LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Message Passing Support on StarT-Voyager

Computation Structures Group Memo 387
July 16, 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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Message Passing Support on StarT-Voyager

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

July 16, 1996

Abstract

No single message passing mechanism can efficiently support all the different types of communication that occur naturally in most parallel or distributed programs. MIT's StarT-Voyager, a hybrid message passing/shared memory parallel machine, provides four message passing mechanisms to achieve very high performance over a wide spectrum of communication types and sizes. Hardware and operating system enforced protection allows direct user-level access to message passing facilities in a multiuser environment. StarT-Voyager's protection scheme improves upon past designs by not requiring strictly synchronized gang-scheduling, and by supporting non-monolithic protection domains. To minimize the development effort and cost, the machine is designed to use unmodified commercial PowerPC 604-based SMP systems as the building block. A Network End-point Subsystem (NES) card which plugs into one of each SMP's processor card slots provides the interface to Arctic, a low-latency, high-bandwidth network currently under development at MIT. This paper describes StarT-Voyager's message passing mechanisms and their predicted performance.

1 Introduction

Messages of a variety of types occur naturally in parallel and distributed applications. The most obvious variation is the amount of the data transferred in a message. The data size is likely to be small for control messages amounting to no more than a few bytes, *e.g.* when conveying a simple request or an acknowledgment. On the other hand, block data transfers on the order of many kilobytes of data are also common, especially with coarse-grain parallel applications or when supporting a parallel file system. Another variation concerns the location of the message data. In general, for the smallest sized messages, the processor registers are the source and desired destination. As message size increases, caches are the source and desired destination. For really large messages, main memory DRAM becomes the most likely repository of source data, and the desired destination. The most significant performance measure also varies with message size. Low per-message processor overhead and communication latency are significant for small messages. As message size increases, maximizing bandwidth and minimizing amortized processor overhead become increasingly important. Thus, message size affects engineering design tradeoffs.

Cache-coherent distributed shared memory (CCDSM) protocols are examples of parallel programs that require a variety of message types. Experimental flexibility in protocol implementation is important because protocols are still an active area of research. Consequently, many CCDSM machines implement the protocols in software on embedded processors which communicate via message passing. In CCDSM protocols, requests for a cache-line, invalidations and acknowledgments are small register-based messages, containing only an address, and a message type identifier. A cache-line data transfer from one node to another involves medium sized messages. Entire pages of data may be migrated as program locality changes, giving rise to large messages.

While few will disagree that a spectrum of message passing sizes is needed to support general parallel processing, most available parallel machines today provide only a single message passing mechanism. For messages that are larger than the native message size, software has to packetize, send and reassemble the message, thereby increasing processor overhead and latency. Messages that are smaller than the native message size are often aggregated into larger messages in order to amortize the overhead; such aggregation, however, increases the granularity of parallelism. Though a single message passing mechanism can emulate message passing of all sizes, it is clear that *a single mechanism cannot do so efficiently.*

This paper describes the message passing mechanisms of the MIT StarT-Voyager machine. StarT-Voyager is designed to support efficiently a complete spectrum of message passing requirements by providing four message passing mechanisms – Basic, Express, Tag-On, and DMA messages. The design facilitates protected communication among multiple users and provides the underlying support for coherent distributed shared memory[13].

The following section discusses StarT-Voyager's architecture. This is followed by a description of the message passing mechanisms (Section 3), and the multitasking support in StarT-Voyager (Section 4). Next, Section 5 presents some estimated performance numbers. Section 6 revisits StarT-Voyager's design, discussing possible changes for multiprocessor nodes before we conclude in Section 7.

2 Architecture Overview

The network interface 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. Examples of processors with integrated network interfaces include transputers[45], the iWarp systolic processor[6, 7], dataflow processors like Monsoon[16] and the EMC-R[38], and hybrid processors such as the MIT MDP[17], M-machine[18], and the MIT/Motorola 88110MP[36]. Currently, market forces and engineering effort dictate microprocessor design, making it extremely difficult to take this approach within a commercial microprocessor. The MIT Alewife[1] is an example of a network interface on the L1 cache interface, and the design for StarT-NG[12] proposed one on the L2 cache interface. The tight integration of caches into microprocessors is making this approach inapplicable to most commercial microprocessors. In reaction, placing the network interfaces on I/O buses, which are designed to accommodate third party devices, is currently a popular approach. Commercial examples include machines like the IBM SP-1[22] and SP-2[2, 40], and plug-in extensions like Myricom's Myrinet[5] and DEC's Memory Channel[19], while research efforts include Princeton's SHRIMP[4] and MIT's StarT-Jr[20]. The chief drawbacks of this approach are the difficulties encountered when implementing aggressive cache-coherent distributed shared memory, and lack-luster message passing performance for fine-grain communication. To overcome these deficiencies, StarT-Voyager's network interface is located on the memory bus. This approach is used in machines such as the Stanford DASH[29] and FLASH[26], TMC CM-5[42], Cray T3D[25], Wisconsin Typhoon[37], HP-Convex Exemplar[14, 9], Sequent NUMA-Q[39] and Utah Avalanche[41][1]. These machines are further divided into two groups: the CM-5 and T3D have interfaces that do not make use of a cache-coherent memory bus, while the rest, including StarT-Voyager, have interfaces designed to utilize the cache-coherent support on their memory buses.

