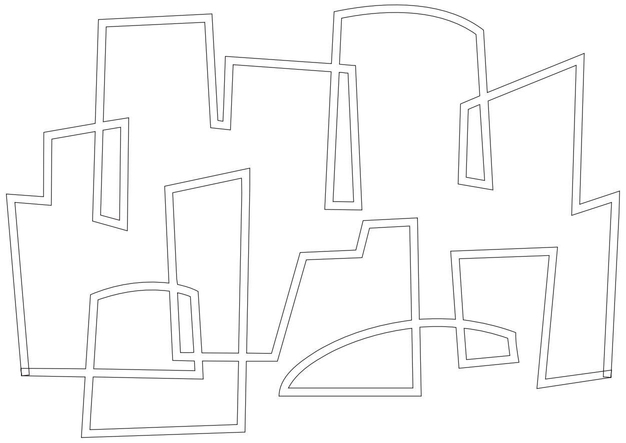
Message Passing Support for Multi-grained, Multi-threading and Multi-tasking Environments

Boon Ang, Derek Chiou, Larry Rudolph, Arvind

1996, November

Computation Structures Group
Memo 394



Message Passing Support for Multi-grained, Multi-threading, and Multi-tasking Environments

Computation Structures Group Memo 394
November 10, 1996

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.

Message Passing Support for Multi-grained, Multi-threading, and Multi-tasking Environments

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

November 10, 1996

Abstract

In order to become generally useful, message passing mechanisms not only need to provide high performance, but also the *three M's*: multi-granularity, multi-threading and multi-processing. In this paper, we discuss these requirements and why they are needed, and describe the message passing mechanisms of StarT-Voyager which provide such functionality while delivering high performance. We discuss how StarT-Voyager implements the three M's and predict its performance. 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. To our knowledge, no architecture supports this entire range of functionality while providing high performance.

1 Introduction

Highly efficient support for interprocessor communication on a general purpose system requires mechanisms that adequately address the three M's of message passing: *multi-granularity, multi-threading and multi-tasking.* While the performance of certain aspects of message passing has improved with the introduction of direct user-level access to network interface[43, 23, 39] and the exploitation of host system capabilities such as coherence on the memory bus[34], architectural support for the three M's remains inadequate.

Parallel and distributed applications exhibit a multitude of communication granularity. At one extreme, a control message that conveys a simple request or an acknowledgment often amounts to no more than a few bytes. At the other extreme, block data transfers of many kilobytes of data are common among coarse-grain parallel applications. In general, the data of the smallest sized messages are transferred between processor registers. As message size increases, caches are the source and desired destination and for really large messages, main memory becomes the most likely repository. The most efficient data transfer mechanism depends on the message size and data location, *e.g.* uncached write and read for very small messages, burst transfer on the memory bus for cache-line sizes, and processor-external DMA for the largest messages.

Multi-threaded execution environments are increasingly common and message passing is a critical part of such environments. Under the common message-passing model of associating a transmit and a receive queue to each process, the basic operations of send and receive each consist of three steps: get a buffer, process, then release the buffer. When simultaneously executing threads share a common message queue, the get buffer and free buffer actions must each be atomic. At the same time, arbitrary interleaving of such actions from different threads should work correctly, and the action or inaction of one thread must not block another thread. Proper design of the hardware network interface can efficiently handle these requirements, which otherwise may require expensive software safe-guards such as locking and message copying.

If the use of high performance message-passing causes a dramatic increase in context-switch overhead such as draining of the network between context switches, then it is a poor design. To move parallel machines out of their niche as special purpose machines for "Grand Challenge" problems to cost effective workhorses, flexible, efficient support for multi-tasking is needed.

While few will disagree that the three M's are desirable in a message passing system, most available parallel machines today fall short in some aspects. Machines usually provide only a single message passing mechanism, have little hardware support for multithreaded access to message passing primitives, and require a large overhead for context switching (when it is available at all).

This paper argues that it is possible to build a cost-effective *and high performance* parallel processing system from a cluster of SMP workstations, a high-performance interconnection network, and a simple well designed network interface unit. A particular implementation supporting the three M's is described in this paper and serves as the proof of concept. Subsequent sections describe how, within an integrated message passing framework, multiple data transfer mechanisms are utilized to efficiently support a diversity of communication granularities. The handshake between the application code and the network interface unit is limited to atomic, independent operations which are compatible with multi-threaded sharing. Multi-tasking is supported with a multitude of hardware supported message passing queues, and virtual destination queue names.

The contributions of this work are:

- a memory-bus based message-passing interface supporting several data-transfer mechanisms, each suitable to different message granularity, data location, and programming model.
- the use of inherently atomic single uncached read or write at the "commit points" of message

passing operations to achieve a thread-safe interface.

- the use of 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 message queues, all of which are always active. Efficient use of expensive resources through the use of a queue hierarchy comprised of a small number of fast resident message queues and a large number of slower non-resident resources, with transparent, dynamic migration of a virtual queue between them.
- A low-overhead one-poll mechanism that returns the highest priority message from a set of message queues.

The organization of the paper is as follows. Foundational material related to message passing support by a memory-bus attached network interface unit is presented in Section 2. Foundational material continues with an overview of StarT-Voyager Network Endpoint Subsystem (NES) in Section 3. The heart of the paper then follows in the next two sections. In Section 4 the message queue organization describes the virtual message queues, the translation mechanism, and message-space protection. The implementation of the two novel message passing mechanisms, Express and Tag-on are then described in Section 5. Estimated performance of the StarT-Voyager NES, obtained with a cycle-level accurate C simulator that has been validated against the Verilog RTL model of the NES implementation, is presented in Section 6. The timing demonstrates that no single mechanism adequately addresses the range of message sizes and execution model. Related work is discussed in Section 7 before we conclude in Section 8.

2 Message Passing on the Memory Bus

The message passing abstraction covers a wide range of application needs. As a foundation to the rest of the paper, a short overview of our assumed message passing software model is presented. A similar short overview is then presented of the hardware considerations for message passing implementations. A final short overview of straight-forward implementations for messages concludes this section.

We focus on the software model in which there is a set of ports in the system with a transmit and receive queue associated with each port. A message space consists of a set of (dynamically) connected ports. The two basic message passing operations, send and receive, each consist of three steps: get a buffer, process, then release the buffer. For a send operation, the buffer, once acquired, is filled with the message data, and the release of the buffer indicates that the message is ready to be sent. Once a nonempty buffer is acquired from a receive queue by some application, the message data in the buffer is examined. The release allows the buffer to be used by another message. If several threads are concurrently sharing a message queue, mutual exclusion is needed to ensure correct operation.

