LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

An NIU Architecture for Open S & M Systems

Computation Structures Group Memo 392
November 25, 1997

Boon S. Ang, Derek Chiou, Larry Rudolph and Arvind

This paper describes reserach done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

An NIU Architecture for Open S & M Systems

Boon S. Ang, Derek Chiou, Larry Rudolph and Arvind

November 25, 1997

Abstract

This paper presents the architecture of a network interface unit (NIU) to provide a wide range of shared memory and message passing (S&M) semantics within the confines of an *open* system composed of a cluster of SMP's and a high speed interconnection network. An open system allows commodity items to be easily assembled and replaced without requiring hardware modifications to the commodity components.

The StarT-Voyager system, which is built using PowerPC 604e-based SMP's from IBM and the low-latency, high-bandwidth Arctic network developed at MIT, is an example of such a system. The StarT-Voyager NIU card plugs into a processor card slot of each SMP and implements shared memory on top of a message-passing layer which is fully exposed to application programs. Four message passing mechanisms are directly supported to achieve very high performance over a wide spectrum of communication types and sizes. The NIU is also capable of supporting protocols for a wide variety of memory models. StarT-Voyager's protection scheme improves upon past designs by not requiring strictly synchronized gang-scheduling, and by supporting non-monolithic protection domains in a multiprogrammed environment. Performance predictions, based on fully synthesizable Verilog show StarT-Voyager's novel message passing mechanisms offer a definitive advantage in a multi-threaded environment without compromising the performance in a single-threaded environment. Preliminary shared memory simulations are also very promising.

1 Introduction

It is generally believed that scalable parallel computers will consist of commodity SMP servers augmented with an intelligent interface to a high-performance interconnection network. Most designs of the network interface unit (NIU) directly support only one communication mechanism or one memory model[19, 7, 14, 28], and thereby limit the functionality and performance. This paper presents an architecture that overcomes these limitations and provides mechanisms to support a variety of message passing and shared memory models in an open system.

Shared memory and message passing (S&M) provide different programming abstractions and the debate as to which is better has been raging for more than a decade with no clear winner in sight. Each appears to have its advantages, and each is likely to be around for a long time[15]. Unfortunately, while it is generally accepted that large distributed shared memory systems (DSM) should be built on top of a message passing substrate, DSM systems do not expose this substrate. Similarly, in systems that only support message-passing, it is difficult to implement cache-coherent DSM since there is no access to internal hardware cache state. Employing two different hardware solutions works but is expensive and wasteful both in terms of development time and hardware costs. An appropriate architecture should support applications with complex communications interactions without compromising on protection.

As SMP's become commonplace, they not only become the natural basic building blocks of a larger multiprocessor, but also enable an *open* scalable parallel processing system. An open system allows commodity items to be easily assembled and replaced without hardware modifications. They avoid the difficult task of tightly coupling the development and delivery of both the processing nodes, the network, and the processor-network interface unit (NIU), Machines such as Flash [16] and Alewife [1] share goals similar to StarT-Voyager's but are not open systems. Connecting the NIU to a standard I/O bus is the typical way of providing an open system [2, 31, 4, 10, 3, 11], but current I/O bridges severely limit the performance and do not expose enough processor cache state to build cache coherent DSM's. The multiple processor slots of SMP's make ideal attachment points and do not require modifications to the SMP architecture. However, cooperation with the SMP producer is required because the memory bus interface, unlike the I/O bus, tends to be proprietary.

A minimal hardware system meeting our design goals is an NIU consisting of an embedded microprocessor and interface units to the SMP and the network. As the number of processors within the SMP and the number of concurrent, interactive, parallel jobs increase, the embedded microprocessor is likely to become a performance bottleneck. The solution is to migrate common case functionality to specialized functional units.

The StarT-Voyager machine embodies these features in an actual implementation consisting of 32 IBM Doral PowerPC 604e-based dual SMPs and the Arctic [5] high performance interconnection network. The StarT-Voyager NIU contains an embedded processor with hardware assistance to enhance performance. *StarT-Voyager can implement virtually any type of message-passing, shared memory, or synchronization primitives found on any multiprocessor with performance competitive to that of a direct implementation.* It also allows easy extensions to the semantics of memory access operations. StarT-Voyager is expected to be operational in the spring of 1998.

We begin with a discussion of the requirements for the NIU for open S&M systems (Section 2) and then present a minimalist design for such an NIU in Section 3. The paper suggests various hardware enhancements for improving performance in Section 4. The actual StarT-Voyager design is given in Section 5, and its projected performance, based on synthesizable Verilog code, is given in Section 6. Finally we present some conclusions in Section 7.

2 Requirements

The basic requirement is to form a scalable and open parallel processing system from a hardware platform consisting of commercial SMP's and a high speed interconnection network. It is envisioned that the system will be used for a wide range of high performance applications, including both technical and commercial computing. Thus, the main system requirement is to support multiuser, multithreaded, parallel jobs in a time-sharing mode that permits flexible scheduling. The main communication requirement is to support a wide range of distributed shared memory and user-level message-passing models. The following requirements for these flexible communication mechanisms implicitly include low development cost as well.

2.1 Shared Memory

The basic requirement is to support a large cache-coherent shared memory space. Because applications exhibit a range of shared memory data usage patterns and the efficiency of any particular coherence protocol depends on the usage pattern, an efficient shared memory implementation should allow for multiple protocols. In addition, in order to provide very large caches to minimize conflict and capacity cache misses, it makes sense to use local DRAM as a cache for global state when there is locality of reference, as done with S-COMA [30]. When shared memory is used in a sparsely accessed mode, it makes sense to support a NUMA [28] style of shared memory.

Shared memory models are still evolving because of both programming and performance concerns. It is likely that there will be several different models for shared memory and within each there may be several adaptive coherence protocols to exploit particular access patterns. Hence, flexibility is of critical importance. The system should provide support so that cache coherence protocols are programmable and multiple protocols can operate at the same time, although on different addresses.

2.2 Message Passing

Different parallel programs have different message passing requirements. Consequently, flexible mechanisms for managing, sharing and polling message buffers are required to build an efficient message passing substrate. Some programs have very static message passing patterns; it is possible to determine at compile time which processor will send messages to which other processors, and at what time. Others have more dynamic message passing patterns, which can only be determined during runtime and may even vary from run to run.

Different programs, or different phases of the same program, tend to send messages of very different sizes, which come from very different levels in the memory hierarchy. In order to support a wide range of message passing codes efficiently, the following spectrum of message types needs to be supported:

Short: Data for short messages consists of a word or two that generally resides in registers and is to be sent as soon as possible. Short messages are generally received in registers as well to permit their immediate processing.