StarT-Voyager is a parallel system constructed by connecting a collection of commercial SMPs with a fast network which interfaces to each SMP's coherent memory bus via a Network End-

[1]Several of these machines support either coherent shared memory or message passing in hardware but not both.

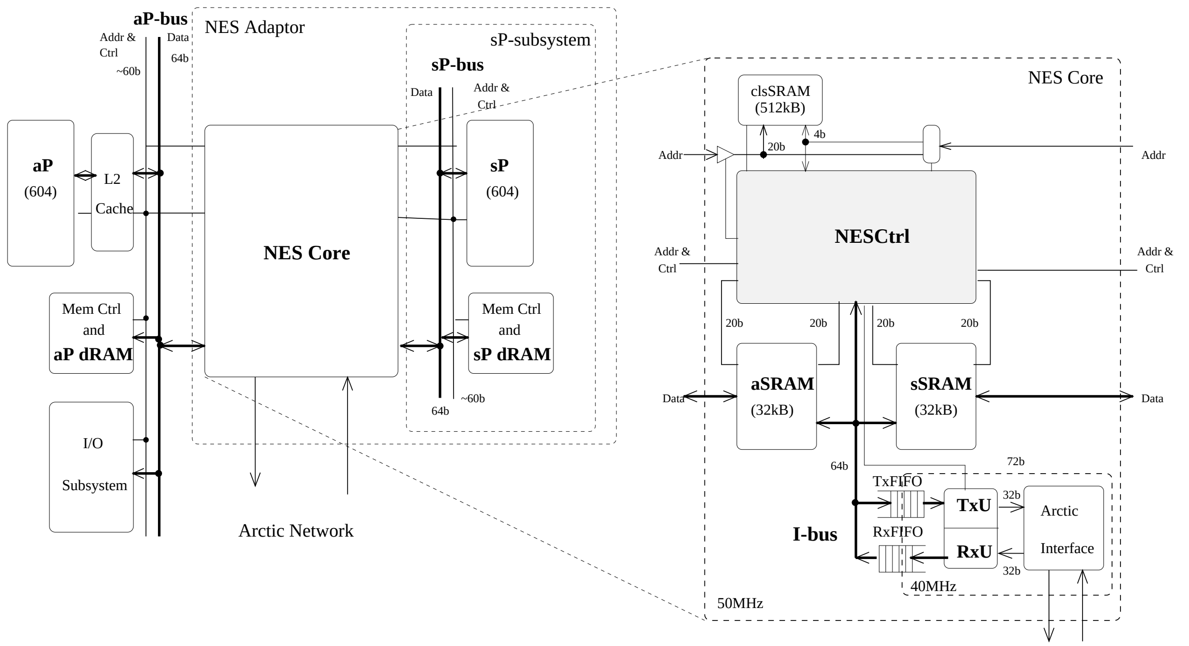

Figure 1: A StarT-Voyager site.

point Subsystem (NES) card. Each SMP, referred to as a *site*, is a desktop-class, commercial, dual-PowerPC 604 SMP. It uses a typical PC/workstation class motherboard with the usual paraphernalia of memory controller, DIMM slots, PCI bus and bridge chip, etc., but has space for two processor cards. Each processor card contains one 604 processor and an in-line L2 cache. This organization facilitates replacing one of the two processor cards in each SMP site with an NES card, effectively making each site into a uniprocessor system (extensions to our design for multi-processor SMP sites are described in Section 6). 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 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

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

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 appears similar to the aP complex. The sP subsystem, however, has additional features. 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). 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 message queues.

The NESCtrl, a small ASIC, implements all the hardware control functionality. It also holds 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 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 3.3) 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.

We plan to build a 32-site StarT-Voyager system by the summer of 1997, and a smaller 4-site prototype a few months earlier. The design and engineering specification of StarT-Voyager has been completed (July 1996) and implementation, currently underway, is divided into three parts: the Arctic network, the NES card, and the NES daughter card which encompasses the TxU, RxU and the Arctic interface. The Arctic router chip is expected to tape out by September 1996 and an Arctic network is being constructed for the StarT-Jr[21] system to be operational in the Fall of 1996. The NES card is made up of off-the-shelf parts except for the NESCtrl chip which is being designed by a small team of students and staff.

3 Message Passing Mechanisms

As each of StarT-Voyager's four message passing mechanisms has its own performance criteria, each has its own unique design tradeoff decisions. Although the particular microprocessor and the exact low-level implementation details ultimately dictate the performance, there are many universal design choices. Many depend on the architecture of today's SMPs and are unlikely to change drastically in the near future. StarT-Voyager's message passing design has benefited from the Active Message work at Berkeley[44, 32, 31, 15] and Cornell[43, 10]. Their implementation and evaluation of Active Message on various platforms illuminates the efficiency and limitations of various message passing designs. After presenting several common design considerations in the next section, each StarT-Voyager message type is examined individually.