2.1 Assumptions

This paper focuses on hardware consisting of a cluster of symmetric multiprocessor (SMP) workstations, connected with a high-bandwidth, low-latency, multi-staged interconnection network. A network endpoint subsystem (NES) interfaces the network to the *memory* bus of each SMP. The design of a low-latency, low processor overhead, high bandwidth message passing system requires exploiting the strengths of the commodity hardware and avoiding the weaknesses inherent in the microprocessor's design. The following is a list of key SMP characteristics to consider. (Examples are drawn from a PowerPCs 604 based SMP to provide concrete illustrations.)

1. Off-chip accesses are expensive. Bus transactions require multiple bus clock cycles and a processor clock cycle is typically three or more times faster, so any off chip access translates to tens of processor clock cycles.
2. If sufficient data is involved, burst transfers are more efficient than uncached accesses.
3. Main memory has long access times – *e.g.* for a typical PC-class machine, read latency is about 10 bus clock cycles and write latency is half of that.
4. The weak memory model, found increasingly among today's processors, often demands the use of memory barrier instructions. These can be expensive *e.g.* SYNC on the PowerPC 604 was measured to require 15 to 20 processor clocks.
5. Cache management instructions, when available in user mode, are expensive. On the PowerPC 604, a single cache-line Flush takes around 20 processor clocks.
6. Coherent bus protocol details, such as whether cache-to-cache transfer is supported, are important to low level network interface unit design choices.
7. Atomicity primitives for dealing with races between concurrent or parallel threads are not cheap; a pair of load-reserve/store-conditional instructions, commonly used in modern microprocessor for taking a lock, takes over 30 processor cycles on the 604 even under the no contention situation.

2.2 A Basic Message Passing Implementation

A basic implementation of message passing support within a network interface unit connected to the memory bus of an SMP exploits the above architectural assumptions. Thus, transmit and receive queues consist of a *cacheable* message buffer region, and *uncached* producer and consumer pointers. The message buffer region is arranged as a circular FIFO with the whole queue visible to application software. Message data is thus efficiently transferred over the bus. The producer and consumer pointers are needed to indicate when a message is ready to be transferred. They are uncached so that a pointer update immediately triggers processing, thereby avoiding the need for full coherence. A cache-line can either be actively flushed from processor or externally fetched by the network interface unit; the exact mechanism depends on the processor architecture. Message data cache-lines may be eagerly fetched before the length field in the message header is read.

Although clever implementations of uncached pointers may imply atomic updates, the above scheme is not thread safe due to the circular FIFO management. In a multi-threaded environment, the entire "get buffer, process, release buffer" sequence may need to be executed within a critical section.

2.3 Implementation of Long Messages: DMA

DMA is an efficient mechanism for moving contiguously located data between the memory of one site and that of another. Most of the packetization and formatization of blocks of data required for transmission across the interconnection network can be done by simple hardware within the network interface. All that is needed is a mechanism for the processor to communicate the request to the network interface unit. A mechanism similar to the basic message passing one outlined above is sufficient.

3 StarT-Voyager Architecture Overview

StarT-Voyager is a parallel system designed to support both message passing and coherent distributed shared memory. StarT-Voyager is constructed from a collection of commercial SMPs connected by a fast network which interfaces to each SMP's coherent memory bus via a Network Endpoint Subsystem (NES) card. Each SMP, referred to as a *site*, is a desktop-class, commercial, dual-PowerPC 604 SMP. It uses a stock PC/workstation class motherboard which can accommodate two processor cards. Each processor card normally contains one 604 processor and an in-line L2 cache. For StarT-Voyager, we replace one of the two processor cards in each SMP site with an NES card, making each site into a uniprocessor system[1] This 604 processor, referred to as the application processor, or *aP*, runs a copy of AIX, augmented with a parallel layer to coordinate parallel job execution.

Each NES card is attached to the Arctic network, a Fat-tree [27] network constructed with Arctic router chips[8]. Both the Arctic router chip and the network boards are designed and implemented at MIT as part of the StarT-Voyager project. Each Arctic router chip is a 4x4 packet-switched router with support for variable length packets, virtual-cut-through, FIFO ordering, and two logical networks. The Arctic network employs hardware link-level flow-control to provide a lossless packet routing service. 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 NES card and the network, each site has a peak network communication bandwidth of 320 MBytes/sec. A fast router together with the Fat-tree topology allows the Arctic network to provide redundancy[2], the ability to exploit network locality and very high bisection bandwidth – over 5 GBytes/sec in a 32 node machine.

Figure 1 shows the organization of a StarT-Voyager site, with details of the NES. The NES manages several regions of memory, each providing a different functionality. The NES is designed for both speed and flexibility. The NES Core provides dedicated control hardware and data paths for the most common operations. Actions that are expected to occur very frequently are handled *completely* in hardware and are thus very fast. An embedded processor, referred to as the service processor (*sP*) provides flexibility and extensibility. 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 further reduced by the symmetry of the organization: from the NES Core's perspective, the sP-subsystem is almost identical to the aP complex. The sP subsystem, however, has additional functionality. The sP observes all aP memory operations to one of the NES address regions called the *sP Serviced Space*, and is capable of initiating transactions on the aP-bus, controlling any NES state, and handling all exceptional situations. These features are extremely flexible and can be used to extend the functionality of StarT-Voyager. They will be used to implement coherent shared memory and *non-resident message queues* (see Section 4.2). Both the aP and sP have access to the same message passing mechanisms, but each has its own set of resources. Two banks of dual-ported SRAM, the aSRAM and sSRAM, provide storage space for resident message queues.

The NESCtrl, a small ASIC, implements all the hardware control functionality. It also holds

[1]We desired multiple processors per site, but currently no such PowerPC 604 motherboard exists at a reasonable price.

[2]Multiple paths through a Fat-tree network between most pairs of source-destination sites provide redundancy and hence fault tolerance.

