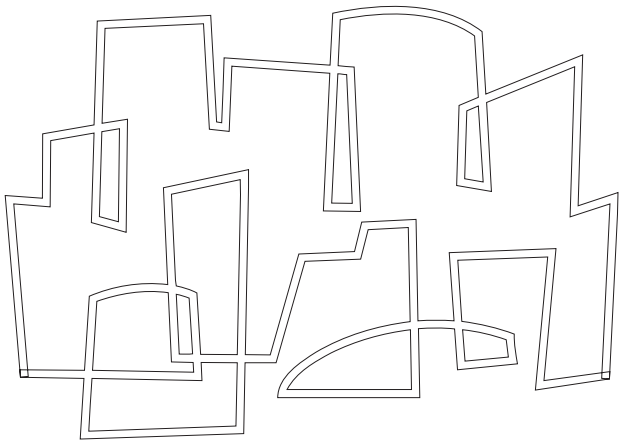
The StarT-Voyager Parallel System

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

In proceedings of International Conference on Parallel Architectures and Compilation Techniques (PAC)

1998, October

Computation Structures Group
Memo 416



LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

The START-VOYAGER Parallel System

Computation Structures Group Memo 416
December 8, 1998

Boon S. Ang, Derek Chiou, Larry Rudolph and Arvind
Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Sqaure, Cambridge, Massachusetts
{**hahaha,derek,rudolph,arvind**}**@lcs.mit.edu**

To appear in the International Conference on Parallel Architectures and Compilation Techniques (PACT '98) October 13-17, 1998, Paris, France

This paper describes research 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.

The StarT-Voyager Parallel System

Boon S. Ang, Derek Chiou, Larry Rudolph and Arvind
Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Sqaure, Cambridge, Massachusetts
{hahaha,derek,rudolph,arvind}@lcs.mit.edu

Abstract

This paper presents the communication architecture of the StarT-Voyager *system, a parallel machine composed of a cluster of unmodified IBM 604e-based SMP's connected via a high speed interconnection network. A custom network interface unit (NIU) plugs into a processor card slot of each SMP, providing a high-performance message passing substrate that supports both fast user-level message passing and cache-line coherent shared memory. The substrate consists of four hardware implemented message passing mechanisms to achieve high performance over a wide spectrum of communication patterns.* StarT-Voyager *also introduces a novel protection scheme which improves upon past designs by not requiring strictly synchronized gang-scheduling, and by allowing system code and multiple user applications to share the network simultaneously without compromising protection nor performance. Performance predictions based on synthesized Verilog code 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 promising.* StarT-Voyager *hardware will be available in September, 1998.*

1. Introduction

Symmetric multiprocessors (SMP's) are rapidly becoming the typical desktop workstation, making clusters of SMP's the future standard of scalable parallel computers. Clustering SMP's presents challenges, not only in terms of performance, but also in terms of deciding which communication mechanisms to support and how to implement them well. System issues, such as enforcing communication protection, its implications for job scheduling, and interaction between communication and job scheduling are other important issues that remain to be studied. The StarT-Voyager system described in this paper is designed to address all of these issues.

Most parallel machines support a single communication mechanism, either message passing or shared memory, and emulate all others[12, 5, 9, 14]. Today's parallel applications, however, utilize a wide range of communication mechanisms, effectively guaranteeing that some communication must be emulated; this can significantly reduce achievable performance. Ideally, a machine should provide direct support for all common communication mechanisms. Furthermore, because the network is currently an expensive part of a parallel system, these mechanisms should share the network. Safe network sharing requires an appropriate protection mechanism to guarantee non-interference between unrelated jobs. Multiple processor SMP nodes present further motivation for having this ability: a single SMP can potentially have multiple parallel jobs running simultaneously on different processors. If user applications are given direct access to message passing, some underlying network sharing protection mechanism has to be in place.

Consisting of cluster of unmodified SMP's connected by a custom NIU to a high speed network, the StarT-Voyager system provides a set of architectural mechanisms designed to efficiently support a wide variety of message passing and shared memory models. A simple but flexible protection model allows multiple communication protection domains to co-exist on the same network.