3.1 Considerations when Interfacing to Commercial Microprocessors

Since memory mapping is the only way to interface the NES to most commercial microprocessors, several characteristics of microprocessor buses must be considered when designing a message passing interface. One is that virtually all modern microprocessor buses are optimized for cache-line burst transfers. In the 60X bus[3], a burst transfer allows a cache-line (32-bytes) of data to be transfered in 6 bus cycles[3], as compared to the 24 bus cycles needed to do 8 uncached 4-Byte transfers[4]. This difference is accentuated in StarT-Voyager by the 3:1 processor clock to bus clock ratio. Aside from superior bus occupancy, burst transfers also use processor store-buffers more efficiently, reducing processor stalls due to store-buffer overflow.

Given these characteristics, messages, except for the shortest ones, are more efficiently implemented by mapping the message buffer space as cacheable memory. Since a cache-line can be displaced by another cache-line wanting to use the same physical cache entry, a message passing interface that uses cacheable message buffers must use some scheme other than a bus transaction event for transferring control information from an application program to the NES. A simple option is to organize the message buffer space as a circular queue, and employ *uncached* producer and consumer pointers for coordination. Any change in these pointers is taken as a control signal. Another option is to use a full/empty-bit in each message buffer for coordination.

In order to increase the amount of data transmitted by each memory operation, it is possible to use *address bits* to pass data from the processor to the NES[23]. Specific lower-order (to minimize impact on TLB entry utilization) address bits are viewed as data bits by the NES while the higher-order bits are constant to mark this special region of memory. When the NES sees memory operations to that region of memory, it extracts the additional data from the address bits. If data transfer to/from memory on the NES is involved, the NES must generate internal address since the bus address used does not point to a specific memory location. The amount of data that can be carried by address bits is small, 10 bits to fit into one 4-KByte page if accesses are restricted to 32-bit aligned accesses, but can be useful. This technique is mostly useful for uncached accesses rather than cached accesses, since cache-lines are large compared to the amount of data that can be carried on the address and cache thrashing is likely to occur. An additional bonus of using uncached bus operations is that they have a one-to-one mapping to memory operation instructions, allowing uncached accesses to initiate NES actions.

Cacheable memory requires the maintenance of coherence between the processor and NES. In the case of message queues, both the processor and the NES read and write the space, albeit in a fixed orderly fashion. Coherence can be maintained either by the processor with explicit instructions to flush specific cache-lines or by the NES issuing coherency operations on the bus, which we call *Reclaim*. Both have their disadvantages. Processor-issued `flush` instructions, which completely removes cache-line from cache, and `clean` instructions, which writes-back dirty data but keeps an Exclusive copy, are expensive to execute, requiring anywhere between 10 to 20 processor cycles on the 604. Having the NES maintain coherence reduces processor overhead, but adds complexity and overall latency. Ultimately, the choice depends on whether processor overhead or latency is more critical to an application. Some of these issues concerning cacheable network interface was studied by Mukherjee et al.[34].

A second consideration is whether the coherent bus supports *cache-to-cache transfers*, where a cache with modified data is able to supply it directly to another cache without first writing it

[3]This assumes a slave device that is able to accept or supply data at that rate. NES SRAM will be able to match that speed, but not aP-DRAM.

[4]A few microprocessors are able to aggregate uncached memory writes to *contiguous* addresses into larger units for transfer over the memory bus, but the 604 is not one of them.

back to main memory. Though the answer does not affect logical correctness of a message passing interface design, it influences the latency, bus occupancy and main memory DRAM bandwidth utilization of some designs. For example, if cache-to-cache transfers are supported, a message passing interface with message buffers that ultimately reside in main memory can actually achieve direct data transfer between the processor cache and the NES. This gives it the same direct data transfer path as a design that places all the message buffers on the NES, while utilizing only a small memory on the NES as a cache of message buffers which are backed up by the larger, cheaper main memory. Unfortunately, the 604 does not support cache-to-cache transfers.

The microprocessor's memory model also impacts the design. In particular, weak memory models found in many modern microprocessors including the PowerPC family allow memory operations to get out-of-order. Consequently, any message-launch memory operation might get reordered so as to appear on the bus before the corresponding message-compose memory operations appear. All processors that have a weak memory model provide a *barrier* operation (`sync` in PowerPC processors) which enforces ordering but unfortunately, it is generally expensive, requiring between 12 and 20 cycles on a 604.

With these characteristics of microprocessor memory system in mind, we now describe StarT-Voyager's four message passing mechanisms. All the message passing mechanisms are directly accessible from user code without violating protection in a multitasking environment (see Section 4). The different mechanisms are statically mapped into separate physical address regions. Packing data bits into addresses is employed extensively in our design. For example we specify message queues, and encode message destinations and payload data using address bits. Except for DMA messages which are not explicitly received by application software, the NES automatically appends to every out-going message a *logical source identifier*, which the receiver can use as the destination for a reply. All message destinations specified by user code are logical destinations which are translated by the NES (See Section 4 for more details on logical destination names.)