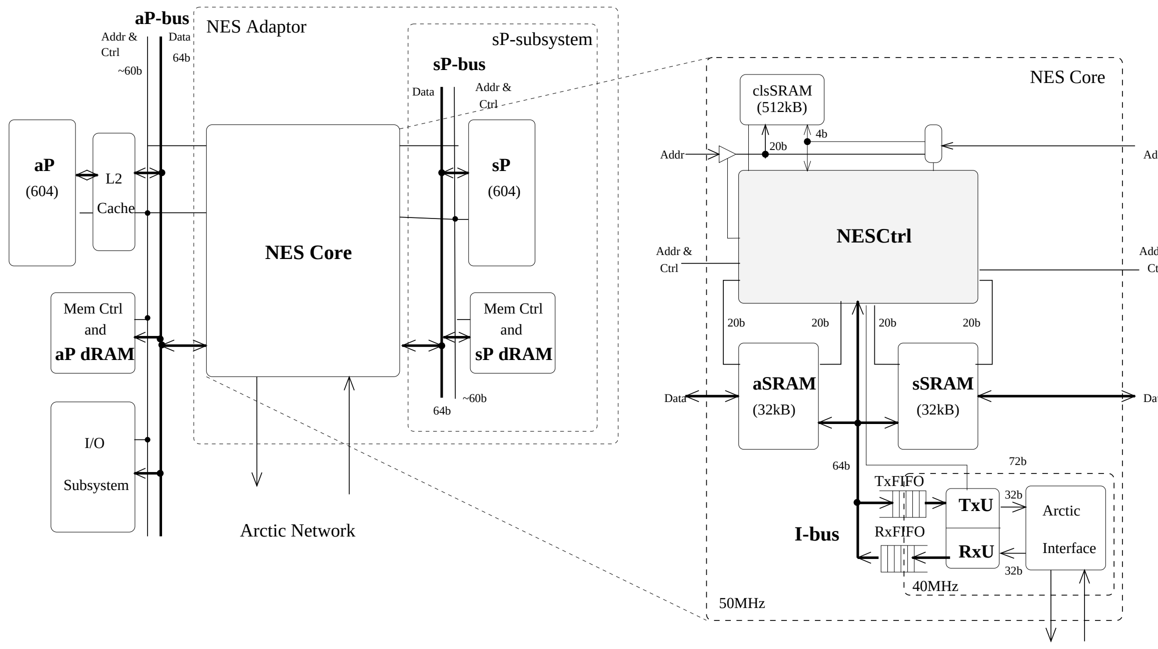


Figure 1: A StarT-Voyager site.

all the control state, including the queue pointers of message queues. The I-bus, the central data path of the NES core over which all messages flow, has a 64 bit-wide data path running at 50MHz. This roughly matches both the bandwidth of the 60X bus which runs at the same speed and has the same data path width, and the 320 MBytes/s aggregate bandwidth between an NES and the Arctic network. Outgoing messages undergo destination address translation (See Section 4.1) and are tagged with a source identifier 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) at the destination NES. The TxU is also responsible for the expansion of an Express message (Section 5.1) into an Arctic packet, while the RxU performs the compression. Finally, the Arctic Interface handles low-level signaling and analog electrical conversions (TTL to ECL) between the NES and the Arctic Network itself. The clsSRAM, which stores cache-line state bits in coherent distributed shared memory implementations, is not involved in message passing.

4 StarT-Voyager Message Queue Organization

The StarT-Voyager parallel system supports the three M's of message passing through several mechanisms. Hardware-enforced virtual message destination name space private to each process, together with virtual memory protection of private, memory-mapped message queues ensures full protection under multi-tasking.

4.1 Destination Translation Mechanism

All user-level messages in StarT-Voyager are directed towards destination queues in a virtual queue name space that is private to the sending job, and subjected to translation into physical queue name by the NES hardware. Sharing many similarities with traditional virtual memory systems,

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

the queue name translation offers protection by restricting the destination queues addressable by a job. As in a virtual memory system, such a mechanism not only offers protection, but allows transparent, dynamic swapping of a virtual queue between different hardware resources. System code on StarT-Voyager uses this capability to transparently switch message queues between fast Resident queue resources and slower but cheaper and more numerous Non-resident queue resources (see below). The virtual queue name space also facilitates dynamically adjusting the number of processors devoted to a parallel job, allowing several virtual queue names to map to the same physical queue.

The NES hardware consults a hardware *Destination Table*, analogous to a page table, to translate a virtual queue name into a physical destination site number required by a network packet for routing, and a received queue name used at the destination site to identifier the destination queue. Figure 2 illustrates the entries of the Destination Table [3]. The Destination Table is configured by software through an NES address space which should only be accessible to system software.

The NES also attaches a source identifier to each outgoing message, which the destination process can use to reply to a message. For symmetry in translation the source identifier must look like a logical destination specifier at the destination. In StarT-Voyager, it is convenient to keep the source identifier in the Destination Table to facilitate attaching of the source identifier to an out-going message. The logical destination specifiers should have the same meaning across all of the processes in a single parallel job, allowing the passing of logical destination specifiers to other processes.

Fast context switching is made possible by multiple messages queues. Specifically, each StarT-Voyager site has 512 pairs of transmit/receive queues. In a standard allocation scheme each process of a parallel job is assigned its own unique set of message passing queues[4]. Access to a local queue is controlled by the virtual memory mapping of the processor, *i.e.*, only specific queues are mapped to a process's virtual memory space. A running parallel job can be suspended and "swapped out" by suspending and swapping out each of its constituent processes independently. Though the process is no longer running on the processor, its message queues are still active and messages can continue to arrive and be queued into them. Swapping out a parallel process requires no global coordination to ensure correctness. This is in contrast to traditional MPP's such as the CM-5[28] which supports only a single set of network queues, forcing the queues as well as the network to be entirely swapped out at the same time a parallel job is being swapped out. The liberation of job scheduling from globally synchronous gang scheduling opens up the possibility for novel

[3]An inverse table could be associated with each receive queue so that only messages from a specified set of sources will be accepted. Such additional hardware would protect against untrusted OS or sP code at a source. StarT-Voyager does not implement this additional hardware.

[4]The smallest allocation unit is two transmit and two receive queues.

coscheduling approaches[41].

The virtual destination queue name space approach is more general than other machines' approach of using a Parallel Job Identifier (PJID) which is added automatically to every outgoing message and verified at the destination[36]. The latter scheme defines monolithic protection domains in which every process can communicate with every other process in the domain. While suitable for parallel applications, it does not offer enough flexibility in a distributed environment, where a server application may want to communicate with a number of client applications without allowing a client to communicate directly with every other client. StarT-Voyager's virtual destination naming scheme supports such non-uniform communication patterns.

