchronous* in that each node broadcasts its portion of the current solution vector as soon as it completes a sweep. The nodes only synchronize at time steps (each comprised of 5 sweeps). The solution converges more quickly, in terms of time steps, than with a more conventional approach where nodes exchange partial solutions only at the synchronization points. This comes at the cost of significant increases in the

Fig. 3. Asynchronous Update SOR Performance

# of Procs	Message Count	Message Data	Bus Utilization		Run Time		Runtime ratio
			cache	no cache	cache	no cache	
4	446	1.476	1.97	2.16	40.92	41.61	98.3
8	500	1.72	1.13	1.34	19.38	20.14	96.2
16	1225	1.85	1.157	1.388	10.61	11.44	92.7
32	2500	1.92	1.255	1.494	6.53	7.4	88.2

frequency and volume of communication. This is impractical for systems with high relative communication overheads (see the study by Chandra, et al.[8] for a detailed description of the program and evaluation on another architecture).

Figure 3 reports a number of metrics obtained from running the program on 4 different sized systems, with and without Widget cacheing enabled. The average count of messages per node is given, as well as the volume of data (in millions of bytes) sent by each node. This is followed by the bus utilization, in millions of cycles, for the cache/nocache cases, and the run time (minus program initialization time) for the cache/nocache cases, also in millions of cycles. The Widget cache decreases bus utilization by 8.8% to 16.7%. The improvement increases with processor count. When the Widget caches the data, it provides it directly to the CPU on a miss in the first level cache. This eliminates the extra trip from the Widget to memory. The lower latency of misses to the Widget cache compared to that of main memory is evidenced by the ratio of cache/nocache runtimes in the column labeled *Runtime ratio*. There is a clear trend towards lower run times as the number of processors is increased, which in turn drives up the communication to computation ratio. The lower latency afforded by the Widget cache clearly enhances the scalability range of the program.

6 Related Work

Many groups are currently researching efficient messaging for workstations. The Hamlyn[6] effort at Hewlett-Packard Labs is similar in many respects to the Avalanche approach. It utilizes a sender-based protocol, HP Runway-based workstations, and the Myrinet interconnect. Their interface uses an I/O controller LaNai rather than the system bus. They report a one-way message latency of 12.7 microseconds with a large portion of the time spent in the interface processor and in high-latency bus translation hardware. Illinois Fast Messages[16] is another I/O-bus resident system, in this case using SPARCstations and Myrinet. Their protocol uses a more traditional buffering approach. Their use of the I/O bus LaNai interface results in a 32 microsecond latency for small messages. U-Net[2] attempts to achieve many of the same goals as Avalanche, but using completely off-the-shelf hardware. They arrive at many of the same conclusions, such as the retention of system call interface for message transmission. Positioning the

network interface on an I/O bus and the use of a processor in the fabric interface limits their latency to approximately 32 microseconds for a one cell ATM message.

Hewlett Packard's Afterburner[10] efforts have shown that the bottleneck for communication between machines is due to copy overhead. Sender Based protocols used in the Avalanche work are designed to eliminate any unnecessary copies of message data. Both the Princeton SHRIMP[11, 3] group and Digital Equipment Corporation with its Memory Channel[12] provide user-level messaging by mapping local memory pages to receiver memory pages. Message transmission becomes a series of simple stores to mapped pages, relying either on non-cacheable pages or the availability of a write-through mode in the cache to ensure that stores become visible to the network interface. This approach can deliver excellent latency for small messages (4.5 and 5.4 microseconds respectively). Neither provides the DMA capability to avoid CPU cycle stealing to support bulk transfers. Neither has the capability of serving incoming data to cache, or achieves the latency and bus utilization benefits exhibited by the Avalanche Widget.

Stanford's FLASH[13] architecture places the network interface on the memory bus by replacing the standard memory controller with the MAGIC (Memory And General Interconnect Controller) chip. The MIT Alewife[1] also places the network interface on the memory bus but connects to a custom memory controller. Both preclude the use of commercial workstation boards.

The two distinguishing features of Avalanche are its whole system approach, which attacks costs from the communication protocols all the way to the physical interconnect, and its treatment of interface card memory (the SB) as a second level communications cache to minimize the latency effects of memory percolation.