3.2 Basic Messages

StarT-Voyager's Basic Message mechanism provides protected access to Arctic's native packet transfer service. With a 32-bit header specifying the logical destination queue and other options, and a variable payload of between four and twenty-two 4-byte words, a Basic Message is ideal for communicating an RPC request, or performing a small scatter/gather transfer. Microprocessors are optimized for communicating data to/from registers and caches.

The Basic Message interface consists of separate transmit and receive queues, each with a cacheable message buffer region, and *uncached* producer and consumer pointers for exchanging control information between the processor and the NES. The message buffer region is arranged as a circular FIFO with the whole queue visible to application software, enabling concurrent access to multiple messages. The producer and consumer pointers are mapped into uncached space to reduce latency. An uncached write pointer update immediately triggers NES processing. We do not implement cached producer and consumer pointers because of attendant design complexity and limited benefits.

The four steps of composing a Basic Message are shown in Figure 2. The Basic Message transmit code first checks to see if there is sufficient buffer space to send the message. The software maintains a copy of the producer and consumer pointers (P, C in top half of figure) to minimize reading them from the NES. This copy of the consumer pointer is updated again only when it indicates that the queue is full. By then, space may have freed up since the pointer was last read from the NES. The NES may move the consumer pointer at any time if it launches any messages, an event which occurrs in our example between steps 1 and 2. If the transmit code finds that enough buffer space

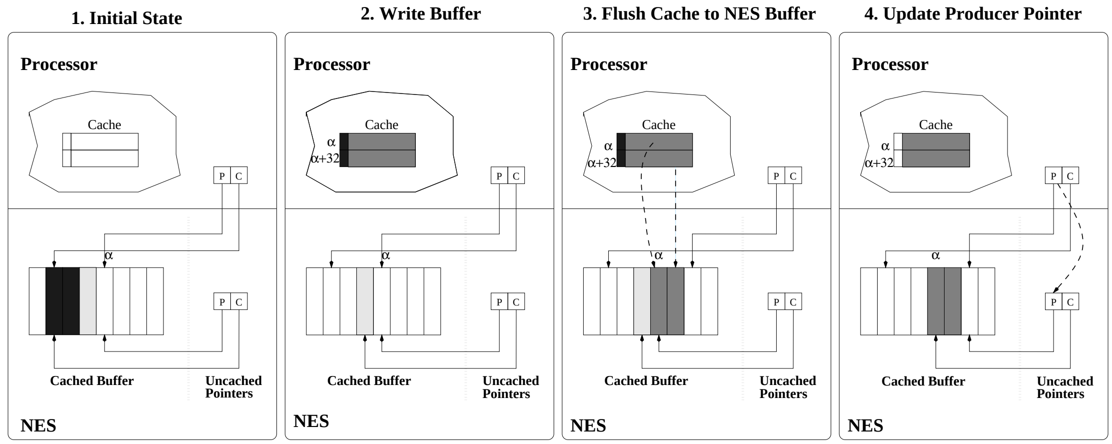

Figure 2: Sending a Basic Message

is available, the message is written into the cached buffer space, with the write effectively going to the processor cache (step 2). Next, the processor issues `clean` instructions to write the modified cache-lines to the corresponding NES buffer locations (step 3). Due to PowerPC's weak memory model, a barrier instruction is required after the `clean` instructions and before the producer pointer is updated via an uncached write. This write (step 4) prompts the NES to launch the message, after which the NES frees the buffer space by incrementing the consumer pointer. With NES Reclaim, the NES issues `clean` bus operations to maintain coherence between the processor cache and the NES buffers, eliminating the need for the processor to issue `clean` instructions (but not the `sync`) in step 3. In this case, the pointer update will cause the NES to reclaim the message, then launch it.

The interface is designed primarily for message reception by polling, although an application can request for an interrupt upon message arrival. This choice is available to a receiver on a per receive queue basis, or to a sender on a per message basis. When polling for messages, an application compares the producer and consumer pointers to determine if there is any message. Messages are read out directly from the message buffer region. Coherence maintenance is again needed so that the application does not read a cached old message. This can be done either explicitly by the processor with `flush` instructions or by NES Reclaim.

A unique aspect of the Basic Message buffer queue is its memory allocation scheme. Buffer space in this queue is allocated in cache-line granularity and both the producer and consumer pointers are cache-line address pointers. Allocation in smaller granularity is undesirable because of the coherence problem caused by multiple messages sharing a cache-line. The other obvious choice of allocating maximum-sized buffers was rejected because it does not work well with either software prefetching of received messages, or NES Reclaim. The main problem is not knowing the size of a message until the header is read. Therefore, both prefetching and a simple implementation of NES Reclaim must either first read the header, and then decide how many more cache-lines of data to read, or blindly read all three cache-lines. The former introduces latency while the latter wastes bandwidth. With cache-line granularity allocation, every cache-line contains useful data, and data that is fetched will either be used for the current message, or subsequent ones. Better buffer space utilization is another benefit of this choice.

