Message Passing Support in the Avalanche Widget [1]

Mark R. Swanson
Ravindra Kuramkote
Leigh B. Stoller
Terry Tateyama
E-mail: {swanson,kuramkot,stoller,ttateyam}@cs.utah.edu
WWW: http://www.cs.utah.edu/projects/avalanche

UUCS-96-002

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

March 10, 1996

Abstract

Minimizing communication latency in message passing multiprocessing systems is critical. An emerging problem in these systems is the latency contribution costs caused by the need to percolate the message through the memory hierarchy (at both sending and receiving nodes) and the additional cost of managing consistency within the hierarchy. This paper, considers three important aspects of these costs: cache coherence, message copying, and cache miss rates. The paper then shows via a simulation study how a design called the Widget can be used with existing commercial workstation technology to significantly reduce these costs to support efficient message passing in the Avalanche multiprocessing system.

[1]This work was supported by a grant from Hewlett-Packard, and by the Space and Naval Warfare Systems Command (SPAWAR) and Advanced Research Projects Agency (ARPA), Communication and Memory Architectures for Scalable Parallel Computing, ARPA order #B990 under SPAWAR contract #N00039-95-C-0018

Contents

1 Introduction

The Avalanche Widget is intended to be a single chip device that provides support for efficient, low latency message passing as well as support for distributed shared memory. The Avalanche system will be a closely coupled cluster of commodity workstation processor boards connected with a high performance commercially available interconnect fabric. The current existence of commercially available high performance microprocessors at moderate costs makes it impractical in terms of both performance and cost for multiprocessor designers to design their own custom processor units. Similarly, first and second level caches in current processors are very closely integrated with the CPU chip and its timing models; the cache controllers and in some cases the caches are frequently on the processor chip itself. Changing these caches presents nearly the same difficulties as implementing a new processor. Modern processors also tend to provide mechanisms suitable for building small-way shared-memory multiprocessors. This usually takes the form of a system bus interface and protocols that support cache coherence between processors. The Avalanche approach has therefore been to utilize as much existing high performance workstation technology as possible and to leverage off the coherent bus protocols that they provide to create an efficient low-latency interface between the processor and first level cache; the communication fabric; and main memory. This interface takes the form of a custom interface chip which we call the Widget.

The specific Avalanche goal is to develop a distributed memory multicomputer that does not compromise processor performance and still delivers effective distributed system utilization. We place the Widget *acting as another processor* on the coherent system bus (Runway) of an HP PA-RISC microprocessor and connect it to a high performance commercially available communication fabric called the Myrinet. With this choice, we need to meet the following constraints:

- The Widget interface must not delay the 120 MHz Runway bus.
- The Widget must be fast enough to participate in the Runway coherency protocols without delaying the rest of the system.

One advantage from this approach is that the Widget can work with any processor that supports the Runway bus specification (this includes the HP 7200, and PA-8000 processors), without needing to re-engineer or re-implement the Widget. Another advantage is that it positions the Widget to provide an intermediate performance point in the memory hierarchy between the processor's cache and the main memory. We can do this without interfering with either the processor or its caches. This design also does not preclude use of future higher performance DRAM technology, since we also use HP's main memory controllers as well. Adaptation to other processor families would require re-implementation of the bus interface and coherence protocol engines, but the overall design should be valid for any system that provides for small-way shared memory multiprocessing.

This paper primarily discusses the message passing aspects of the Widget. The distributed shared memory aspects are described in [3].

Efficiency in message passing encompasses several cost components. Traditionally, the metrics have been latency and bandwidth. They are not however independent metrics, and from the application perspective the important metric is latency. Given this perspective, we define latency to be the elapsed time between when the sending process initiates a message send and when the received message is resident in the top level cache and is available to be consumed by the receiving process. Examining the latency components reveals several key contributing factors:

- fabric induced - the time spent propagating the message through the fabric. The two subfactors are the fall-through times at each switching point in the fabric and the time to transport the message that is limited by the available bandwidth of the fabric.
- overhead–the time spent at both sending and receiving nodes to produce or consume the message. This typically involves measuring processor involvement in terms of stalls due to resource contention and context switches, and considers code path lengths involved with protocol processing. Unfortunately it has been typical to ignore the time spent in percolating the message up through the memory hierarchy to the point where the receiving process can really consume the message. In this paper, we define overhead to only include time during which the processor cannot be performing other work, and it does not include controller or transmission delays.