Medium: Data for medium messages is generally several words in length. As such, they are likely to reside in the cache when ready to be sent. Similarly, their reception should be into the cache. Such messages are required for cache coherence protocols.

Long: Data for long messages is mostly if not entirely found in memory. Thus, a long message communication involves a memory-to-memory transmission often accompanied by shorter message transmissions indicating when the whole message body is ready for transmission and when it has been successfully received.

Efficiency concerns dictate that message latency, *i.e.*, end-to-end transmission time as well as processor overhead time, should be proportional to the message size. Thus, short messages should have very low latency while long messages may suffer longer latencies. Finally, a protection mechanism is required to prevent unauthorized transmission and reception of messages.

2.3 System Requirements

There are few "cluster" operating systems in use today. Both to keep the development costs down and for greater fault tolerance, we require each SMP in the cluster to run its own copy of the operating system. Parallel or cluster wide services should be layered as extensions to the standard OS. By limiting the interaction of OS's on different sites to a well defined and protected interface, it is easier to isolate faults to individual SMP.

A multi-user environment in an open S&M system must provide protection mechanism that ensures the logical independence of communication belonging to non-interacting jobs. This means not only the ability to control the sending and receiving of messages, but also confining message traffic blockage to the perpetrating job. Most parallel machines today enforce protection via job scheduling. The processors are assigned to a parallel job as a *gang* and the job is not scheduled until the system has been drained of all messages belonging to the previous job.

Using gang scheduling as the means of enforcing protection prevents concurrent use of the network by user applications and system support facilities, such as a parallel file system. The need for more sophisticated protection mechanisms is especially acute in an S&M system, due to the presence of both system generated shared memory and user generated message passing traffic in the network.

No current distributed parallel systems adequately addresses job scheduling in the presence of page faults and temporal variations in program parallelism. For many, the protection model requires gang scheduling and leaves no other options. When a page fault occurs, such a scheduling regime either allows the page faulting processor to idle or to run a sequential job. Ideally, the scheduler should allow a parallel application's processor utilization to expand and contract over time, with other parallel jobs scheduled on unused processors.

Finally, parallel programs may employ shared memory and message passing to interact both with its constituent processes and threads and with the processes and threads of other concurrently executing parallel programs. For example, a server application may want to communicate with a number of client applications without allowing a client to communicate directly with other clients. This requires the protection scheme to enforce non-monolithic, and possibly asymmetric communication domains.

3 A Base NIU Design

This section describes a simple NIU design that supports the full set of capabilities outlined above. It provides a base design, upon which hardware enhancements can be added systematically to improve performance while maintaining the same interfaces to the network, and to the SMP memory bus. Before describing the design it is important to understand why a pure software approach is insufficient. Given direct access to the network, low level message passing libraries easily implement user-level message passing. Distributed shared memory can also be handled in software through the use of sophisticated compilation tools such as Shasta [27] that also directly interface to the network communication. But these schemes become less attractive in multiuser, multiprocessor, SMP environments where many processes wish to communicate over the network concurrently. Either processes are trusted to correctly handle packets belonging to other processes, thereby compromising protection, or the operating system must mediate network access, thereby compromising performance.

3.1 Assumptions: SMP Memory Bus and the Network

For our design, the SMP must be capable of supporting multiple bus *masters* and *slaves* as well as bus *snoopers*. A bus master can initiate all bus operations, including cache management operations such as flush. It is assumed that the SMP systems support *intervention* in which there are two slave devices associated with the same address range – a primary, *e.g.* the memory controller, and a secondary, *e.g.* the NIU bus slave unit. The secondary unit can dynamically intervene preventing the primary from satisfying a bus transaction and itself responding to the transaction. Traditional look-aside caches require such intervention support.

Finally, it is assumed that the NIU is able to delay committing to a bus transaction. In particular, consider the case of a processor requesting exclusive access to a cache line. In most SMP's, once a slave has committed to such a bus transaction, it is assumed that the initiator has ownership of the cache line. Further transactions to the same address are disallowed until the first

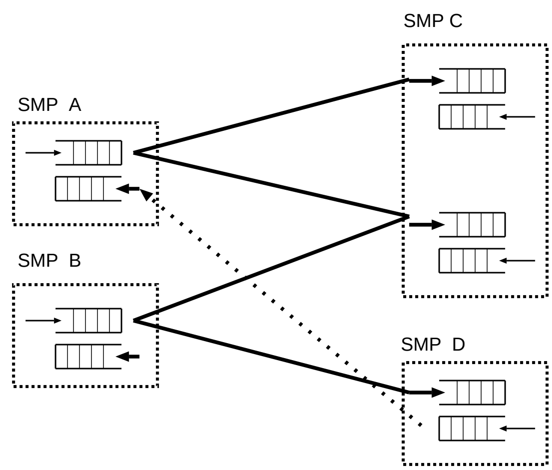

Figure 1: Messages are sent from TxQ's to RxQ's without any a priori restrictions. A TxQ at SMP A can send to two different RxQ's located at SMP C, and a TxQ at SMP B can also send to two different RxQ's but located at separate SMPs. Notice that the bottom TxQ (at SMP D) can send to an RxQ at SMP A, although the reverse is not true.

transaction has been satisfied. We require that a slave can cause a transaction to be *retried* at a later time rather than having to commit to it. In some systems, the transaction will be continually retried, while in others it will only be *rerun* when the slave indicates it is ready. Still other systems allow a slave to *take back* or undo its commitment.

There are fewer assumptions on the interconnection network. We assume reliable transmission, and support for at least two message priorities in the network. The latter is desirable to help avoid deadlocks in cache coherence protocols. A FIFO network is convenient for cache coherence protocols, though not an absolute necessity.

3.2 Message Passing with Virtualized Queues

To provide protection, flexibility, and efficient use of resources, message passing communication is implemented with operations on *virtual transmit* and *receive queues*. The sender enqueues a message into a local *transmit queue* (TxQ) and the receiver dequeues a message from a local *receive queue* (RxQ). Messages in the TxQ specify virtual destination RxQ's (see Figure 1). Virtualization of message queue names enables migration of virtual receive queues, and permits a queue to be locally mapped to local hardware resources. Virtual receive queue names go through translation twice: once before being launched into the network and once upon arrival at the destination NIU. A *conversion table* is used to give the physical destination NIU site number required by a network packet for routing, and associates a virtual RxQ name used at the destination site.