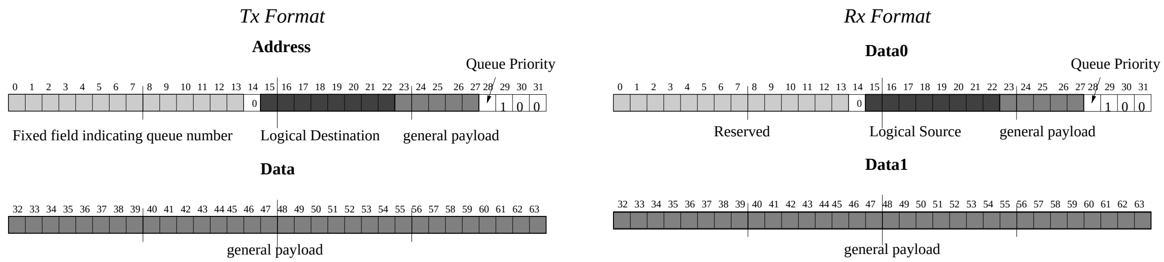

Figure 3: Express Message Formats

3.3 Express Messages

Messages with a minimal amount of data are common in many applications for synchronization or communicating a simple request or reply. Basic Messages, with a cached message buffer space, is a bad match because the bandwidth of burst transfer is not needed while the overhead of coherence maintenance, weak memory model, and explicit handshake remain. Express Messages are introduced in StarT-Voyager to cater to such small payloads by utilizing a single uncached access to transfer all the data of a message and thus avoid the overheads of Basic Messages.

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 packs the transmit queue specifier, message destination and 5 bits of data into the *address* of an uncached write. The NES automatically transforms the information contained in the address into a message header and appends the data from the uncached write to form an 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 larger (virtual and physical) address space and can also have a detrimental effect on TLB 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] may be employed to mitigate this problem but depends on both processor architecture and OS support.)

Unlike Express Message transmits, address bits cannot be used to convey message data to the processor when receiving an Express Message. 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. Like an Express Message transmit, an Express Message receive can be accomplished with a 64 bit uncached read into an FPR. The data is subsequently move into GPRs via (cache) memory[5]. Alternatively, two 32 bit loads into GPRs can be issued to receive the message.

Unlike Basic Messages, 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[6]. 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

[5]PowerPC does not support 8 Byte load into a pair of GPRs nor direct data transfer between a FPR and a GPR.
[6]This involves asserting the guarded bit in PowerPC architecture.

Active Messages[44], the Empty Message can be treated as a legitimate message with a "no action" message handler. Unlike for receiving, the use of producer and consumer pointers is required for transmit queues to ensure that they do not overflow.

3.4 Long Message: DMA

StarT-Voyager's DMA support is an efficient mechanism for moving contiguously located data between the memory of one site and that of another. It resembles the usual DMA facility in that an aP can unilaterally achieve the movement without the other site's aP's involvement. This is in contrast to data movement effected through normal message passing, where matching request and reply messages are needed. The DMA facility can be though of as a remote memory get or memory put operation.

StarT-Voyager's DMA facility is designed to be "light weight" so that it can be profitably employed for relatively small sized transfers. To reduce per-transfer aP overhead, the design decouples page pinning from transfer request. The cost of the former can be amortized over several transfers if traditional system calls are used to pin pages. Alternatively, StarT-Voyager can support system software designs where the aP implements VMM cooperatively with the sP, allowing the latter to effect any necessary page pinning. User code initiates DMA transfers through an interface similar to a Basic Message transmit queue. The *logical* source or destination, source data location, destination data location, and length are specified in a DMA request message to the sP, which together with the other sP involved in the transfer, performs the necessary translation and protection checks before setting up DMA hardware to perform the transfer. An option for messages to be sent upon completion is also available.

Hardware support consists of a DMA Source Engine, a DMA Destination Engine, and a *DMA channel*. The sP's at the source and destination sites exchange a series of messages to establish the physical addresses of the source and destination memory locations. Then, both the source and destination sP must perform local initialization actions before the transfer can happen completely by the hardware. The sP at the destination site allocates a DMA channel for the transfer, and also initializes the channel's *packet counter* in its Destination Engine to the expected number of packets[7]. The sP at the source site issues a command to the DMA Source Engine specifying both the starting source and destination physical memory addresses, the amount of data to transfer, and the DMA channel number.

Once set up this way, the Source Engine reads data from the aP's DRAM, packetizes it along with the destination physical memory address and the DMA channel number, and launches the packets into the network. Upon receiving such a packet, the Destination Engine writes the data into its aP DRAM at the specified address, and decrements the DMA channel's counter. When the count reaches zero, the destination sP is notified of DMA completion. The design requires no per-packet handshake. Each sP command can request up to 4k Bytes (a page size) of data to be transferred in this autonomous way; beyond that, the source DMA engine must be issued separate commands for each page transfer[8] although several commands can be queued at the DMA Source Engine.