The focus of this work lies in examining the overhead component more closely, especially the memory system component. We examine memory system costs and then describe an architecture that attempts to mitigate or avoid those costs wherever possible. While some of the resulting efficiencies do show up in simple latency metrics, others are only apparent in time-to-completion measurements for higher level activities, such as application run time and system throughput.

We begin by identifying particular aspects of the memory system costs in the next section. Next we describe the design features of the Widget intended to address these costs. This is followed by a description of the current implementation of the Widget. Simulation results for the proposed system are presented, and then we finally discuss related work and conclude with remarks on the implementation status and future research directions.

2 Memory System Costs in Message Passing

We identify three kinds of memory system costs that may occur in a message passing system:

1. maintaining coherence between cached message buffer locations and data retrieved from or stored to message buffers by DMA-like message operations;
2. explicit copying of data between buffers or between network interface and a buffer;
3. getting incoming message data into the highest level of the memory hierarchy in order to satisfy accesses to that data by the receiving process.

The remainder of this section describes each of these costs in more detail.

2.1 Cache Coherency Maintenance

Cache coherency maintenance can impose costs on both the send and receive operations. If the send operation utilizes a DMA-like facility, either the software must ensure that dirty cache lines in the outgoing message have been flushed to main memory *or* the DMA function must have access to coherent read operations that obtain the data from either the cache or the memory, whichever is up-to-date. The former approach has three potential drawbacks. First, the processor must execute instructions that serve simply to flush dirty data back to main memory; this represents a latency cost that is difficult or impossible to hide. This is especially true in high performance systems that employ write-back caches with significant write buffering; the sender may need to wait for unrelated writes to be retired before it can be assured that explicitly flushed lines have been sent to the main memory. Secondly, depending on the design of the memory system, including system bus and the caches, the latency of read operations from an I/O device such as the network interface to the main memory may be much greater than that of reads that could be satisfied by the cache. Thirdly, in a system where message buffers reside in the user process' address space, the process may have further need of the data. Flush operations generally invalidate cache entries in addition to cleaning them; the application would be required to wait for cache misses to recently cached data.

On the receive side, cache coherency is less likely to directly impact latency, but real costs still accrue. Any lines in a receive buffer in a non-coherent system must be ensured not to reside in the cache; two cases are possible in order to avoid hazards due to race conditions:

1. in systems which support **flush - forces writeback** operations, the network interface must deposit the received data *after* all affected lines have been flushed. Flushes performed after a message arrives might force stale dirty lines in the cache to overwrite the message data.
2. in systems which support **purge - invalidate without writeback** operations, the cpu must complete purging affected lines before accessing the received message data.

Depending on the application, appropriate buffer management may make it possible to perform the necessary flush/purge operations at times when the application is otherwise idle, as when it is waiting for a message arrival. On the other hand, idle time at processors is what parallel processing architects try to minimize. Therefore relying on such opportunities is guaranteed to compromise system performance.

If the processor/system bus combination provide support for cache coherent operations, placing the network interface on that bus with the processor(s) can eliminate the need for the software-directed flushing of outgoing message bodies. This does not necessarily mean that the resulting system will realize savings equal to the time costs just outlined. The inclusion of coherence protocols has profound effects on the design of cache controllers, bus protocols, and memory controllers, and the effect is usually an increase in complexity, overhead, and latency for memory operations. Furthermore, there may be subtler, second order effects on processor performance due to the necessity of "stealing" cache tag access cycles to perform coherency checks and cache data accesses to supply data on hits, potentially interfering with cache accesses by the processor. Synchronization of cache operations that is explicit with non-coherent caches may still occur implicitly in coherent ones as part of the coherency protocol; there may, in fact, be more synchronization, though it will be less obvious and harder to quantify. On the positive side, not only is the processor freed from explicit flush actions and attendant delays and explicit synchronization, but savings in system bus bandwidth can also be realized if data can be moved directly between the cache and the network interface.

2.2 Message Copying

Message copying costs have been the subject of many studies [1, 12, 4]. A common theme is that the initial attack on this cost should focus on the message passing software. In this paper we assume use of software, such as Direct Deposit[11], a sender-based protocol[14, 6], that supports direct transfers of message data between user process memory and the network interface. The central theme in these protocols is the use of a connection oriented approach where at connection setup time, the sender gains ownership of receive buffer memory space which the sender then manages. This results in the ability directly DMA received messages into the the receiving processes address space. In the absence of such a capability (that is, in the presence of extraneous message copying), the costs described in this section remain, although their effect will likely be overwhelmed by the direct costs of the extra copies.