7 Conclusions

This paper has specifically described a set of design choices that have been made by the Utah Avalanche project which result in very low-latency message passing that is appropriate for NOW and COW style multiprocessing architectures. The approach, while specific to a particular commercial processor and fabric choice in the Avalanche implementation, is not vendor specific. It takes advantage of the fact that modern microprocessors are being designed to be used in both uniprocessor and small-way SMP product platforms. The primary feature that supports this capability is the provision of a system or memory bus that supports cache coherent transactions between the processors and the memory system. By providing an interconnect network interface that attaches directly to the coherent system bus, it is possible to significantly reduce the software overhead associated with message communication.

The paper has also described a new efficient communication protocol (DD) which further reduces message overheads and which provides the additional benefit of permitting the system to be insensitive to task synchronization delays.

These delays are imposed by systems which only support synchronous message transactions. The extremely efficient polling mechanism and low-latency for small messages make DD an excellent target for languages such as Split-C[9], and for message passing libraries such as Active Messages[21] and Thinking Machine Corporation's CMMD, that depend on such efficiency. The Avalanche network interface has also been designed to provide direct hardware support for these new protocols which significantly reduces software overheads on both the send and receive side while maintaining the level of security that is expected from existing TCP/IP based systems. The result is that message latency is reduced to approximately 3.5 microseconds and the need for overhead intensive gang scheduling mechanisms is unnecessary.

The results presented here are based on architectural simulations, but the chip design is underway as a .6 micron CMOS ASIC. The simulation timing models are based on currently available implementation data. A 32 node prototype system is expected to be operational in early 1998.

References

1. AGARWAL, A., BIANCHINI, R., CHAIKEN, D., AND JOHNSON, K. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (June 1995), pp. 2–13.
2. BASU, A., BUCH, V., VOGELS, W., AND VON EICKEN, T. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 1995).
3. BLUMRICH, M., ET AL. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (April 1994), pp. 142–153.
4. BODEN, N., ET AL. Myrinet – A Gigabit-per-second Local-Area Network. *IEEE MICRO* 15, 1 (February 1995), 29–36.
5. BRYG, W., CHAN, K., AND FIDUCCIA, N. A High-Performance, Low-Cost Multiprocessor Bus for Workstations and Midrange Servers. *Hewlett-Packard Journal* 47, 1 (February 1996), 18–24.
6. BUZZARD, G., ET AL. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating System Design and Implementation* (October 1996).
7. CHAN, K., ET AL. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal* 47, 1 (February 1996), 25–33.
8. CHANDRA, S., LARUS, J., AND ROGERS, A. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems* (Oct. 1994), pp. 61–73.
9. CULLER, D. E., ET AL. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93* (Nov. 1993), pp. 262–273.
10. DALTON, C., ET AL. Afterburner: A Network-Independent Card Provides Architectural Support for High-Performance Protocols. *IEEE Network* (July 1993), 36–43.

11. DUBNICKI, C., IFTODE, L., FELTEN, E., AND LI, K. Software Support of Virtual Memory Mapped Communication. In *10th International Parallel Processing Symposium* (Apr. 1996).
12. GILLETT, R., AND KAUFMANN, R. Experience Using the First-Generation Memory Channel for PCI Network. In *HOT Interconnects Symposium IV* (Aug. 1996).
13. HEINRICH, M., ET AL. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems* (Oct. 1994), pp. 274–285.
14. HEWLETT-PACKARD CO. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.
15. HUNT, D. Advanced Performance Features of the 64-bit PA-8000. In *COMPCON '95* (1995), pp. 123–128.
16. PAIKIN, S., LAURIA, AND CHIEN, A. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '88* (1995).
17. STOLLER, L., KURAMKOTE, R., AND SWANSON, M. PAINT- PA Instruction Set Interpreter. Tech. Rep. UUCS-96-009, University of Utah - Computer Science Department, September 1996.
18. STOLLER, L., AND SWANSON, M. Direct Deposit: A Basic User-Level Protocol for Carpet Clusters. Tech. Rep. UUCS-95-003, University of Utah - Computer Science Department, March 1995.
19. SWANSON, M., AND STOLLER, L. Low Latency Workstation Cluster Communications Using Sender-Based Protocols - Computer Science Department. Tech. Rep. UUCS-96-001, University of Utah, March 1996.
20. THEKKATH, A., AND LEVY, H. Limits to Low-Latency Communications on High-Speed Networks. *acm Transactions on Computer Systems* 11, 2 (May 1993), 179–203.
21. VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active Messages: a Mechanism for Integrated Communication and Computation,. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May 1992), pp. 256–266.
22. WILKES, J. Hamlyn - an interface for sender-based communication. Tech. Rep. HPL-OSR-92-13, Hewlett-Packard Research Laboratory, Nov. 1992.