[7] As an optimization, re-initialization of a DMA channel's count can be done remotely from another sP via a direct message to the Destination Engine. This removes the latency of setting up a channel in cases where repeated, relatively small DMA transfers are common in an application.

[8] The channel count at the destination can span multiple pages.

3.5 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 NES SRAM's. It is especially useful for implementing coherent shared memory protocol, and for multi-casting a medium sized message. 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 32 Bytes (or 64 Bytes) of message data can be found. 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 which 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 protocol 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, for example 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 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.

3.6 One-poll

In order to minimize the overhead of polling from multiple receive queues, StarT-Voyager introduces a novel mechanism, called *One-poll*, which allows one polling action to poll simultaneously from a number of Express Message receive queues as well as Basic Message receive queues. A single uncached read specifies in its address the queues to poll from and obtains the highest priority Express Message. If the highest priority non-empty queue is a Basic Message queue, a special Express Message that includes the Basic Message queue identifier and its queue pointers is returned. If there are no messages in any of the polled queues, a special Empty Express Message is returned. One-poll is useful to user applications, most of which are expected to have four receive queues: high priority Basic Message, low priority Basic Message, high priority Express and Tag-On Message, and low priority Express and Tag-On Message. The sP has nine queues to poll. It is clear that the one-poll mechanism will dramatically reduce polling costs.

4 Protection and Multiuser Concerns

StarT-Voyager was designed to support a multiuser, multitasking, loosely gang-scheduled environment. To achieve this goal, StarT-Voyager employs a combination of protection and translation

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

mechanisms and provides multiple message queues at each site.

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[9]. No queue is assigned to two independent processes simultaneously. 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. Furthermore, a process is allowed to send messages to other processes (i.e., enqueue messages in remote receive queues) only if it is so specified in a predetermined *Destination Table*.

A message packet header must specify both the destination site and destination queue at the site. Rather than addressing physical sites and queues, the application code uses a logical destination specifier, which identifies an entry in the Destination Table (see Figure 4). The NES translates logical destination specifiers to the appropriate site/queue pair. Since the translation is also a function of the local transmit queue, only specific transmit queues can send messages to specific receive queues[10].

The NES also attaches a source identifier to each outgoing message. The destination process can reply to a message using the source identifier. For symmetry in translation the source identifier must look like a logical destination specifier at the destination. Such a specifier is also kept in the Destination Table to facilitate attaching of the source identifier (see Figure 4). 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. The Destination Table is configured through a separate NES address space and should be accessible only to system software. Thus an application must be granted permission to access remote queues prior to initiating any communication.

With this support, efficient protected multitasking can be implemented easily. A running parallel job can be suspended and "swapped out" by suspending and swapping out each of its constituent processes independently. Since each receive queue is assigned to at most one process, messages can arrive for a process that is swapped out; swapping out a parallel process requires no global coordination, though it may impact performance. Such freedom dramatically reduces the parallel job swapping time needed by traditional MPP's. Traditional MPP's such as the CM-5[28] support 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.

Logical destination translation also enables processes to migrate from one site to another and

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

[10] 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.

from one set of receive queues to another. Moving processes to other sites and receive queues is useful if a parallel job must be moved to a smaller space partition, say to adapt to the failure of a site. Several processes of the same job can be mapped to the same site, each with its own set of message passing queues.

Other machines use a Parallel Job Identifier (PJID) which is added automatically to every outgoing message and verified at the destination[36]. Such schemes define 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 logical destination mechanism supports such non-uniform communication patterns in a protected fashion.

The large number of queues at each 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*. 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. There are several ways to implement non-resident queues. One method is to map the buffer space into the aP's DRAM and the producer-consumer pointers into sP service space. Messages can continue to be received into and sent from non-resident queues with the sP's assistance. The sP implements the transition of a queue between resident and nonresident status when directed by the OS.

5 Projected Performance

This section presents estimates of resident queue performance. These numbers are hand generated, using C code sequences which were compiled into assembly code, hand optimized and then hand scheduled for a PowerPC 604 processor. Only one general purpose register is assumed to be always available for message queue information such as counters and buffer addresses; otherwise, they are assumed to be in L1 cache. Bus latency of accesses to the NES is counted and message data is assumed to be sent from and received into processor registers for all message types except DMA. For DMA, data at both source and destination sites is assumed to reside in main memory. NES performance numbers are derived from a low-level simulator used by hardware designers, assuming that aside from the measured activity, the system is quiescence. In StarT-Voyager, the aPs and sPs will run at 150MHz, the system bus and most of the NES will run at 50MHz, and the Arctic network will run at 40MHz. Three performance metrics are presented: processor overhead, communication latency, and peak bandwidth.