The biggest difference between the StarT-Voyager design and others is our insistence that protection, high performance, and flexibility for both shared memory and message passing must be addressed together in a single design. Of the few machines that support both shared memory and message passing, *e.g.* the Alewife[1] machine, system-level issues like protection and multitasking were not addressed adequately. Our design is as flexible as Flash [10], but requires no modification to the SMP node hardware. Furthermore, we anticipate occupancy of the firmware engine to be far less of an issue in our design compared to FLASH because hardware implements most communication mechanisms as

well as accesses to data in local DRAM that requires no further remote actions[1].

Commercial DSM's, such as the Origin 2000 [11], implement the entire cache protocol in hardware to reduce remote cache-miss latency [16]. Our design provides hardware to handle common case scenarios and to reduce software overheads to the point that it is feasible to execute coherence protocol code on a commodity microprocessor, as proposed in the Typhoon design[19].

After an initial discussion of the requirements for an NIU that supports both message passing and shared memory systems (Section 2) a minimalist design for such an NIU is presented (Section 3). The actual StarT-Voyager design is given (Section 4), followed by its projected performance, derived from simulation with synthesizable Verilog code (Section 5).

2. Requirements

Our basic requirement is to form a scalable, parallel processing system from a hardware platform consisting of unmodified commercial SMP's and a high speed interconnection network. The system is designed for a wide range of high performance applications, including both technical and commercial computing. Meant as a general-purpose computation platform, the machine's 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 efficiently. These requirements, together with a low development budget, significantly constrains the design. The remainder of this section elaborates on these requirements.

2.1. Shared Memory

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. To minimize conflict and capacity cache misses, it makes sense to use local DRAM as a large cache for global state when there is locality of reference, as is done in S-COMA[20]. When shared memory is used in a sparsely accessed mode, it makes sense to support a NUMA [14] 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. Within each model 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 simultaneously, though on different addresses.

[1] Such data may be unshared memory regions or shared memory cache-lines present in the correct state for the desired access.

2.2. Message Passing

Different parallel programs have different message passing requirements. 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. Consequently, flexible mechanisms for managing, sharing and polling message buffers are required to build an efficient message passing substrate.

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, a spectrum of message types needs to be supported so that end-to-end transmission time as well as processor overhead time is proportional to the message size. We identify three message types:

Short: Data for short messages consists of a word or two and are both launched from and received to registers. Latency is of primary importance.

Medium: Data for medium messages consists of a cache-line or two and are both launched from and received to the cache. Latency and bandwidth are both concerns.

Long: Data for long messages consist of a page or several pages and are launched from and received to memory. Bandwidth is the primary concern. A long message communication is often accompanied by shorter message transmissions indicating when the whole message body is ready for transmission and when it has been successfully received.

2.3. System Requirements

A multi-user environment must provide protection mechanisms that ensure 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*. The job is not scheduled until the system has been drained of all messages belonging to the previous parallel job. A common approach to provide protection between

multiple message streams is to attach a Parallel Job Identifier (PJID), that corresponds to the gang-scheduled parallel job currently running, to every outgoing message. The PJID is subsequently verified at the destination[18].

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. It is preferable to provide non-monolithic and non-symmetric communication domains to be set up over the same physical network. The need for more sophisticated protection mechanisms is especially acute in a system that supports both shared memory and message passing.

No current distributed parallel system 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. In some cases, 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. 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 functionally supports the full set of capabilities. 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 the SMP memory bus. We begin with our assumptions.

3.1. Hardware Assumptions

Our NIU is designed for a memory bus that supports a snoopy cache-coherence protocol. It relies on the provision in snoopy bus protocols for snoopers to delay or retry bus operations. Our design further assumes that the protocol supports *intervention*, *i.e.*, a snooping device can dynamically take over the responsibility of satisfying a memory request from the primary slave device (typically main memory).

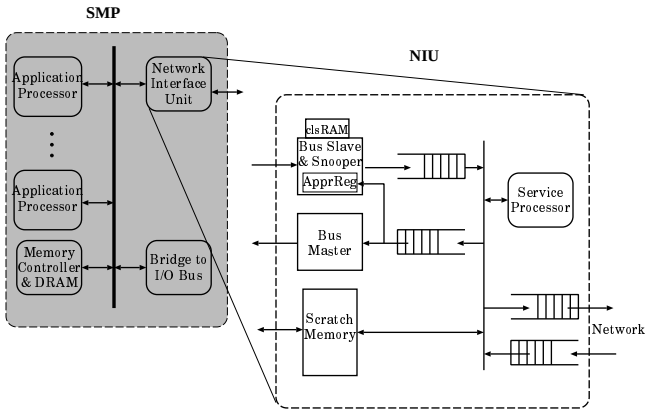


Figure 1. A minimal NIU Design. It interfaces with the SMP bus both as a slave and master on one side and with the network on the other side. The embedded service processor does all the work and provides flexibility.