To enforce flexible protection, applications refer to destination RxQ's queues using short logical names associated with a local TxQ (Figure 2). Messages can only be sent to destinations with valid entries in the table. Protection is enforced by restricting who can modify the conversion table. A virtual source field, which is also read from the conversion table, is automatically included in each message so that the receiver can communicate back with the sender, while still allowing the sending site to locally map its receive queues.

Our scheme provides greater flexibility than common schemes such as attaching to every outgo-

TxQ	Logical Dest 0	Logical Dest 1	...	Logical Dest n
0	$\langle$site$_A$, RxQ$_\beta$, source$_i\rangle$	$\langle$site$_B$, RxQ$_\delta$, source$_j\rangle$	...	$\langle$site$_C$, RxQ$_\zeta$, source$_k\rangle$
1	$\langle$site$_A$, RxQ$_\theta$, source$_l\rangle$	$\langle$site$_E$, RxQ$_\iota$, source$_m\rangle$	...	$\langle$site$_F$, RxQ$_\lambda$, source$_n\rangle$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	$\langle$site$_G$, RxQ$_\nu$, source$_o\rangle$	$\langle$site$_H$, RxQ$_\pi$, source$_p\rangle$	...	$\langle$site$_I$, RxQ$_\sigma$, source$_q\rangle$

Figure 2: Conversion Table: each row contains the logical destination to $\langle$site, RxQ, source$\rangle$ mapping for a transmit queue.

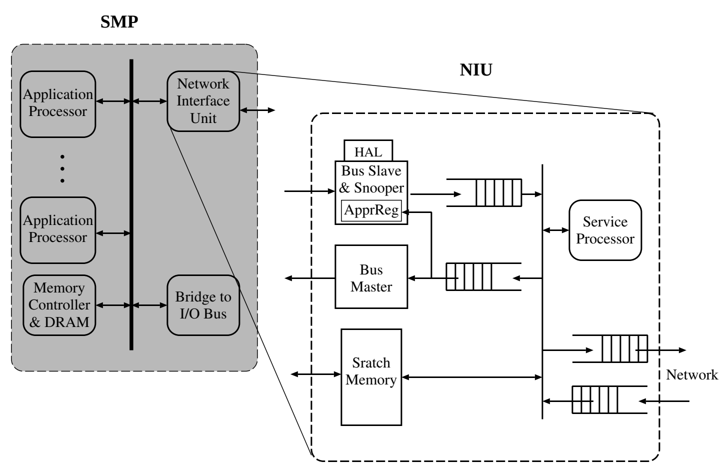

Figure 3: A minimal NIU Design. It interfaces with the SMP bus both as a slave and master on one side and with the network on the other side. The embedded service processor does all the work and provides flexibility.

ing message a Parallel Job Identifier (PJID) which is subsequently verified at the destination[25]. In particular, it allows non-monolithic and non-symmetric communication domains to be set up over the same physical network. The former recognizes that not all parties in a communication network are equal: some entities are more trusted and are allowed to communicate with both trusted and untrusted parties.

3.3 An Overview of the NIU Architecture

With these assumptions, our minimal or base NIU design consists of four parts (see Figure 3): (i) an embedded service processor (sP); (ii) a pair of transmit and receive queues that interface to the interconnection network; (iii) an interface to the SMP bus, consisting of master, slave, and snoop units, along with a special "approval" register and *HAL* bits for S-COMA CCDSM support; and (iv) a scratch memory or buffers accessible by all three major components.

The NIU presents a memory-mapped interface with memory operations on specific regions of memory invoking different operations (see Figure 4). These memory operations are interpreted by the sP, which performs most of the functionality of the NIU. Messages to and from the network are mediated by the sP forming and deciphering network packets and enforcing protection. The hardware network transmit and receive queues are necessary since the sP speed may not match the

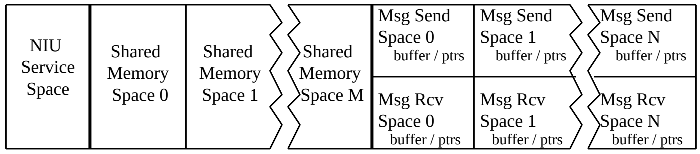

Figure 4: The virtual memory is divided into various spaces each with different semantics. The operating system, through the page table mechanism, can control access to these spaces. The implementation may require the OS to conform to certain mapping restrictions of these spaces.

Type	Size (words)	aP Ovrhd	Lat-ency	Band-width	Comments
Express	1+	low	low	low	Synchronization, Bookkeeping
Tagon	10+	low	med	med	Cache Coherent Shared Memory Protocols
Basic	100+	med	med	med	Generic message passing
DMA	1000+	low	high	high	Page Migration, Coarse Grained Apps

Figure 5: Each StarT-Voyager message type has different fields and is optimized for its own usage pattern in terms of processor overhead, latency, and bandwidth.

network's.

The interface to the SMP bus is more involved. Via the bus master, the sP can issue commands to move data between the scratch memory and the SMP's DRAM or it can invalidate cache lines in the SMP processors. The bus slave captures memory accesses from any application processor (aP) in the SMP and eventually satisfies them using the scratch memory to buffer data. Unlike I/O bus interfaces, the NIU can maintain cache coherence with the SMP processors.

Since we retry operations to avoid committing, some mechanism is required to avoid sending the same request to the sP repeatedly. Our solution is to employ an *Approval register*. When a transaction is to be processed by the NIU, it is recorded in the approval register and requested to be retried. At the same time, the NIU begins processing the transaction. The transaction will continue to be retried by the SMP while being processed by the sP. When the sP completes its actions, the bus slave then commits and satisfies the transaction.

When the slave unit services a cache line write action, it stores the data in the next free location of the scratch memory. It then informs the sP of the bus address and action. The sP knows implicitly where to find the cache line. Similarly, when the slave unit services a cache line read action, the sP will inform the slave where the data is located in the scratch memory.

These are all the mechanisms needed to meet our requirements. The distributed shared memory is implemented on top of message passing, and so the latter is described first.

3.4 Meeting the Message Passing Requirements

Latency and processor overhead times that are proportional to the message length can be achieved through a spectrum of message semantics, two of which, Basic and DMA, are fairly standard while two others, Express and Tag-On, are novel. Each is optimized for its own usage pattern and has its own advantages, see Figure 5. Since Basic messages form the foundation upon which the remaining three are build, we begin with its description.