5.1 Processor overhead

Processor overhead is the number of cycles that the sending or receiving processor spends to compose and launch or receive a message. The number of cycles is counted from the time each routine starts executing to the time it executes the last instruction. The processor pipeline and bus interface buffers are assumed to be empty when each routine starts executing. The receive overhead is divided into two parts: definite overhead, and load latency. Definite overhead is the receive overhead that must be incurred. Load latency is the latency of reading data from the NES card which translates

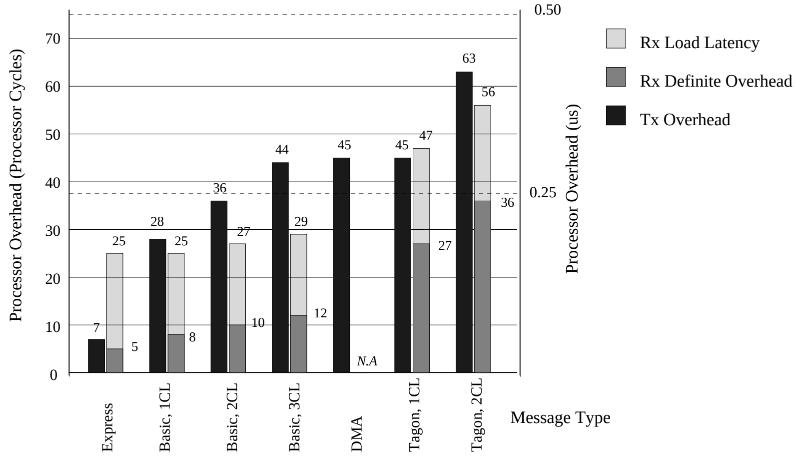

Figure 5: Processor overhead for the different message passing mechanisms, assuming NES Reclaim is used to maintain message buffer coherence for Basic Messages.

into processor overhead if the data is used immediately[11]. This overhead can be reduced with software prefetching, and in the very best case, contributes zero processor overhead. Figure 5 shows the processor overhead for sending and for receiving all four types of messages[12].

Processor overhead for Basic Message does not include the cost of reading the consumer pointer of a transmit queue, or to write the consumer pointer of a receive queue because these operations can be amortized over several messages with Basic Message's interface design. The overhead reported for DMA does not include that of pinning pages and indicates the processor overhead per DMA transfer when pinned pages are used repeatedly. DMA has no receive component and thus no corresponding overhead.

Processor overhead for Basic Messages reported above assumes NES Reclaim is used. If aP software is maintaining coherence, the processor overhead increases significantly as shown in Figure 6. The lower processor overhead of NES Reclaim comes at a small cost of slightly increased latency and slightly lower bandwidth, but the difference is under 10% in all cases.

5.2 Communication Latency

Communication latency measures the total time for message communication, starting from the time the message transmit routine begins execution to the time the first word of the message is in a register at the destination (or the memory copy is completed in the case of DMA). The reported numbers assume no resource contention in the NES or the Arctic network. Communication latency is divided into five parts: processor transmit, NES transmit, Arctic network, NES receive and processor receive latencies as described in Figure 7, which also shows the latency for communicating with messages of various types. Because the number of network hops between source and destination sites depends on their relative locations on the Fat-tree, numbers for the two extreme cases: one hop and nine hops (maximum number of hops for StarT-Voyager's 32-site system) are presented.

[11] Pipelined, out-of-order, superscalar and speculative execution can only sustain the pipeline for a very small number of cycles, insufficient to cover the expected latency of reading from the NES.

[12] In this section whenever StarT-Voyager's performance is referenced, it should be read as "is expected to be" as opposed to "is".

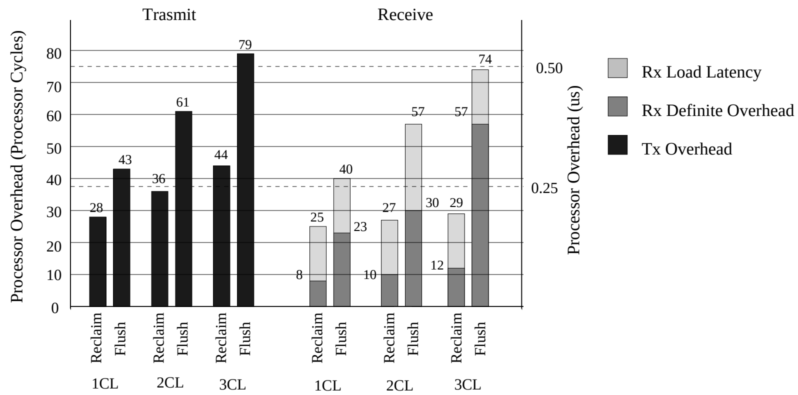

Figure 6: Comparison of Basic Message processor overhead with message buffer space coherence maintained by software (Flush), and by NES hardware (Reclaim).