A network interface that copies message data directly into memory can eliminate *explicit* message copying on message reception, subject to the structure and style of the application program. Explicit in this case means that code must run on the CPU to directly read or write to the network interface. We have already discussed cache coherence costs that arise for such an interface. An additional cost arises due to address alignment conflicts between the memory system and the network interface.

Any given network interface will support some minimum granularity of message size and address alignment. If this alignment does not match the user process' needs, then a copy operation is unavoidable. Given that the interface's granularity matches the user process' requirements, difficulties can arise if that granularity does not match that of either the network transfer unit or the coherency unit of the bus/memory system. As an example, consider a message that is smaller than a cache line in a system where the cache line size is the coherency unit size. The network interface will need to obtain both the contents of and private ownership of the target cache line and *merge* the incoming message into it before making the message available to the processor. We shall see that for short messages the latency of obtaining the cache line can be a major component of overall message latency.

2.3 Moving Incoming Data through the Memory Hierarchy

Arriving message data must be stored somewhere by the receiving device. In our model, the data is deposited directly into the receiving process' address space. This a *logical* operation that avoids the need for subsequent explicit copying from receive buffers into the receiving process space. A number of options arise for effecting the *physical* deposition of the data. The design of the interface usually dictates the choice in this regard. One common option is to present the data as a fifo to be drained by the device driver. This obviously involves an explicit copy by the processor. Another common practice is to store incoming message data in ram on board the network interface. By itself, this option does not fit well with our logical deposit of the data into the user's address space, since (some portion of) the onboard storage would need to be part of the user's address space. Such onboard storage is usually orders of magnitude smaller than main memory; this would severely restrict the number of users and/or the size of buffers accessible to them. This approach also suffers from difficulties in effectively integrating this special purpose storage into the memory hierarchy. The approach most likely to fit with logical direct deposit, with large numbers of potentially large buffers, is a standard DMA one; having the interface deposit the data in main memory.

A problem arises with the standard DMA approach, however, if the receiving application actually touches the data, since that data must then travel up the memory hierarchy to the processor as a result of a miss in the processor's cache(s). The large disparity in processor and memory speeds make it likely that the application will experience significant latencies as a result. Depositing the data directly into a high level cache can avoid such misses, but confounding effects such as cache pollution and cycle stealing, however, prevent this approach from being better, in the general case. An evaluation of the benefits of this approach is complex and more amenable to modeling than an analytic evaluation. We present some simulation results in a later section.

3 Driving Down the Hidden Costs of Messaging

In this section, we propose an architecture to eliminate or at least ameliorate the memory costs of message passing wherever possible.

3.1 Organization of the Widget

The basic organization of the Widget is shown in Figure 1. Data and control are separated at the periphery of the device in much the same way as they are in the Flash project's MAGIC[9]. Message data, whether outgoing or incoming, is staged in the Shared Buffer (SB), which is a dual-ported SRAM array logically arranged into 128-byte lines. As the figure indicates, data flows into or out of the SB only to the system bus or the network interface; the other components are generally not interested in the contents of messages and never need touch the data itself. Each port of the SB can supply/accept 8 bytes of data per clock at 120 MHz to match the burst bandwidth of the system bus; this bandwidth is more than sufficient for the network interface, which has burst bandwidth of 320 MB/second when performing concurrent send and receive operations. The division of the SB into 128 byte blocks, while the system's native cache line size is 32 bytes, is driven by bandwidth and overhead considerations. These considerations arise more strongly in the Widget's distributed shared memory function than in it's message passing function, but the concerns are related. Fallthrough times in the fabric are sufficiently large that they must be amortized over the larger block size to keep bandwidth up. The larger block size also increases the reach of the cache tags in the Widget's cache controller for larger messages, at the cost of potentially sparse utilization of the shared buffer for small messages.