4.2 Message Queue Hierarchy

The large number of queues at each StarT-Voyager site are far too many to be supported directly in hardware. Each NES supports 16 pairs of transmit and receive queues (8 Express/Tag-On and 8 Basic) directly in hardware. The hardware queues act as a software-managed cache for the 512 pairs of transmit and receive queues. The queues mapped to hardware queues are called *Resident* while the others are called *Non-resident*. Because both Resident and Non-resident queues are accessed via memory-mapped interface, application code is not directly aware of whether its queue is mapped to Resident or Non-resident resource. This mapping can thus be changed dynamically and transparently by the system software.

When a message packet arrives at an NES, the NESCtrl determines if the destination queue specified in the message header is resident and if it has enough room for the message. If both conditions are true, the NESCtrl enqueues the message accordingly. Otherwise, the NES enqueues the message in the *miss queue* serviced by the sP. Non-resident queues in StarT-Voyager are implemented by mapping the message buffer space into the aP's DRAM and the control/status state into sP Serviced address space. Messages can continue to be received into and sent from non-resident queues with the sP's assistance. Mapping of the control/status state into sP Serviced address space allows the sP to support the non-resident queues without actively polling on the state of each queue. Instead, access to control/status state results in bus transactions which trigger sP processing in an event driven fashion.

4.3 The OnePoll Mechanism

With an application potentially having a large number of receive queues, it becomes expensive to continually poll each one. Response time and processor overhead for any receive is increased. To address this overhead, StarT-Voyager provides the ability to poll from multiple queues *simultaneously*. Application code specifies to the NES which queues to poll using the usual single-queue polling mechanism but to a phantom queue. Either a special empty message is returned, indicating that all the queues are empty, or the message from the highest priority queue is returned. For ease of implementation, there is a fixed priority between queues. Since the polling mechanism depends on an understanding of the StarT-Voyager message passing types, the implementation details are postponed to the end of the next section.

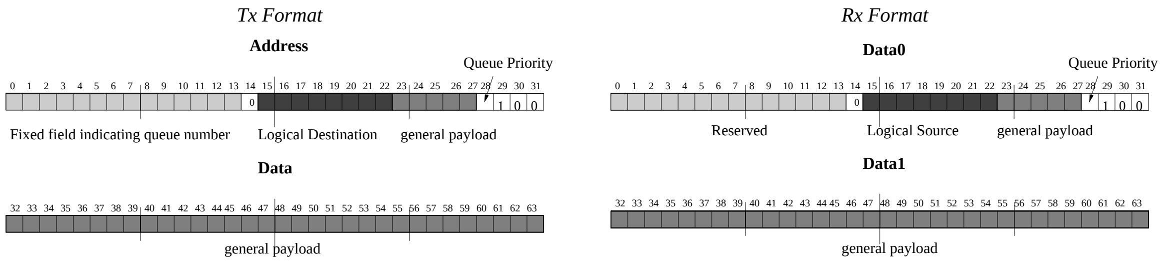


Figure 3: Express Message Formats

5 Two Novel Message Passing Mechanisms: Express and Tag-On Messages

StarT-Voyager provides four message passing mechanisms, two fairly standard: Basic and DMA[5], and two more novel: Express and Tag-On. Express messages have very low processor overhead suitable for very short messages. The Tag-On mechanism is an extension of Express message mechanism. Both these mechanisms are unique and include thread-safe interfaces.

5.1 Express Messages

The Express message mechanism provides the ability to atomically send/receive a message in a multi-threaded environment *without* synchronization instructions, dramatically reducing the cost of sharing network hardware between multiple threads and/or multiple SMP processors. It does so by enabling an entire message send/receive to be performed using a single inherently atomic uncached write or read. In order to maximize the data sent by an Express message, low-order address bits are used for the logical destination address and additional data, permitting almost 5 bytes of data to be sent with a single uncached 4 byte write. Such address bit stealing has been proposed before[23], but we are familiar with only one other machine (Cray T3E) that uses it.

A major challenge of the Express message design is to maximize the amount of data transported by a message while keeping each compose and launch to a single, uncached memory access. The Express message mechanism assumes that the (i) message send queue specifier, (ii) message destination, and (iii) 5 bits of data are packed into the *address* of an uncached write. The NES automatically transforms the logical destination into a message header and appends the 5 bits of data in the address to the 32b of written data to complete the Arctic message. A diagram showing a simplified format for sending and receiving Express messages is given Figure 3. Additional address bits can be used to convey more information but consumes a larger (virtual and physical) address space which can negatively impact TLB's if the information encoded into the address bits do not exhibit "good locality". Alternate translation mechanisms such as PowerPC's block-address translation mechanism[33] can mitigate this problem, but such mechanisms are highly dependent on both the processor architecture and OS support.

To reduce the data read by a receive handler, the NES reformats a received Express message packet into a 64 bit value as illustrated in Figure 3. A receive can be accomplished with a 64 bit uncached read into an FPR. The data is subsequently moved into GPRs via (cache) memory[6]. Alternatively, two 32 bit loads into GPRs can be issued to receive the message.

[5]The details are available in a technical report.

[6]PowerPC does not support 8 Byte load into a pair of GPRs nor direct data transfer between a FPR and a GPR.

The addresses for accessing Express message queues do not specify specific queue entries. Instead, the NES provides a FIFO push/pop interface to transmit and receive Express messages. When a processor performs a write to enqueue the message, the NES provides the necessary buffer address to put that message into NES SRAM. Likewise, when the processor performs a read to receive a message; the NES provides the necessary SRAM address from which to read the message. Speculative loads from Express message receive regions are disabled by setting the page attributes appropriately[7]. When an application attempts to receive a message from an empty receive queue, a special Empty Express message, whose content is programmable by system code, is returned. If message handler information is encoded in the message, such as in Active messages[44], the Empty message can be treated as a legitimate message with a "no action" message handler.

Because interaction between application code and the NES occurs only through single uncached accesses, the actions are inherently atomic. The Express message interface is thus thread-safe, and multiple threads of a single job running simultaneously on an SMP can shared an Express message queue without extensive mutual exclusion locking.