To achieve reasonable performance, the interconnection network should be a dedicated high bandwidth, low latency network, what is referred to as a system area network (SAN). In particular, we assume that the network provides reliable transmission, and supports at least two message priorities to avoid deadlocks in cache coherence protocols. A FIFO network is convenient for cache coherence protocols, though not an absolute necessity. Assuming such appropriate interconnection network and standard SMP's, the rest of this section outlines the NIU features.

3.2. An Overview of the NIU Architecture

Our minimal or base NIU design consists of four parts (see Figure 1): (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 (ApprReg) and cache-line state *CLS* bits for S-COMA cache coherent distributed shared memory 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 2). 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

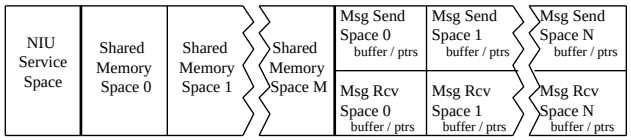


Figure 2. The physical address space is divided into various regions each with different semantics. The operating system, through the page table mechanism, can control access to these spaces.

transmit and receive queues are necessary to meet real-time requirements: the sP speed may not match the 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 operations emanating from an aP are retried by the NIU to give it time before 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 handled by the NIU, it is recorded in the approval register and the transaction is retried. At the same time, the sP begins processing the transaction, such as fetching data from another SMP node. The pending transaction is continually retried until 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 written data in the scratch memory. It then informs the sP of the bus transaction type and address. The sP takes over further processing of the bus transaction, which may include sending the data to some remote node. Similarly, when the slave unit services a cache line read action, the sP informs the slave where the data is located in the scratch memory.

These are all the mechanisms needed to meet our requirements. We now describe how these mechanisms are used to implement message passing and shared memory.

3.3. Meeting the Message Passing Requirements

Traditionally, message passing incurs a fairly high per-message fixed overhead, making it necessary to aggregate communication into large messages. In our design, we achieve latency and processor overhead times that are proportional to the message length through a spectrum of message passing mechanisms[3], two of which, Basic (supporting medium message sizes) and DMA (supporting long message sizes), are fairly standard while two others, Express (supporting short message sizes) and Tag-On (supporting medium message sizes), are novel. Each is optimized for particular but not uncommon usage pattern. Since Basic messages form the foundation upon which the remaining three are build, we begin with its description.

Basic Messages: The Basic message mechanism provides support for variable length messages up to a few cache-lines long. It transmits and receives through circular buffers implemented by a continuous region of cached memory and uncached producer/consumer pointers.

Transactions to the pointer regions are serviced by the NIU bus slave. An update to a producer pointer of the transmit queue (TxQ) causes the sP to launch the message. The sP issues aP bus operations via the bus master module 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, that sP polls for incoming messages, removes the message from the network FIFO and reformats before writing it into DRAM via the bus master. It then updates the receive queue producer pointer. Receive 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 any additional messages.

DMA messages: The DMA message mechanism provides support for memory-to-memory, long messages. Latency is traded for bandwidth. The source sP communicates with the destination sP to translate the addresses (user application makes requests with virtual addresses which has to be translated into physical addresses) and check access permissions. The source sP fetches data from the aP DRAM using bus master operations, formats a network packet, and then commands the network transmit queue to launch the message. This process must be repeated until all data has been transmitted. Reception handling is analogous.

Express messages: The Express message mechanism provides support for very small messages (several bits more than a word) which originate from and arrive in registers. Bandwidth is sacrificed for low processor overhead and low latency. The Express message mechanisms combines message composition and transmit into a single memory operation, and message receive into a single memory operation. Since only a single operation is performed for each transmit or receive, atomicity is achieved without synchronization operations, allowing multiple threads to share the same message queues without synchronization adding significantly to overhead.