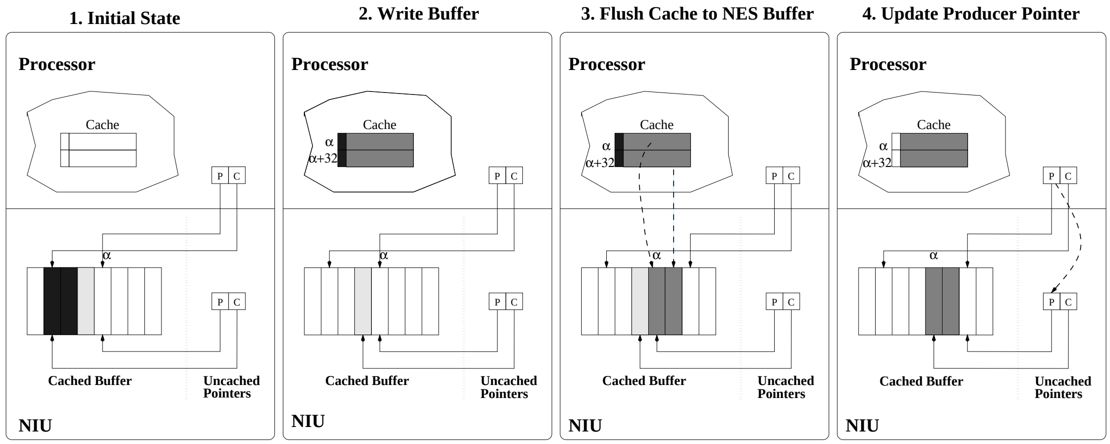

Figure 6: Sending a Basic Message. From software's point-of-view, a message is sent as follows. The consumer and producer pointers are compared to ensure there is free buffer space (Step 1). A message consisting of a destination, size, and data, is stored into the next free buffer location, which is cached (Step 2). Finally, the producer pointer is then updated usually after a "sync" or "fence" command to ensure that the update occurs after the message store completes (Step 3). A receive operation occurs in an analogous fashion, with an update to the producer pointer signalling that the message has already been read.

Basic Messages: The Basic message mechanism provides support for variable length messages up to a few cache-lines long. It transmits and receives through FIFO buffers which are emulated by a continuous region of cachable memory and producer/consumer pointers which are uncached. Figure 6 shows the software steps involved for transmitting a message.

The NIU bus slave services transactions to the pointer regions. Buffers are allocated to DRAM. An update to a producer pointer of the TxQ causes the sP to begin preparing to launch the message. The sP issues bus transactions to fetch the buffer data from the aP's cache or DRAM. After checking for message validity, the sP formats a network packet for the message information into the transmit queue and launches it into the network. At the receiving site, the sP removes and reformats the packet, places it into DRAM and updates the consumer pointer. Queue overflow is handled by writing the data in some scratch DRAM space and sending a high priority message to the source advising it to hold back forwarding additional messages.

DMA messages: The DMA message mechanism provides support for memory-to-memory, long messages. Latency is sacrificed for bandwidth. The source sP communicates with the destination sP to agree on validity and DRAM locations. The source sP issues a series of NIU bus master commands to fetch data from the local DRAM, formats a network packet, and then commands the network transmit queue to launch the message. Reception handling is analogous.

Express messages: The Express message mechanism provides support for very small messages (a few bits more than a word) which originate and arrive in registers. Bandwidth is sacrificed for low processor overhead and low latency. Express messages provide the ability to atomically launch a message in a multi-threaded environment *without* synchronization instructions, dramatically reducing the cost of sharing network queues between multiple threads and multiple SMP processors. To increase the usefulness of Express messages, a tag field in addition to a data word is communicated.

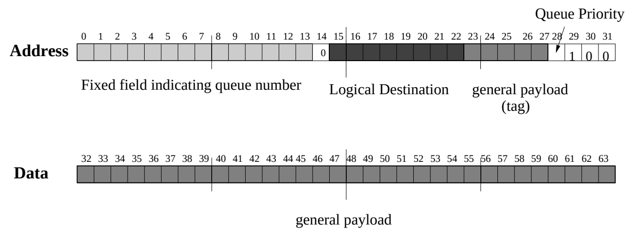

Figure 7: Express Message Transmit Format. A destination, priority, five bit tag field, plus 32 bits of data are specified with a single uncached write operation.

An entire message can be sent by using a single inherently atomic uncached write as explained below.

An Express message can be thought of as a Basic message in which the buffer plus producer/consumer pointers emulating a FIFO are replaced with a real hardware FIFO. In reality, the producer pointer is automatically updated when the message body is written. The user repeatedly writes into the same address and, if the NIU is fast enough to avoid the message buffer overflows, he is freed from having to manage the buffer pointers. Even if the NIU is not fast enough, the user can reduce the pointer checking overhead by periodically determing the amount of buffer space available and sending fewer messages than that.

The single uncached write operation is captured by the NIU bus slave and the address and data lines are forwarded to the sP. The address carries much information about the message, specifying the particular Express message queue, the destination, and the tag. The data word is also included in the message. A large address range is allocated to each Express message queue – the size of the range is a function of the number of destinations and the number of tag bits (See Figure 7).

Message reception works in a similar way. A load instruction to an Express RxQ address, automatically causes the consumer pointer to be incremented and the message source, tag, data to be returned. An empty queue causes a special (configurable) empty message to be returned. Microprocessors that support a doubleword read with a single atomic load instructions are also threadsafe for message reception. In other machines, two load instructions are required.

Tag-On messages: The Tag-On message mechanism extends the Express message mechanism to allow additional data to be appended to an out-going message. Additional address bits are used to specify an offset into the message space that contains one or two cache lines of data to be appended to the express message. Once again, the sP may have to reclaim these cache lines for launching the packet into the network. The decoupling of the header from the message data, allows message data to be located in non-contiguous addresses. This is useful in coherence protocols when sending a cache-line of data between sites as well as for multi-casting.

Multiple Message Regions and OnePoll

Finally we note that *multiple message queues* can be supported just by allocating different ranges of physical memory addresses to different queues. The appropriate semantics for each type of queue is implemented under the control of the sP. Some control over the virtual to physical page mapping in the aP is required. In particular, when a process is assigned a transmit or receive queue and allocated a region of virtual memory, the OS must ensure that this region is mapped to the appropriate region of physical memory with the correct semantics.