5.2 Tag-On Messages

The Tag-On message mechanism extends the Express message mechanism to allow additional data from NES SRAM to be appended to an out-going message. The Tag-On message mechanism was designed to eliminate a copy if message data was already in the NES's SRAM. It is especially useful for implementing coherent shared memory protocols, and for multi-casting a medium sized message. It is also thread-safe. As composed by an application, a Tag-On message looks similar to an Express message with the addition that several previously unused address bits now specify the SRAM location where the additional 1.5 cache-lines or 2.5 cache-lines of message data can be found. The address field is kept small by specifying only an offset from a programmable base address known to the NES, so that it will fit within the limited size of an Express message.

At the destination NES, a Tag-On message is partitioned into two parts that are placed into two separate queues. The first part is its header which is delivered like an Express message, via a queue that appears to be a hardware FIFO. The second part, made up of the data that is "tagged on", is placed in a separate buffer similar to a Basic message receive queue in that it utilizes explicit buffer deallocation.

Tag-On messages have the advantage of decoupling the header from the message data, allowing them to be located in non-contiguous addresses. This is useful in coherence protocols when shipping a cache-line of data from one site to another. Suppose the sP is responding to another site's request for data. In StarT-Voyager, this is achieved by the sP first issuing a command to move the data from aP-DRAM into aSRAM, followed by a Tag-On message that ships the data to the requester. In addition to the cache-line of data, a Tag-On message inherits the 37 bit payload of Express messages which can be used in this instance to identify the message type and the address of the cache-line that is shipped. Cache-line data may also be brought into the NES without the sP asking for it, *e.g.* the aP's cache may initiated a write-back of dirty data. In such cases, Tag-On message's ability to decouple message header and data allows the data to be shipped out without further copying.

Tag-On messages are also useful for "specialized" multi-casting. To multi-cast some data, an application first moves it into NES SRAM. Once that is done, the application can economically send it to multiple destinations using a Tag-On message for each one. Thus, data is moved over the system memory bus only once at the source site, and the incremental cost for each destination is an uncached write to indicate a Tag-On message.

[7]In the PowerPC architecture, disabling speculative loads is done by asserting the guarded bit.

5.3 OnePoll Implementation

By extending the Express message receive mechanism slightly, StarT-Voyager provides the OnePoll mechanism, the ability to poll from multiple queues *simultaneously.* The NES implements the Express message polling mechanism with internal pointers that generate the SRAM address from where to read an Express message. Empty message is thus implemented with a dedicated buffer in the SRAM containing the Empty message. It is straight forward to extend this to choose an SRAM address from among those of several queues, based on whether they are empty/non-empty, and their relative priority. 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.

6 Projected Performance

This section presents the estimated performance of StarT-Voyager using the three LogP[13] metrics:

- **Processor overhead:** Processor overhead captures the overhead incurred by application code when sending and receiving messages. These were generated by running representative code on a PowerPC 604 machine. To model accesses to NES SRAM, the test code is crafted to cause L1 misses which hit in L2 to simulate NES SRAM accesses. Several areas where the machine on which measurements were taken differed from StarT-Voyager are: a 2:1 processor/bus ratio instead of 3:1; and a look-aside L2 cache instead of in-line L2 cache.

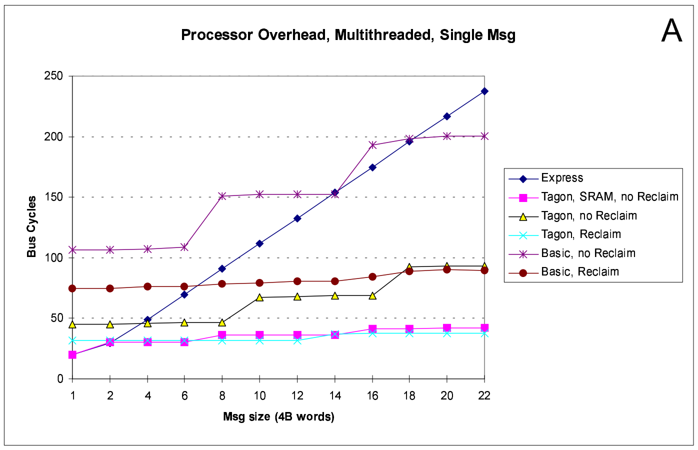
- **Latency:** Latency measures the one-way end-to-end latency, including the software overhead for sending and receiving messages. The processor overhead component is obtained as described above; the latency through the NES and network is obtained with a cycle-by-cycle RTL C simulator. This simulator is extremely accurate as it also serves as the hardware specification for our hardware implementors. The latency reported includes the time for one network switch (2 hops).

- **Gap/bandwidth:** Gap measures the sustainable interval between messages of a specific type. The inverse of gap multiplied by message size gives bandwidth. We obtained two sets of gap numbers, one from our RTL C simulator showing the bottleneck in the NES and network, and another from the Processor Overhead as measured by our test program. The larger of these two number is the gap number reported.

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 bus and network and not the NES.

6.1 Performance in a Multi-threading Environment

Figures 4 and 5 show the performance of the various message passing mechanisms in a multi-threaded environment where a message queue is shared between several threads. The horizontal axis is message size in units of 4B words; the vertical axes is in bus clocks (20 ns) for all but Bandwidth, which is in Bytes/sec. Each graph includes six different cases. Most of the them are self explanatory. "No Reclaim" vs "Reclaim" refers to how cached message data is kept coherent. The no Reclaim case involves software using explicit Flush instructions to push message data out of the caches. For the Reclaim case, the NES extracts transmit data from the aP cache and flushes old receive data from the aP cache. In Reclaim, the NES does not implement full consistency, but