Uncached writes and uncached reads are used to implement Express messages transmit and receive respectively. Uncached operations appear on the bus every time they are executed, avoiding the need for the NIU to poll transmit queues and write to receive queues. By using an address range rather than a specific single address as the transmit address, additional information, such as the message destination and a small amount of data, can be encoded into the address space, allowing the uncached operation to convey information beyond the data word. Express message queues look like hardware FIFO's that are stored into with special addresses to transmit and read from with special addresses to receive.

The single uncached write transmit operation is captured by the NIU bus slave and the address and data information are forwarded to the sP for reformatting into a network packet. Message reception works in a similar way, with the sP performing the reverse formatting. The sP also manages the receive queue FIFO; in response to a load instruction to an Express RxQ address, the sP returns the received message at the front of the FIFO. A receive from an empty queue causes a special (software programmable) empty message to be returned. Microprocessors that support a double-word read with a single atomic load instructions are also thread-safe for message reception. In other machines, two load instructions are required.

Tag-On messages: The Tag-On message mechanism extends Basic and Express message mechanisms to append additional data to an out-going message. The location of that data is specified as an offset from a base address, so that a small bit-field in the header (among the address bits for Express messages) is sufficient for specifying it. When transmitting a message, the sP examines the header to determine if Tag-On is used; if so, it reads that data from DRAM at the appropriate time using the bus master. Tag-On allows message data to be located in non-contiguous addresses; it is useful in coherence protocols when sending a cache-line of data between sites as well as for multi-casting.

3.3.1. Multiple Message Regions and OnePoll

Multiple message queues are supported 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. When a process is assigned a transmit or receive queue which is mapped into a process's virtual address space, the OS must ensure that this region is mapped to the physical memory region with the correct semantics.

There is often the need to poll several queues, and service messages from them according to some priority. This can be a fairly expensive operation as each queue needs to be polled separately. 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 in the priority order. When one is found, it is used to supply the data to the load. The sP can insert information into the returned message that indicates which queue the message came from if necessary. If all are empty, a special programmable empty message is returned. Basic message queues can also be included in the OnePoll. When the highest priority non-empty queue is a basic message queue, a special Express message that identifies the non-empty Basic message queue and its status pointers is returned.

TxQ	Logical Dest 0	Logical Dest 1	...
0	$\langle \text{site}_A, \text{RxQ}_\beta, \text{source}_i \rangle$	$\langle \text{site}_B, \text{RxQ}_\delta, \text{source}_j \rangle$	...
1	$\langle \text{site}_A, \text{RxQ}_\theta, \text{source}_l \rangle$	$\langle \text{site}_E, \text{RxQ}_\iota, \text{source}_m \rangle$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	$\langle \text{site}_G, \text{RxQ}_\nu, \text{source}_o \rangle$	$\langle \text{site}_H, \text{RxQ}_\pi, \text{source}_p \rangle$	...

Figure 3. Translation Table: each row contains the logical destination to ⟨site, RxQ, source⟩ mapping for a transmit queue.

3.3.2 Protection

To provide protection, flexibility, and efficient use of resources, message passing communication is implemented with operations on *virtual transmit* and *receive queues*. When a sender enqueues a message into a local transmit queue, it specifies a *virtual* destination receive queues which is subjected to translation with information from a translation table. As shown in Figure 3, this translation is a function of both the transmit queue, and the destination name; effectively, each transmit queue has its own translation context. The sP performs the translation, and only trusted code is allowed to modify the translation table.

Similar to the case in virtual memory implementation, the underlying NIU places no restrictions on which transmit queues are allowed to send messages to a particular receive queue. Instead, the protection is done at name translation: certain receive queues just are not addressable from certain transmit queues. This scheme opens up possibilities for more unusual communication domains as illustrated in Figure 4. Aside from enforcing protection, virtualization of message queue names also enables transparent migration of virtual message queues and the processes that use the queues.