Often there is need to poll several queues, and service messages from them according to some priority. This can be a fairly expensive operation without hardware support. A small extension of the Express message receive mechanism, the *OnePoll* mechanism, is provided for this purpose. Specifically, an application registers a set of queues in priority order to the OnePoll unit. A load from the "one poll" address causes the sP to examine each queue for non-empty status. As soon as one is found, it is used to supply the data to the load. If all are empty, a special programmable empty message is enabled. Basic messages can also be included in the OnePoll, by returning a special Express message that identifies the chosen Basic message queue that is non-empty, and the values of its status pointers.

3.5 Meeting the Shared Memory Requirements

The cache coherent distributed shared memory implementation requires cooperation with the local operating system. When the OS maps virtual memory addresses corresponding to system-wide global shared memory addresses to physical page frames, the NIU must be informed of the correspondence in order to maintain its inverse map from physical memory to system-wide memory. Some frames correspond to NIU slave space, while others correspond to DRAM locations that are snooped by the NIU.

A large, sparsely accessed shared memory is best implemented as NUMA, which is mapped to slave space. A cache line miss or eviction by an aP is serviced by the sP. The sP may access local DRAM if it is the home site or make a request to a remote site for a cache line to satisfy a transaction.

When there is sufficient DRAM to support local copies of active shared memory pages, it is efficient to treat the DRAM as a level 3 cache using an S-COMA implementation [30]. The coherence protocol is divided into a hardware part that specifies the agent responsible for the transaction and a firmware part that executes on the sP. For each cache line, the NIU maintains its state, *shared*, *exlusive*, *invalid*, or *other protocol*. The latter cache-line state satisfies the requirement for multiple protocols in which all transactions are captured and forwarded to the sP for processing. A function of the state and bus transaction determines if the memory controller or the NIU is to respond to the transaction.

4 Enhancements

In the base NIU design the sP is involved in almost every action. Two performance problems can arise: (i) the latency of going through the sP may be unacceptable for latency sensitive operations; (ii) as the network becomes faster and the number of aP's per site increases, the sP *occupancy* becomes a serious performance bottleneck. Our solution is to use specialized hardware functional units within the NIU to handle common case scenarios, taking the sP "out of the loop". We describe how to eliminate the sP from most message passing operations, then discuss some further hardware enhancements to ease system-level issues as well as to support multiple cache coherence protocols.

4.1 Improved Message Passing Performance

In order to handle Basic messages without invoking the sP, hardware assistance is needed for three functions: (i) FIFO message queue management; (ii) message formatting; and (iii) queue name translation.

First, each Basic message TxQ and RxQ buffer are mapped to some known place in the scratch pad memory. Hardware support for FIFO queues then consists of registers and logic for manipulating the producer and consumer pointers, allowing the bus slave unit to satisfy operations on the message queues. The addition of transmit and receive hardware units takes care of reformatting messages in the hardware TxQ's for transmission into the network, and steering of the receive messages into RxQ's. When protection is enabled, queue name translations occur during transmission via hardware table lookup. To support multiple hardware queues, TxQ and RxQ states are organized into "queue state register files". The sP is still used to handle exceptional situations. For example, if the validity check of the destination fails, or a message arrives for a full RxQ, the sP is invoked to take the appropriate action.

Express and Tag-On messages are implemented on top of the Basic implementation, with hardware "wrappers". The slave unit's processing of an Express transmits uncached write automatically updates the producer or consumer pointer, and then invokes the Basic engine.

Three composable, simple hardware units implements DMA: a reader, a sender and a receiver. The reader performs block bus operations and the sender performs block transmits with data and bus operations to execute on the receiving site. They can be coupled to run in a pipelined fashion, moving data through a queue in scratch memory or used separately to provide block bus operations or multicast of blocks of data. The receiver decomposes the message into data and bus op parts, and issues the bus operations with the provided data. The DMA receiver uses remotely settable counters to determine when to notify its sP of completion.

4.2 Resident and Non-resident Message Spaces

In order to support a large number of message queues without the expense of a large scratch memory, the design should retain the ability for the sP to service message passing operations. We refer to queues that have dedicated hardware state as *Resident* and all other queues as *Non-resident.* Because both are accessed via memory-mapped interface, application code is not directly aware of whether its queue is mapped to NIU slave space (Resident) or to DRAM (Non-resident). This mapping can thus be changed dynamically and transparently by the system software. In essence, Resident queues are used as a firmware controlled message queue cache.

All non-resident queues share a single hardware queue with the overflow queue and are handled by the sP. Other required hardware includes a lookup table to allow dynamic, local mapping of receive queues into Resident resources. Packet arrival results in a table lookup (similar to cache-tag matching) to determine if the packet's destination queue is one of those mapped to the hardware. If not, or if that destination queue is already full, the message is routed via a special queue to the sP for processing as explained in Section 3.

Since all queues are active, where resident or not, more flexible scheduling policies are possible. Swapping out a parallel process requires no global coordination to ensure correctness. The OS can locally decide the best use of the resident message buffer space. This is in contrast to MPP's such as the CM-5[18] which supports only a single set of network queues, forcing the queues as well as the network state to be entirely swapped out at the time of context switch. The liberation from globally synchronous gang scheduling opens up the possibility of implementing novel co-scheduling approaches[29].

4.3 Multiple CCDSM Protocol Support

A single cache coherence protocol such as MESI cannot efficiently support all memory access patterns. For example, false sharing can be alleviated by multiple-writer protocols. One way to implement multiple protocols would be to mark such cache lines as special and let the sP handle accesses to them. Such an implementation, however, is likely to perform poorly.

Cache Line State	Bus Transaction Type: Read		Read Atomic		Rd w/ Modify		Write	
	Cntl	NIU	Cntl	NIU	Cntl	NIU	Cntl	NIU
Invalid	no	yes	no	yes	no	yes	no	yes
Exclusive	yes	no	yes	no	yes	no	yes	no
Shared	yes	no	yes	yes	no	yes	no	yes
Update Exclusive	yes	no	yes	no	yes	no	yes	yes
Update Shared	yes	no	yes	yes	no	yes	yes	yes

Figure 8: Each cache line has up to eight states. The table has been extended to provide for the simultaneous support for multiple protocols on a cache line basis.

HAL bits allow hardware to automatically handle bus operations that do not require additional action, without intervention by the sP. By providing additional HAL bits, we can directly support multiple protocols (which may wind up being a single large protocol). A possible encoding for three HAL bits per cache line is shown in Figure 8. The first three states can behave with the usual semantics while the remaining five states can have a different action mapping for each transaction type. With the additional bits and without further modifications, however, the sP is the only entity capable of changing the HAL bits.