Control information, comprised of transmission commands on the send side, passes through the Protocol Processing Engine (PPE), which converts those message level commands into a series of packet transmission commands that are in turn passed on to the network interface (NI). No maximum packet size is imposed by Myrinet, but the Widget uses a self-imposed maximum of 128. This fixed upper limit allows the PPE to stage data of long messages into the SB while a previous packet is injected into the fabric. An optimization in the PPE/NI interface allows the PPE to *request* that the NI append a subsequent, logically adjacent packet to a previous one if the previous one has not been fully injected when the PPE requests injection of the next one. This optimization seeks to avoid paying for repeated fallthrough times in the fabric. The flexibility of this approach allows packets to different connections to be interleaved if the software requires it and it also allows distributed shared memory messages to be injected with a maximum wait time of one packet.

On the receive side, control information consists of packet headers and message arrival notifications. The NI forwards packet headers to the PPE, while depositing the packet data into the SB. The PPE passes responsibility for message data off to the Cache Controller once the PPE has performed its protocol functions, which include validity checking on incoming packets based on state in a receive slot descriptor (`rslot`); the `rslot` is specified in the packet header. The PPE assigns an address to the incoming message data based on a buffer base address in the `rslot` and an offset from the packet header. The PPE optionally issues a message arrival notification, also based on state in the `rslot`.

A bookkeeping cache (BC) is used to provide fast access to `rlsots` and other data structures used by the PPE. Because there is potentially a large amount of such bookkeeping data, and because it is shared, albeit on an infrequent basis, with the processor, it resides in main memory and is cached as needed on the Widget.

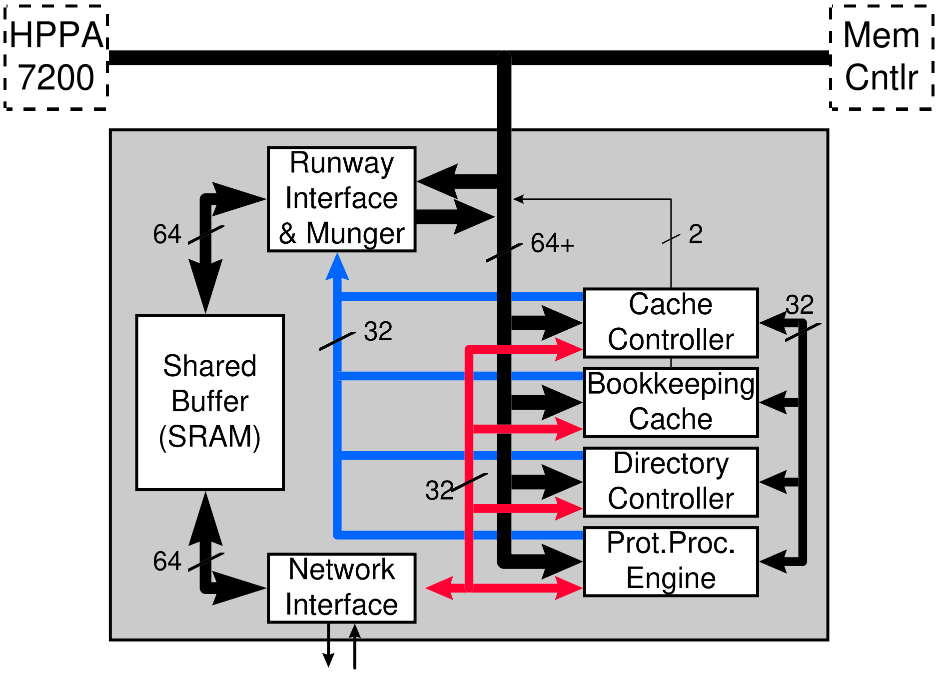

Figure 1: Widget Block Diagram

On the fabric side, the Network Interface (NI) is responsible for separating/reintegrating control and data as it passes into or out of the Widget. On the system bus side, this function is performed by the Runway Interface Module (RIM).

3.2 Cache Coherency Maintenance Approaches

The Widget's RIM coordinates access to the system bus for the Widget. In particular, the RIM issues coherent read requests for message data on behalf of the PPE. The returned message data is stored in the SB. Because the Widget uses coherent operations that are snooped by the processor, the sending process need not perform explicit flushes of outgoing message data.

For incoming messages, the Widget maintains coherence by gaining ownership and, where necessary, the contents, of all of the cache lines occupied by an incoming message. The incoming data is stored in the SB. The addresses of the lines, and their locations within the SB, are given to the CC by the PPE. The CC enters the lines in its tag arrays and snoops the system bus for accesses to these cache lines that the PPE has provided to it (and which it owns), thus maintaining cache coherency for incoming message data.