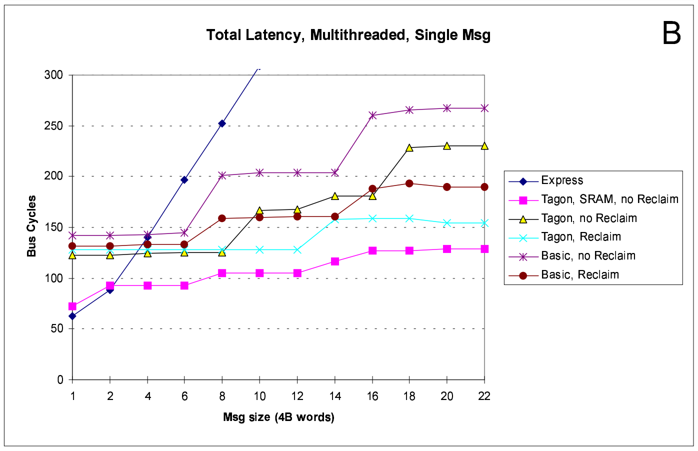
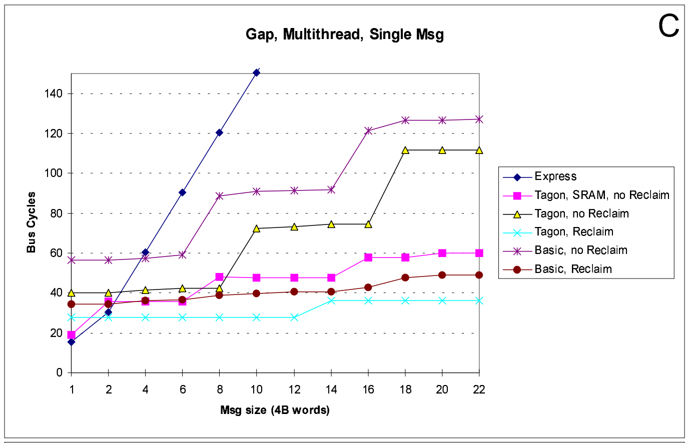


Figure 4: Processor overhead and latency in a multi-threading environment.



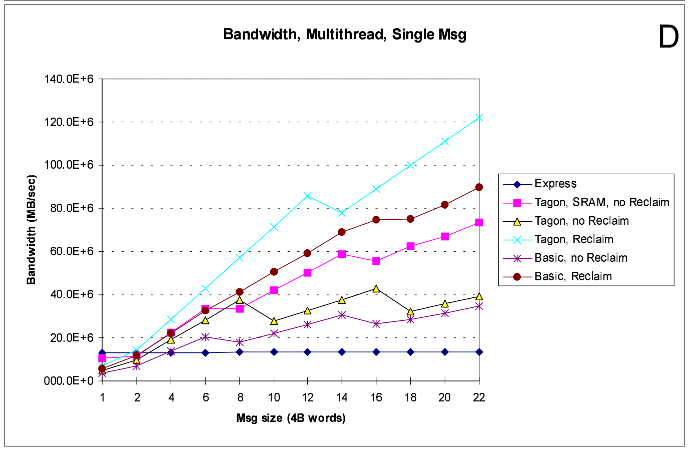


Figure 5: Gap and bandwidth in a multi-threading environment.

performs coherence operations when notified by pointer update that messages has been sent or received. Such lazy coherence requires no cache-tags and is easy to implement. "Tag-On, SRAM Resident, no Reclaim" refers to the case where the data that is "tagged on" is already in the NES SRAM. This is the case for the second and subsequent destinations of a multi-casted message.

For Basic message queues, correct operation requires locking the queue when sending or receiving a message. Owing to their inherent thread-safe design, Express and Tag-On messages queues do not require locking. As shown in the figures, Tag-On messages have the lowest processor overhead, latency, and highest bandwidth across almost the entire range of message sizes. Basic messages is always worse than Tag-On messages. Comparing Tag-On with Reclaim and Basic with Reclaim, one sees that processor overhead for Basic is more than twice that for Tag-On. In fact, the performance numbers for Basic message does not take into account actual lock contention between multiple threads; the current numbers only account for the cost of successfully obtaining a lock on first try.

The performance numbers also shows that Reclaim support 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. This cross-over point is lower than we had expected and is due to the fact that uncached accesses are relatively inefficient compared to cache-line burst transfers both in its bus and bus interface buffer utilization. Nevertheless, in certain applications, such as cache-coherent distributed shared memory protocols, single-word messages are so common, they warrant special consideration. A mechanism with such low overhead and latency will likely be used by applications that previously did not use small messages because of their high cost.

6.2 Performance in a Single-threading Environment

Though Tag-On messages are better than Basic messages in a multi-threaded environment, are they still competitive in a single-threaded environment? Figures 6 and 7 show the performance of the different message types when used in single-threaded fashion. Because locking is no longer necessary, Basic messages now deliver comparable, and sometimes slightly better performance than Tag-On messages. The difference is not significant in most cases. Reclaim remains a better choice than explicit software Flush, while Express message is still better for messages of one word.

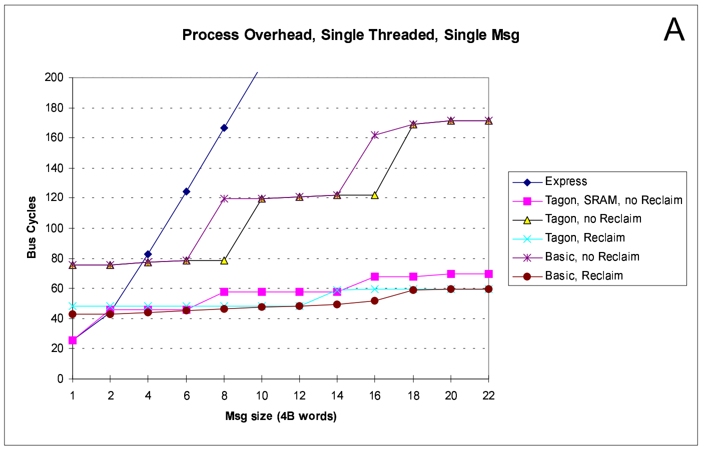
With little penalty in the single-threaded scenario and significantly better performance in the multi-threaded case, Tag-On is a better mechanism. Basic message does have some benefits which are not reflected in the performance numbers. The use of pointers allows aggregation, so that when several Basic messages are sent out or received together, some savings in handshake between application code and NES 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 far superior than all but the very best supercomputers today.

7 Related Work

The network interface unit in a parallel system can be located at one of four places, each at a different distance from the processor: (i) in the processor core; (ii) on a cache interface; (iii) on the cache-coherent memory bus; or (iv) on the I/O bus.