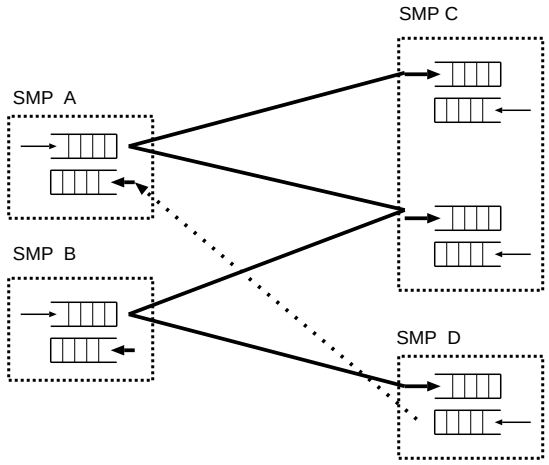


Figure 4. Messages can be sent from TxQ's to RxQ's without any a priori restrictions. A TxQ at SMP A can send to two different RxQ's located at SMP C, and a TxQ at SMP B can also send to two different RxQ's but located at separate SMPs. Notice that the bottom TxQ (at SMP D) can send to an RxQ at SMP A, although the reverse is not true.

3.4. Meeting the Shared Memory Requirements

As with message passing, distributed shared memory also requires operating system support to map virtual pages that are shared to the correct physical addresses where shared memory is being supported. Such physical address regions include both NIU slave space and DRAM locations that are snooped by the NIU. To support S-COMA style shared memory, the NIU must also know about virtual-to-physical mappings as it needs an inverse map from local physical to system-wide memory addresses.

NUMA shared memory is supported by NIU slave space. The sP treats bus transactions to the slave space as accesses to memory in remote nodes, communicating with sP's on the remote nodes to satisfy the bus transactions. Of course, the sP must also be receiving, processing and responding to requests initiated by other sP's that have similar requests.

S-COMA[20] is supported using snooped DRAM space and additional tag bits (CLS), effectively making that part of DRAM a large L3 cache. 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. The NIU maintains state for each cache line: *shared*, *exclusive*, *invalid*, or *other protocol*. The last 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 a bus transaction.

4. StarT-Voyager Implementation

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 is generally unacceptable; (ii) as the network becomes faster and the number of aP's per site increases, the sP *occupancy* becomes a serious performance bottleneck. In StarT-Voyager, specialized hardware functional units within the NIU handle common case scenarios, taking the sP "out of the loop".

Each StarT-Voyager 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[2]. Each site runs AIX 4.2, 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[4] designed as part of this project. The Arctic chip is a 4x4 packet-switched router which supports variable length packets and two priority levels. It does virtual-cut-through routing and can maintain 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 5 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 604 processor, a memory controller, and DRAM. Design complexity is 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 state bits per cache line and is accessed by the aP snoop device to implement cache coherent distributed shared memory protocols.

The NIUCtrl, a small ASIC, implements all the hardware control functionality and contains all the control state, including the producer/consumer message buffer pointers. A total of 512 queues per site are supported for all communication. This number is dictated largely by our desire to encode the destination RxQ's in small enough space to fit