3.3 Message Copying

On message transmission, the PPE, as mentioned above, reads the message data coherently from the virtual address within the sender that was specified in the transmission request. There is no need for a software directed copy, though, of course, bus and memory (or cache) bandwidth are required since the entire message does transit the bus on its way to the Widget. There is a second copy within the Widget from the SB to the fabric, performed by the NI. This copy imposes no additional bandwidth requirements on the system, and, as we shall see, is effectively hidden by overlapping operations for all but the shortest messages.

The initial, unavoidable copy of an incoming message is performed by the NI. It collects byte-serial data from the fabric, forms it into doublewords, and writes the data into the SB. The NI passes the packet header to the PPE, along with the location of the packet in the SB. Based on connection information and an offset within the packet header, the PPE associates physical and virtual addresses with the message data and hands it off to the CC. At that point, the data is effectively resident within the receiving process' address space, since the CC will respond to bus transactions requesting the cache lines comprising the message.. Copying from the Widget to main memory is not required to accomplish "delivery" of the message.

3.4 Memory Hierarchy Injection

A message is effectively present in the memory hierarchy once the CC assumes ownership for the lines comprising it and the NI has finished depositing incoming data into the associated SB locations. When the processor accesses the message data, the CC can provide the data, and transfer ownership, significantly faster than the main memory can. Since the CC has given up ownership to the data, it can safely free up the associated SB locations once the contents are placed on the bus. The Widget serves as a kind of second level cache for incoming message data.

As with any cache, finite size can become a problem. The CC must deal with both capacity and conflict constraints. The Widget's design introduces a decoupling of the data storage and the tag storage used by the CC. The data storage (SB) has other uses such as staging outgoing messages and holding incomplete incoming message cache lines that the PPE is not ready to hand off to the CC. Thus a capacity miss can occur even when tag space is available. Conflicts can occur too, of course, when incoming message data maps to a tag entry that is already in use. In either case, the CC must dispose of the unwanted cache line data, of which it has the only valid copy. It does this by writing back the line to main memory. In the case of a capacity miss, an LRU algorithm is used to select the victim; in the conflict case, the the current occupant of the tag entry is the victim. Obviously, subsequent accesses to such data will pay the full cost of moving back up the memory hierarchy. Currently this write back only occurs as a reaction to conflict and capacity events. An obvious improvement is to utilize a more context sensitive CC which is capable of determining whether the message data is likely to be used and if not to eagerly send some data to memory to avoid writing back other, more likely to be used, data. This context sensitivity issue is also part of the Widget design but is beyond the scope of this paper. If the application has good temporal message locality (i.e., messages are consumed soon after arrival), the SB should seldom experience capacity problems, as lines would be freed by the act of consumption.

4 Efficient Message Passing in a Cluster of Commodity Workstations

In today's commodity systems, processors and caches–controllers, tag rams, and data rams–are generally very tightly coupled and exhibit extremely fine tuning of their respective timing characteristics. Modifying any one of these components or their interfaces and timing characteristics is likely to require re-engineering all of them to maintain desired performance levels. In Avalanche, we have chosen to avoid designing any mechanism that would require non-standard features in the processor or closely coupled caches. Instead we have designed a device that sits on the same system bus as the processor–and which is, from other bus clients' perspective, just another processor. Because we have chosen not to modify the cpu or high-level caches, a system with an Avalanche Widget installed should run uniprocessor applications at full, non-degraded speed. Message passing applications should also execute at full rated processor speed with the exception of operations that touch incoming message data, where any degradation will appear as cache misses; the scope of this degradation will be a measure of our success in the design and implementation of the Widget.

4.1 The Simulation Model

The simulations we report were performed using the PAINT simulation environment. PAINT provides cycle-by-cycle simulations of a distributed memory multicomputer composed of HP PA-RISC processors[5]; it is based on the MINT simulator developed by Veenstra[13]. In keeping with HP's system design approach, the memory hierarchy we model consists of just two levels: a large, direct mapped, virtually indexed, coherent, single level cache and a main memory on a high frequency, split transaction bus. In particular, we model single-issue processors running at 120 MHz with non-blocking writes, read interlocks, and up to 6 outstanding cache misses. We model a bus similar to HP's Runway bus[8] that operates at 120 MHz, is 64 bits wide, and delivers an effective bandwidth of 768 MB per second. The interconnect in our simulations is based on Myrinet[2]; we model a 160 MHz, 8-bit wide, full duplex connection to the fabric switch, with 1 cycle transmission delays to and between switches, and a switch fallthrough time of 45 cycles (at 160 MHz).