5 StarT-Voyager Implementation

The high-level design outlined in the previous two sections is implemented in the StarT-Voyager system. Each site is a desktop-class, commercial, dual-PowerPC 604e SMP. The processor card contains one 604e processor and an in-line L2 cache. The StarT-Voyager system replaces one of the two processor cards in each SMP site with an NIU card, making each site into a uniprocessor system[1]. Each site runs AIX extended to handle Voyager address mapping requirements and a scheduler for parallel job coordination.

Each NIU card is attached to the Arctic network, a fat-tree network constructed with Arctic router chips[5] which were designed as part of this project. The Arctic chip is a 4x4 packet-switched router which supports variable length packets, with two priority levels. It does virtual-cut-through routing and has support for maintaining FIFO ordering under user directive. Each link is 16 bit-wide and runs at 80MHz to achieve 160 MBytes/sec bandwidth. With two links, one in each direction, between each NIU card and the network, each site has a peak network communication bandwidth of 320 MBytes/sec.

Figure 9 shows the organization of a StarT-Voyager site, with details of the NIU. The sP is organized with its own sub-system comprising of off-the-shelf parts: a PowerPC 604e processor, a memory controller, and DRAM. Design complexity is further reduced by the symmetry of the organization: from the NIU Core's perspective, the sP subsystem is almost identical to the aP complex. Both have access to the same message passing mechanisms, but each has its own set of resources. Two banks of dual-ported SRAM provide scratch memory. The clsSRAM provides three HAL bits per cache line (Section 4.3) and is accessed by the aP snoop device to implement CCDSM protocols.

The NIUCtrl, a small ASIC, implements all the hardware control functionality and the enhanced hardware functionality (Section 4) and contains all the control state, including the producer/consumer message buffer pointers. A total of 512 queues per site are supported for all

[1]We desired multiple processors per site, but could not find an adequate PowerPC 604 motherboard for our purposes.

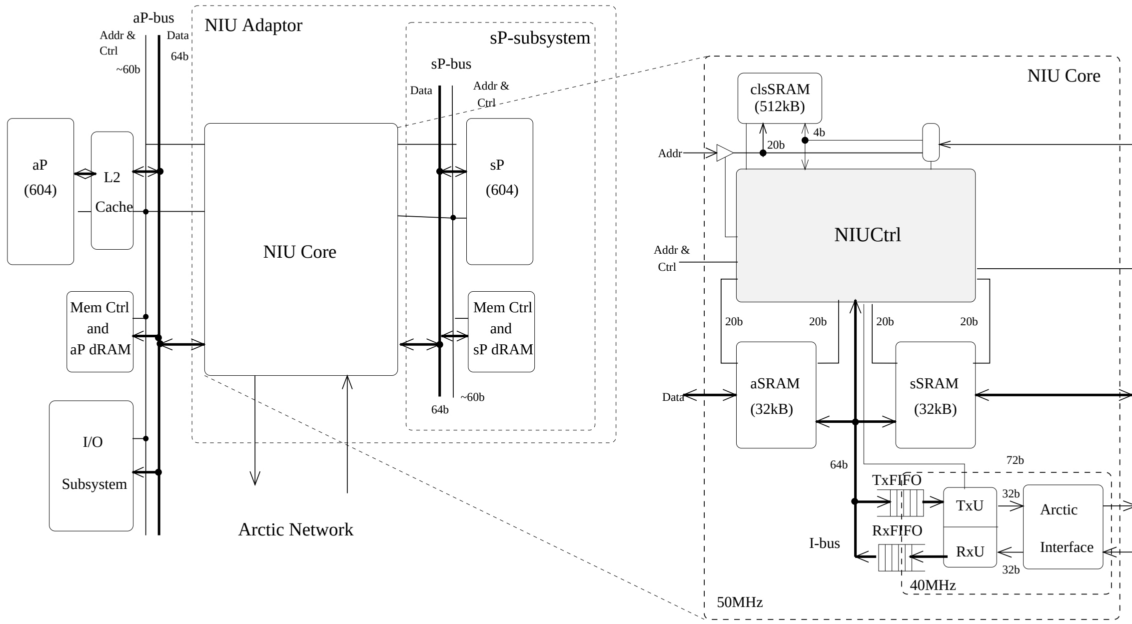

Figure 9: A StarT-Voyager site.

communication. This number is dictated largely by our desire to encode the destination RxQ's in small enough space to fit the Arctic message headers. Express messages consist of 32 bits of data, 5 bits of tag, 9 bits for RxQ specifier, and 1 bit for priority. Tag-On messages specify either one or two cache lines to be appended as well. Scratch memory is organized to hold 16 pairs of resident transmit and receive queues (8 Express / Tag-On and 8 Basic).

All messages in the NIU core flow over the the I-bus, which is 64 bit-wide and runs at 50MHz. This roughly matches both the bandwidth of the PowerPC 60X bus which runs at the same speed and has the same data path width, and the 320 MBytes/s aggregate bandwidth between an NIU and the Arctic network. Outgoing messages undergo destination address translation (Section 4) in the Transmit Unit (TxU), which also computes and appends a CRC to the message packet. This CRC is checked at each Arctic stage, and again in the Receive Unit (RxU). The TxU also formats messages into Arctic packets. The Arctic Interface handles low-level signaling and analog electrical conversions (TTL to ECL) between the NIU and the Arctic Network itself.

The Arctic network has been operational since September 1997. The NIU card is expected to be fabricated by January 1998. Since the OS extensions and cache coherence protocols have been under development for some time, we expect the whole system to be operational in the spring of 1998.

6 Projected Performance

Estimated message passing performance of StarT-Voyager is measured using the three LogP[8] metrics. The numbers are obtained with micro-benchmarks running on a simulator which models the StarT-Voyager NIU with sythesizable Verilog code.

- **Processor overhead:** This is the application software overhead for sending and receiving the message.

- **Latency:** We measure the one-way end-to-end latency, including the software overhead for sending and receiving messages, and 2 hops on the network (one network switch).

- **Gap/bandwidth:** Gap measures the sustainable interval between messages. The inverse of gap multiplied by message size gives bandwidth.

Two sets of numbers are presented, when each message queue is only used by a single thread, and when each message queue is shared by multiple, dynamically forked short threads.

Graphs in Figure 10 plot the various performance metric as the message payload, in units of 4Byte words, is increased. We focus on the performance of small to medium size messages as these are the more challenging cases. Large block transfers are efficiently handled by our hardware supported DMA, where performance is limited by the bandwidth of the memory bus and network and not the NIU.