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

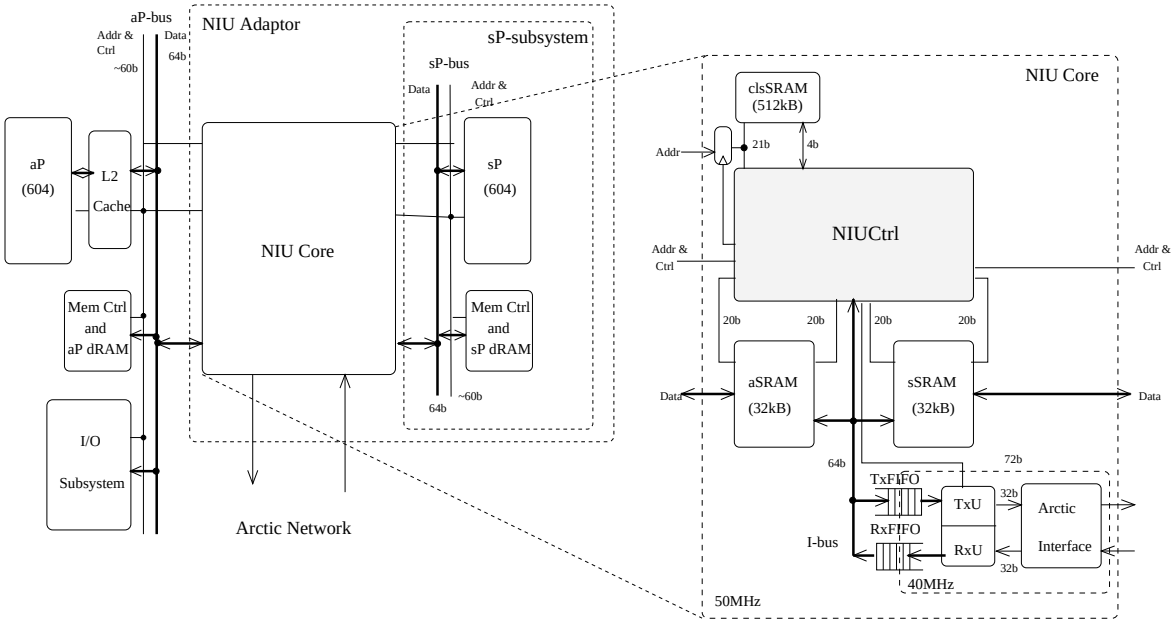


Figure 5. A StarT-Voyager **site.**

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 IBus, which is 64 bit-wide and runs at 35MHz. Outgoing messages undergo destination address translation 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.

All message transmit described in Section 3 is implemented within hardware, dramatically improving performance over the base NIU design. Message receives are handled in hardware if the receive queue is present (only 16 of the 512 receive queues is resident in hardware at any given time) and handled by the sP if not (Non-resident). Shared memory is handled as described in the base NIU.

The StarT-Voyager NIU is actually quite flexible not only because of firmware programmability, but also because parts of the NIUCtrl are implemented with FPGA's. These can be changed quite easily to implement new communication mechanisms and modify existing ones. We describe the micro-architecture details, and its flexibility in much greater detail in another paper [2].

5. Projected Performance

Estimated message passing performance of StarT-Voyager is measured using the three LogP[6] metrics. The numbers are obtained with micro-benchmarks running on a simulator which models the StarT-Voyager NIU with synthesizable 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.

5.1. Message Passing Performance

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 6 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 issue cache-line Flush instructions. 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 the latency metric.

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 cache coherent distributed shared memory protocols, single-word messages are common enough to 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[15, 13, 7, 8, 17].

5.2. Shared Memory Performance

Shared memory performance is highly dependent on cache miss ratios. With its S-COMA support, StarT-Voyager effectively implements a 64MB L3 cache at each site which will drastically reduce capacity and conflict misses in many programs. Hits in this cache are completely handled in hardware and is no slower than a access to main memory in the original machine. Although cache-miss processor is not insignificant as described in the next paragraph, the overall performance will still be very good as long as misses are uncommon.

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 7. A breakdown of the cache-miss latency of a shared memory read which is clean at a remote home site.

To give an idea of the cost of across network cache miss processing, Figure 7 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.

6. Conclusions

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. There are several contributions of this work.

StarT-Voyager provides a memory-bus based message-passing interface supporting several data-transfer mechanisms, each suitable for different message granularity, data location, and programming model. One novel mechanism uses the inherently atomic, single uncached reads and/or writes at the "commit points" of message passing operations to achieve a thread-safe interface.

Our hardware-enforced, private, virtual message destination name space provides a general, flexible means for enforcing protection and facilitating job migration in a multitasking environment. There is 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. To make efficient use of several receive queues, a low-overhead one-poll mechanism is provided that returns the highest priority message from a set of message queues.