4.2 Simulation Results

In Table 2, the latencies of fundamental operations used to implement message passing are shown. For most cases, the times shown are independent of whether the cache implements coherence; where the times differ, the number in parentheses is for the coherent cache. These times are microbenchmarks, of course, and dynamic timings in a real system running real applications will exhibit some variance based on bus contention, coherency requests from other processors, the contents of write queues, etc. Items of note are the significant latency for read misses, which is representative of modern machines, and the significant cost of flushing a page–between 2.3 and 6.5 microseconds. Note also that the time for a read miss when the Widget's CC can supply the data is only 12 cycles, compared to 33 when main memory must supply the data.

In Table 3, the elapsed times for user-to-user message transmissions are shown for varying message sizes and different architectural models. Timings are reported in 120 MHz cycles. Messages of three sizes were sent: the native cache line size (32 bytes), the Widget SB block size (128 bytes), and the system page size (4096 bytes). In all cases, the message data lines are dirty in the cache at the point the send operation is initiated.

Figure 2: Constituent Operation Timings

Operation	Comment	Cycles
Flush cache line	Either clean or dirty	1
Purge cache line	Either clean or dirty	1
Read miss	ignore result	1
Read miss	use result in next instruction	30(33)
Read miss	use result; line in Widget	NA(12)
Read hit	Either clean or dirty, use result	2
Sync cache	Empty write back queue	11
Sync cache	one entry write back queue	31
Flush cache page	all lines clean	273(274)
Flush cache page	all lines dirty	731(799)
Purge cache page	all lines clean or dirty	273(274)

The first group of times is for a "standard" DMA architecture; message buffers are explicitly flushed from the cache before the send is issued and arriving message data is written to main memory rather than being retained in the SB. The bandwidth for the one page transfer comes to approximately 113 MB/second. The time to consume the data (which in this test simply involves reading one word from each cache line) is nearly as large as the time required to perform the complete message transfer.

In the second group of times, we simulate a system with a coherent cache but with no caching capability in the Widget. The message transfer time drops from the previous case by a factor of 16% and the bandwidth increases to 134 MB/second. The decrease in transfer time is accounted for by the removal of the cache flushing operations that the non-coherent cache case required. The time to consume the message actually increases with the addition of cache coherency, as we suggested in our earlier discussion. The read miss time per line is increased slightly (from 30 to 33 cycles); this accounts for the bulk of the difference.

In the final group of times, we model the actual Widget. The cache is coherent and the Widget stores incoming message data in its SB. The message transfer time is essentially unchanged, as we would expect, but message consumption time drops dramatically, as the Widget's CC is able to supply message cache lines at SRAM speeds. Overall, the total real latency for a one page message goes from 8467 cycles for the standard DMA case to 5433 for the Widget case, an improvement of 35%.

The achieved bandwidth of 134 MB/second represents 84bandwidth of a 160 MHz Myrinet. The difference (about 650 cycles for a 4096 byte message) is taken up by software protocol processing on each end, as well as the staging of the initial packet to the sender's SB (which is not overlapped with tranmission time) and the acquisition of ownership of the lines comprising the final 128 bytes at the receiver, which is only partially overlapped with reception time.

5 Related Work

A number of other groups are also pursuing combined message passing and distributed shared memory (DSM) architectures. In general, the Avalanche project differs from the other hybrid approaches in

Figure 3: Message Transfer Timings

Cache Protocol	Received Data Deposited Where	Msg Size in bytes	Cycles to Message arrival	Cycles to Consume Data
Non-coherent	Memory	32	422	50
Non-coherent	Memory	128	547	142
Non-coherent	Memory	4096	4328	4139
Coherent	Memory	32	381	53
Coherent	Memory	128	474	154
Coherent	Memory	4096	3644	4634
Coherent	Widget	32	371	22
Coherent	Widget	128	476	65
Coherent	Widget	4096	3632	1801

viewing message passing as a valid programming model in its own right, and not just as a bulk transfer mechanism that can overcome a shortcoming of DSM in moving large quantities of data.