The performance numbers shows that Reclaim support (*i.e.* NIU explicitly flushes cache lines) for Tag-On and Basic messages is always superior to having the processor performance explicit flushes. We had expected better processor overhead but worse latency with Reclaim support, however the high cost of explicit software Flush makes Reclaim competitive even for latency.

For very small messages, Express messages are superior. The cross-over point from Express to Tag-On messages is about two words and is lower than we had expected. This is due to the inefficiency of uncached accesses compared to cache-line burst transfers. Nevertheless, in certain applications, such as CCDSM protocols, single-word messages are so common, they warrant special consideration.

When multiple threads share a message queue, correct operation of a Basic message queue requires locking the queue when sending or receiving a message, which shows up as increased processor overhead and latency. Owing to their inherent thread-safe design, Express and Tag-On messages queues do not require locking and incur no additional penalty. With little penalty in the single-threaded scenario and significantly better performance in the multi-threaded case, Tag-On is a better mechanism. Basic message does have some benefits which are not reflected in the performance numbers. The use of pointers allows aggregation, so that when several Basic messages are sent out or received together, some savings in handshake between application code and the NIU is possible.

The absolute performance of the message Passing mechanism in StarT-Voyager, with message latency of between 1 to 5 μs, bandwidth of over 100 MBytes/sec with messages of only 88 Bytes is superior to all but the very best supercomputers today[20, 22, 21, 9, 6, 12, 13, 24].

Shared Memory Performance

Shared memory performance is highly dependent on cache miss ratios. By implementing S-COMA in hardware with HAL bits, StarT-Voyager has the capability of having a 64MB L3 cache, which will drastically reduce capacity and conflict misses in many programs.

To give an idea of the cost of across network cache miss processing, Figure 11 gives the latency breakdown for servicing a read cache-miss which finds data in the clean state at the home site. The overall latency is about 15 times the cache-miss latency to local DRAM. Since the sP comes into play only in cases where the data is not in the huge L3 cache, the extra latency may not have deep impact on the overall performance.

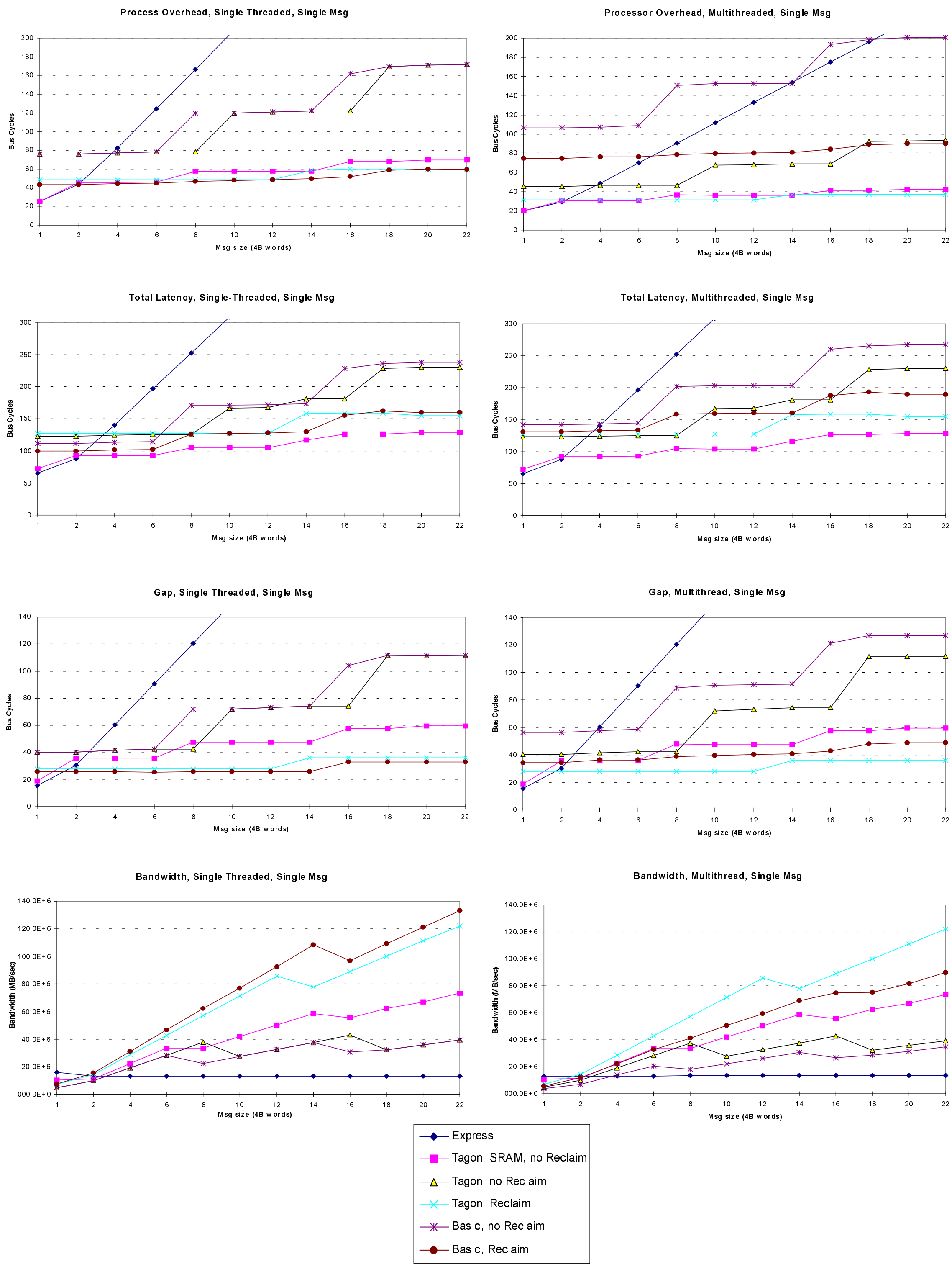

Figure 10: Processor overhead and latency in a multi-threading environment.

Action	Latency (bus cycles)
aP bus op capture	7
Local sP receives request	15
sP sends request to Home	12
Network Latency (2 hops)	26
Home sP receives request	15
Home sP gathers data and sends response	46
Network Latency	26
Local sP receives reply	15
Local sP satisfy bus request	32
Total	194

Figure 11: A breakdown of a shared memory read which is clean at a remote home site.