In terms of shared memory, StarT-Voyager provides

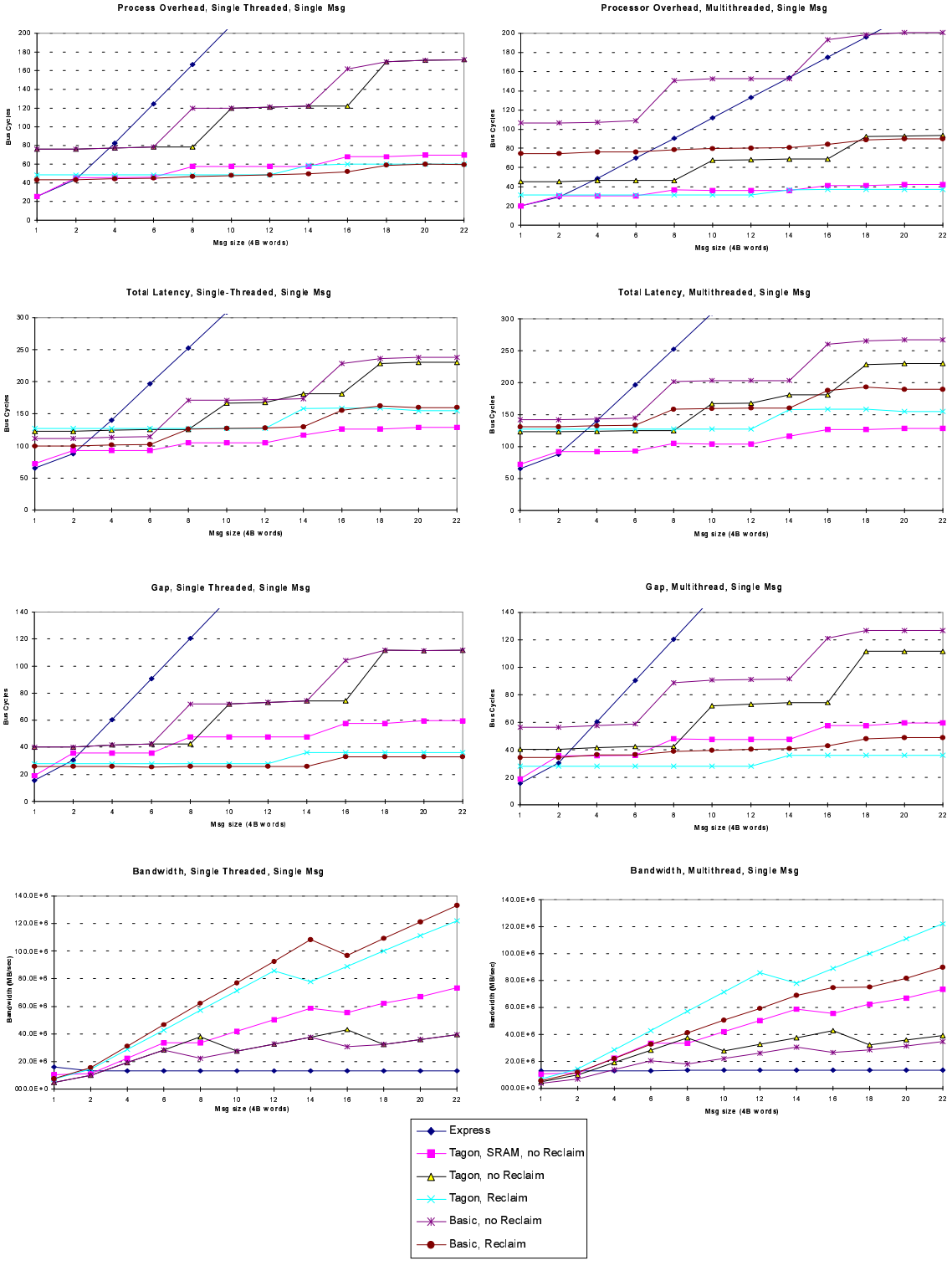


Figure 6. Processor overhead and latency in a multi-threading environment.

both S-COMA and NUMA. Multiple user-defined coherence protocols may be operational simultaneously, due to our integrated, configurable hardware and firmware cache coherent distributed shared memory implementation.

Finally, we note that many of the custom hardware units in the StarT-Voyager NIU 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.

Acknowledgements: This paper describes reserach done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. We would like to acknowledge Daniel L. Rosenband for his excellent implementation of the NIUCtrl ASIC and Mike Ehrlich for his management of the hardware effort. 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.

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 *Conference Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2 – 13, 1995.

[2] B. S. Ang, D. Chiou, D. Rosenband, M. Ehrlich, L. Rudolph, and Arvind. StarT-Voyager: Exploring Scalable SMP Issues. In *Conference Proceedings of Supercomputing 98, Orlando, Florida*, Nov. 1998.

[3] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. Message Passing Support on StarT-Voyager. In *Conference Proceedings of the Fifth International Conference on High Performance Computing, Chennai (Madras), India*, Dec. 1998.

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

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

[6] 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.

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

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

[9] 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.

[10] 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 *Conference Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Il*, Apr. 1994.

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

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

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

[14] T. Lovett and R. Clapp. *Implementation and Performance of a CC-NUMA System*. Sequent Computer Systems, Inc, 1996.

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

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

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

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

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

[20] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architecture. In *Conference Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 80–91. ACM, ACM Press, 1992.