Alewife[7] represents one of the earliest hybrid distributed shared memory/explicit message passing systems. Its approach was very invasive, requiring fabrication of a custom version of a SPARC[10] cpu. Message handling received little support, other than limited DMA capability, necessitating significant processor involvement in message reception and a high reliance on interrupts. Alewife implemented local cache coherence, as does the Avalanche Widget. Little is reported on the memory system performance implications of their design.

The FLASH project's Magic bears significant superficial resemblance to the Avalanche Widget, especially in its provision of separation of control and data within the Magic chip. Three major areas of divergence are apparent however:

1. Magic uses software executing on a microprocessor to perform protocol processing; this appears to limit the parallelism that can be achieved within the device for concurrent send and receive operations;

2. Magic *is* the memory controller in the FLASH scheme; this means that all memory references are mediated by it and implies the potential for noticeable interference between message passing and completely local memory activities,

3. MAGIC implements global cache coherence, which the Alewife report argues effectively against.

Other studies of memory hierarchy and message passing interactions have appeared recently. Pakin reports on simulations of single nodes of various communication architectures on the "fringe" of a multicomputer. The results show the memory system effects of message traffic on an unrelated computation. Although this approach allows some broad brush conclusions, the simplifications involved lead to very weak conclusions.

6 Conclusions

We have described and quantified a number of significant memory system specific costs of message passing. The architecture we proposed addresses these costs and effectively reduces or eliminates them in many cases. We are currently in the process of implementing this design for use in a network of commodity workstations (based on HP's PA8000 processor). This implementation will be reported on as results become available.

Another component of the Avalanche is the provision of distributed shared memory in the same system. This work, too, is reported separately, as will be experiences in mixing the two models in the same system.

The approach to physical message deposition reported here is a very simple, reactive method that can work well for applications with relatively small instantaneous requirements for message data–alternatively this can be stated as applications with good temporal locality between processor and the message passing mechanism. Our future work includes addressing a more general set of applications where not all fresh message data can fit in the Widget cache and an informed decision must be made in deciding what to cache and what to store away in main memory. We will also explore more active placement strategies, such as issuing update writes when there is high probability that the high level cache(s) can profit from data being "pushed" into them in anticipation of imminent use.

References

[1] Banks, D., and Prudence, M. A High-Performance Network Architecture for a PA-RISC Workstation. *IEEE Journal on Selected Areas in Communications 11*, 2 (February 1993), 191–202.

[2] Boden, N., cohen, D., R.Felderman, Kulawik, A., Seitz, C., Seizovic, J., and Su, W. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro 15*, 1 (February 1995), 29–36.

[3] Carter, J., and Kuramkote, R. Avalanche: Cache and DSM Protocol Design. Tech. rep., University of Utah, Apr. 1995.

[4] Druschel, P., and Peterson, L. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles* (December 1993), pp. 189–202.

[5] Hewlett-Packard Co. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.

[6] Kronenberg, N., Levy, H., and Strecker, W. An Analysis of TCP Processing Overhead. *ACM Transactions on Computer Systems 4*, 2 (May 1986), 130–146.

[7] Kubiatowicz, J., and Agarwal, A. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the 7th ACM International Conference on Supercomputing* (July 1993).

[8] Kurpanek, G., Chan, K., Zheng, J., DeLano, E., and Bryg, W. PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface. In *digest of papers, Spring COMPCON 94* (March 1994), pp. 378–382.

[9] Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture* (April 19943), pp. 302–313.

[10] SUN Microsystems, Mountain View, CA. *SPARC Architecture Manual*, 1988.

[11] Swanson, M., and Stoller, L. Direct Deposit: A Basic User-Level Protocol for Carpet Clusters. Tech. Rep. UUCS-95-003, Compuer Systems Laboratory, University of Utah, September 1995.

[12] Thekkath, C., and Levy, H. Limits to Low-Latency Communications on High-Speed Networks. *ACM Transactions on Computer Systems 11*, 2 (May 1993), 179–203.

[13] Veenstra, J., and Fowler, R. Mint: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *MASCOTS 94* (January 1994).

[14] Wilkes, J. Hamlyn - an interface for sender-based communication. Tech. Rep. HPL-OSR-92-13, Hewlett-Packard Research Laboratory, November 1992.