7 Conclusion

We have presented a design for an open, scalable cluster system supporting both shared memory and message passing. For economic reasons, scalable parallel machines in the near future are likely to be constructed this way. The main contributions of this work are:

- A communication architecture which efficiently layers shared memory implementation on a high performance, multi-user message passing substrate.
- An integrated, configurable hardware and firmware CCDSM implementation platform which permits multiple user-defined coherence protocol to be operational simultaneously.
- A memory-bus based message-passing interface supporting several data-transfer mechanisms, each suitable for different message granularity, data location, and programming model.
- Use of inherently atomic single uncached reads and/or writes at the "commit points" of message passing operations to achieve a thread-safe interface.
- Use of a hardware-enforced, private, virtual message destination name space as a general, flexible means for enforcing protection and facilitating job migration in a multi-tasking environment.
- Direct support for a large number of simultaneously active message queues, implemented in a high performance and cost-effective way with Resident and Non-resident queue resources.
- A low-overhead one-poll mechanism that returns the highest priority message from a set of message queues.

The biggest difference between our design and other related work is our insistent that protection, high performance, and flexibility for both shared memory and message passing must all be addressed in one single design. Most earlier designs have less ambitious requirements, usually assuming a single communication mechanism is adequate for most parallel applications. Those that are support both shared memory and message passing, *e.g.* the Alewife machine, do not consider system level issues like protection and multitasking.

Our use of a commodity microprocessor to execute coherence protocol code (also used in Typhoon[26]) is controversial. Many feel that global cache-miss latency would be too high [23] to be practical, motivating designers of commercial DSM's such as the Origin 2000 [17] to implement the entire cache protocol in hardware.

This belief, however, may be based on inefficient designs for hardware supporting the protocol processor. For example, requiring the sP to poll for completion of one event before another can be issued will increase sP occupancy, dramatically increase latency and complicate firmware because of the need to continue servicing new requests. Our complete design solves this problem with hardware command queues which accommodate multiple outstanding commands and enforce ordering between related commands. The sP can service a request by issuing a small stream of commands into these queues without waiting for completion of individual command.

Another insight is the realization that the special NIU hardware units are rather general, and are often used in many different communication operations. Potentially, these can be further developed so the collection of special NIU hardware eventually becomes a more general communication processor. This is the subject of future work.

We are currently in the final phases of hardware implementation and expect hardware in January 1998. Once that is up and running, much experimentation and evaluation will be conducted. The final version of this paper will incorporate numbers from those experiments.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2 – 13, 1995.

[2] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 System Architecture. *IBM Systems Journal*, 34(2):152 – 184, 1995.

[3] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142 – 153, Apr. 1994.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet – A gigabit-per-Second Local-Area Network. *IEEE Micro*, Feb. 1995.

[5] G. A. Boughton. Arctic Routing Chip. In *Proceedings of Hot Interconnects II, Stanford, CA*, pages 164 – 173, Aug. 1994.

[6] C.-C. Chang, G. Czajkowski, and T. von Eicken. Design and Performance of Active Messages on the IBM SP-2. (Work at Dept of Computer Science, Cornell University, Ithaca. Available from http://www.cs.cornell.edu/Info/Projects/CAM/ in Jul 96).

[7] Convex Computer Corporation, Richardson, Tx. *Exemplar Architecture*, Nov. 1993.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subrmonian, and T. von Eicken. LogP: Torwards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego*, pages 1–12, 1993.

[9] D. Culler, L. T. Liu, R. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, 1996. (to appear).

[10] R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, pages 12 – 18, Feb. 1996.

[11] J. C. Hoe. Network Interface for Message-Passing Parallel Computation on a Workstation Cluster. In *Proceedings of Hot Interconnects II, Stanford, CA*, pages 154 – 163, Aug. 1994.

[12] J. C. Hoe and M. Ehrlich. StarT-Jr: A Parallel System from Commodity Technology. CSG Memo 384, MIT Laboratory for Computer Science, July 1996.

[13] V. Karamcheti and A. Chien. FM: Fast Messaging on the Cray T3D. (Work at Concurrent Systems Architecture Group, Dept of Computer Science, University of Illinois at Urbana-Champaign. Available from http://www-csag.cs.uiuc.edu/projects/comm/fm.html in Jul 96).

[14] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: a New Dimension for Cray Research. In *Digest of Papers, COMPCON Spring 93, San Francisco, CA*, pages 176 – 182, Feb. 1993.

[15] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Intergrating Message-Passing and Shared-Memory: Early Experience. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego*, pages 54–63, 1993.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chaplin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Il*, Apr. 1994.

[17] J. Laudon and D. Lenoski. The SGI Origin 2000: A CC-NUMA Highly Scalable Server. In *The 24th Annual International Symposium on Computer Architecture Conference Proceedings, Denver, CO*, June 1997.

[18] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[19] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 92 – 103, 1992.

[20] L. T. Liu and D. E. Culler. Measurements of Active Messages Performance on the CM-5. (Work at Dept of Computer Science, University of California at Berkeley. Available from http://now.cs.berkeley.edu/Papers/papers.html in Jul 96), May 1994.

[21] L. T. Liu and D. E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Proceedings of Intel Supercomputer Users Group Conference*, June 1995.

[22] R. P. Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Proceedings of Hot Interconnects II, Stanford, CA*, pages 40 – 58, Aug. 1994.

[23] M. M. Michael, A. K. Nanda, B.-H. Lim, and M. L. Scott. Coherence Controller Architectures for SMP-based CC-NUMA Multiprocessors. In *The 24th Annual International Symposium on Computer Architecture Conference Proceedings, Denver, CO*, June 1997.

[24] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95, San Diego, CA*, 1995.

[25] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. *T: Integrated Building Blocks for Parallel Computing. In *Proceedings of Supercomputing '93, Portland, Oregon*, pages 624–635, Nov. 1993.

[26] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Il*, pages 325 – 336, Apr. 1994.

[27] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA*, pages 174 – 185, Oct. 1996.

[28] Sequent Computer Systems, Inc. *Sequent's NUMA-Q Architecture White Paper.* (Available from http://www.sequent.com/public/solution/numaq/technology/archindex.html in Jul 96,).

[29] P. G. Sobalvarro and W. E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors . In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science Vol. 949*. Springer-Verlag, 1995.

[30] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architecture. In *The 19th Annual International Symposium on Computer Architecture Conference Proceedings*, pages 80–91, 1515 Broadway, New York, NY 10036, 1992. ACM, ACM Press. ISCA '92-Gold Coast, Australia.

[31] C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker. The SP2 High-Performance Switch. *IBM Systems Journal*, 34(2):185 – 204, 1995.