Though good for small to medium size messages, none of the on-chip (in the processor core) network interfaces provide special bulk transfer hardware. Almost all these machines are proof-



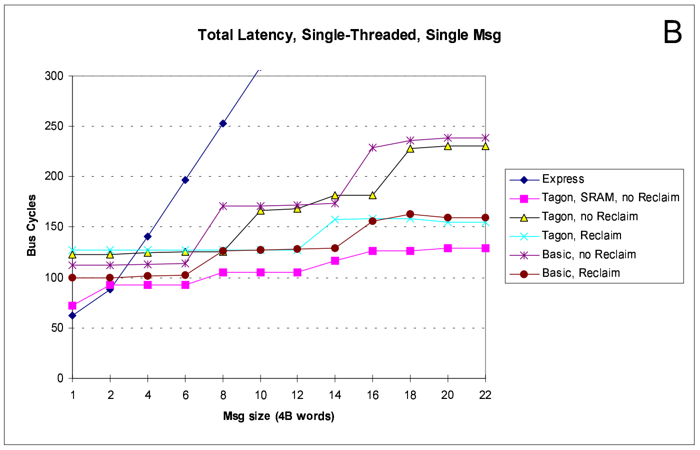


Figure 6: Processor overhead and latency in a single-threading environment.

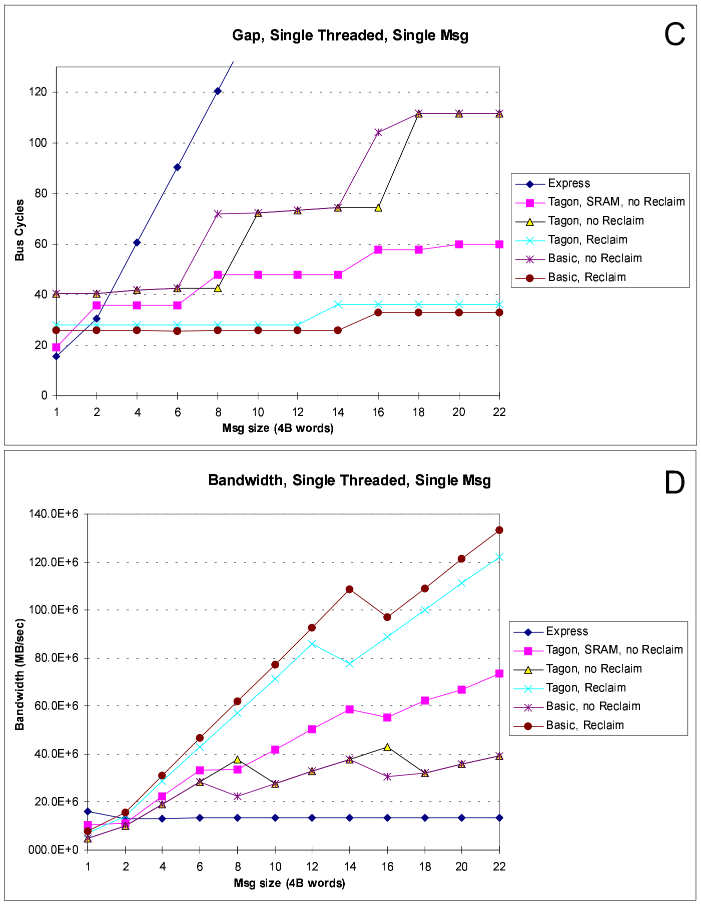


Figure 7: Gap and bandwidth in a single-threading environment.

of-concept experimental platforms where multi-tasking is rarely addressed. Multi-threaded sharing of message queue is not addressed except among the Dataflow machines; these machines support only small messages sent and (implicitly) received with individual instructions. Examples of processors with integrated network interfaces include transputers[45], the iWarp systolic processor[6, 7], dataflow processors like Monsoon[15] and the EMC-R[38], and hybrid processors such as the MDP[16], M-machine[17], and the MIT/Motorola 88110MP[36].

The Alewife[1] is an example of a network interface on the L1 cache interface. Alewife supports multi-part message specification and its Sparcle processor provides hardware multi-threading. Race conditions during access to network interface is prevented by disabling multi-threading during each message interface access critical section. The design for StarT-NG[11] proposed a network interface to the L2 cache interface but only one message type is supported.

Placing the network interfaces on I/O buses designed to accommodate third party devices, is currently a popular approach. Commercial examples include machines like the IBM SP-1[21] and SP-2[2, 42], and plug-in extensions like Myricom's Myrinet[5] and DEC's Memory Channel[18], while research efforts include Princeton's SHRIMP[4] and StarT-Jr[19]. Most I/O bus-based message passing support utilizes main memory DRAM to house user-level message queues. The NIU delivers message between such queues on different nodes. In some designs, messages are sent out by writing directly across the I/O bridge chip to the NIU. Although these projects would like to provide support for the three Ms, research efforts have so far been more concerned about overcoming the challenge of traversing at least two buses, and crossing one or more bridge chips before even reaching the network interface. With multiple user-level message queues housed in main memory, most I/O bus based NIUs support multi-tasking fairly well. Multi-threaded access to the same message queue generally has no special support.

Our work is focused on network interface connected to the memory bus. Many of the machines that support distributed shared memory locate their network interface unit (NIU) on the memory bus. Among them, DASH[29], Exemplar[12, 9], and NUMA-Q[40] do not directly support message passing. Message passing on the T3D[25] is implemented with its remote memory access and atomic fetch-and-add capabilities, the latter for remote message queue space acquisition. Message data is transfered using the machine's shared memory capability. Multi-granularity support includes a bulk transfer engine, and special support for barrier and Eureka synchronizations. Its successor, the T3E[39] provides more traditional form of message passing – messages are launched from special off-chip E registers, and received from buffers in main memory. Support for multiple receive queues facilitates multi-tasking in both the T3D and T3E. FLASH[26] has a programmable memory controller, which can be harnessed to implement a virtual memory mapped NIU. Just about any form of NIU can be implemented in firmware. Thus far, their planned message passing support appears to be the memory to memory bulk copy form. Typhoon[37], while having some hardware support for message passing, is primarily a shared memory machine.

The CM-5[43] is a message passing machine with very respectable performance for its time. However, it does not handle larger messages (messages bigger than a few words) efficiently, and multi-tasking support, though present, is not flexible. The Paragon[22] uses a second processor at each node to copy messages between the otherwise unprotected hardware network interface and software message queues in main memory. This provides protection in a multi-tasking environment but introduces the overhead of copying. The Meiko CS-2[3] also has a message coprocessor, and operates in a similar way as the Paragon, though its design is more efficient in that the coprocessor connects directly to the network.

More recent work by Mukherjee et. al.[34] on Coherent Network Interface (CNI), presents a nice and efficient network interface. The design however does not address the issue of sharing a message queue among several concurrent threads. It also does not adapt the data transfer mechanism as

Machine	processor speed (MHz)	Interface Location	one-way nearest neighbor latency (μs)	Send Overhead (μs)	Receive Overhead (μs)	Peak Bandwidth (MByte/sec)
StarT-Voyager (AM)	*150*	*memory bus*	*1.25*	*0.13*	*0.51*	*140*
CM-5 (CMAM)[30]	33	memory bus	5.7	1.5	1.25	10
Paragon (AM)[31]	50	memory bus	8.4	2.2	1.0	145
Meiko CS-2 (AM)[14]	66	memory bus	10.8	1.7	1.6	39
T3D (FM)[24]	200	memory bus	10	4		*N.A.*
HP+Medusa (AM)[32]	99	graphics bus	10.15	*N.A.*	*N.A.*	12
StarT-Jr (AM)[20]	120	I/O bus	27	1.5	1.1	7
Sparc20+ATM (AM)[10]	50	I/O bus	33	3		14
SP-2 (AM)[10]	66	I/O bus	25.5	4		34
Sparc20+Myrinet(AM)[14]	50	I/O bus	15.7	2.0	2.6	20
Sparc20+Myrinet(FM)[35]	50	I/O bus	25	*N.A.*	*N.A.*	20

Figure 8: Message passing performance of several representative systems. StarT-Voyager's numbers are estimates obtained in the fashion described in Section 6.

communication granularity changes. Multi-tasking, though briefly mentioned, was not discussed in detail.

Figure 8 provides a *very rough* comparison of StarT-Voyager's message passing performance compared to other machines. Aside from StarT-Voyager's numbers, the numbers in Figure 8 are reported by other researchers, and were measured either with Active Message (AM)[44] or Illinois Fast Messages (FM)[35], both of which are very fast message passing libraries, and are taken from many articles[30, 32, 31, 14, 10, 20, 24, 35]. It is impossible to conduct a completely fair comparison because the machines are from different time periods, not all the measurements reported are for the same message size (varies from zero to several words of payload), and some implementations include sliding window protocols to deal with lossy network hardware which also provides flow-control as a side benefit. Nevertheless, the table does provide some insight into how StarT-Voyager's performance compares with these other machines.

8 Conclusions

A 32-node StarT-Voyager is currently being built and is expected to be running both message passing and coherent distributed shared memory by mid 1997. Despite the usual impact on performance that comes when implementation must meet real-world demands, our detailed simulations indicate that StarT-Voyager is expected to provide superior performance while supporting the three M's necessary to make the machine generally usable, a claim that few other machines (if any) can make. The simulations have shown other facts as well. Express messages are significantly faster in all metrics than any other mechanism for messages of about a word. We believe that a substantial fraction of messages in many codes could exploit this mechanism, increasing the importance of Express message performance. Tag-On with Reclaim significantly outperformed Basic messages in all metrics in a multi-threaded environment and equaled Basic messages in a single-threaded environment.

Express/Tag-On messaging is achieved by combining two well known techniques, namely (i) using a single atomic memory operation to indicate an event and (ii) a FIFO push/pop abstraction rather than a random access memory abstraction, along with a lesser known technique, address bit stealing. This combination of techniques should be considered for any processor-to-bus-device communication requiring thread-safety.

Real-world considerations restricted the amount of SRAM embedded in the NES. Fortunately, in addition to providing hardware protection, the virtual queue mechanism enables support for a large number of queues of which only a small number can fit in SRAM. Such tradeoffs are probably not limited to StarT-Voyager.

As a note to processor and system architects, attaching specialized hardware to the memory bus promises great performance benefits. This performance is often limited due to inadequate facilities; for example, inexpensive mechanisms to flush cache-lines, achieve control over memory operations ordering and synchronize would be of great value. A better processor architecture would enable StarT-Voyager to achieve even better performance.

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] E. Barton, J. Cownie, and M. McLaren. Message passing on the meiko cs-2. *Parallel Computing*, 20:497 – 507, 1994.

[4] 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.

[5] 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.

[6] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, , and J. Webb. iWarp: An Integrated Solution to High-speed Parallel Computing. In *Proceedings of Supercomputing '88, Orlando, Florida*, pages 330 – 339, Nov. 1988.

[7] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 70 – 81, May 1990.

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

[9] T. Brewer. A Highly Scalable System Utilizing up to 128 PA-RISC Processors. Technical report, Convex Computer Corporation, 1995. (Available from http://www.convex.com/tech_cache/technical.html in Jul 96.).

[10] 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).

[11] D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and A. Boughton. StarT-NG: Delivering Seamless Parallel Computing. In *Proceedings of the First International EURO-PAR Conference, Stockholm, Sweden*, pages 101 – 116, Aug. 1995.

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

[13] 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.

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

[15] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[16] W. J. Dally, R. Davison, J. A. S. Fiske, G. Fyler, J. S. Keen, R. A. Lethin, M. Noakes, and P. R. Nuth. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, Apr. 1992.

[17] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, Michigan*, 1995.

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

[19] 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.

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

[21] IBM. *IBM Scalable POWERparallel System Reference Guide*, 1993. IBM Publication Number G325-0648-00.

[22] Intel Corporation, Intel Corporation, Supercomputer Systems Division, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006. *Paragon XP/S Product Overview*, 1991.

[23] C. F. Joerg and D. S. Henry. A Tightly-Coupled Processor-Network Interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA*, pages 111–122, Oct. 1992.

[24] 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).

[25] 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.

[26] 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.

[27] C. E. Leiserson. Fat-trees: Universal Networks for Hardware-efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10), Oct. 1985.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* Morgan Kaufman Publishers, Inc., San Francisco, CA, second edition, May 1994.

[34] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[35] 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.

[36] 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.

[37] 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.

[38] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. *16th Annual International Symposium on Computer Architecture*, pages 46–53, 1989.

[39] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA*, pages 26 – 36, Oct. 1996.

[40] 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,).

[41] 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.

[42] 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.

[43] Thinking Machines Corporation, 245 First Street, Cambridge, MA02142, USA. *Connection Machine CM-5 Technical Summary*, Nov. 1993.

[44] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture Conference Proceedings, Gold Coast, Australia*, pages 256 – 266, 1992.

[45] C. Whitby-Strevens. The Transputer. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 292 – 300, 1985.