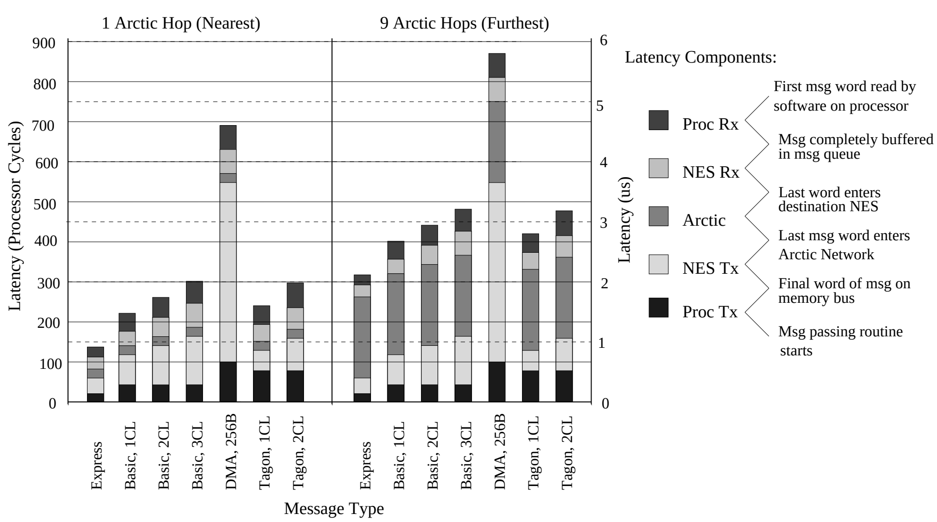

Figure 7: One-way communication latency for StarT-Voyager's message passing mechanisms.

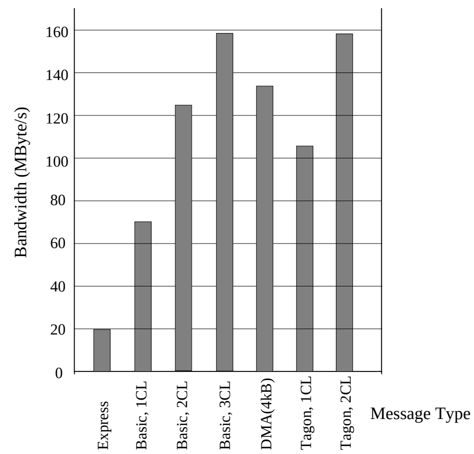

Figure 8: Comparison of peak communication bandwidth that each message passing mechanism can deliver.

Total latency is lowest for Express Message with a $1\mu s$ latency for nearest neighbor communication. Except for DMA, all message types take under 3.5 μs even for messages that have to travel the furthest distances. The DMA latency for 256-byte transfer is also shown. The number assumes that resetting of DMA channel counter is done remotely from the sender sP. The reasonably low DMA latency, together with the low processor overhead to initiate DMA (presented in the previous section) makes it effective to employ DMA for relatively modest-sized transfers. When a message has to go all the way across the network, Arctic network's latency dominates the latency for all non-DMA messages. With future router chips expected to have a higher degree[13], however, network latency should decrease, making the other latency components which are directly dependent on interface design more important.

5.3 Peak Bandwidth

Peak bandwidth is derived by assuming that the source and destination processors are doing nothing but sending or receiving messages as fast as they can. Furthermore, a site is either sending or receiving messages but not both. It takes into account resource contention between successive invocation of the same message passing mechanism. Figure 8 shows the peak bandwidth that each message passing mechanism can deliver.

Express Message, limited by its less efficient use of memory bus bandwidth, has the lowest bandwidth but nevertheless achieves 20 MBytes/sec bandwidth. The highest bandwidth of over 150 MBytes/sec is achieved with the largest Basic Message. This is counter intuitive because one would expect DMA to deliver the highest bandwidth. It is partly an artifact of the assumptions used for generating these numbers: the data source and destination for Basic Messages is assumed to be in the processor registers. In practice, this will not be the case unless the same data is sent on every message, a useless activity. In contrast the bandwidth offered by DMA can be gainfully employed. The limiting factor for Basic Messages is how fast cached message data can be transferred over the system bus, either with processor Flush instructions, or NES Flush bus transaction. DMA

[13] Arctic was initially designed as an 8x8 switch, but due to packaging limitations, it was scaled back to a 4x4 switch.

Machine	processor speed (MHz)	Interface Location	one-way nearest neighbor latency (μs)	Send Overhead (μs)	Receive Overhead (μs)	Peak Bandwidth (MByte/sec)
StarT-Voyager (raw)	*150*	*memory bus*	*0.9*	*0.05*	*0.17*	*158*
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)[15]	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)[21]	120	I/O bus	27	1.5	1.1	7
Sparc20+ATM (AM)[11]	50	I/O bus	33	3		14
SP-2 (AM)[11]	66	I/O bus	25.5	4		34
Sparc20+Myrinet(AM)[15]	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 9: Message passing performance of several representative systems. StarT-Voyager's numbers are estimated, and are raw numbers that do not include the overhead of additional functionalities that are present in a typical message passing library.

performance is limited by the rate at which the DMA Send Engine is reading and sending data packets.

5.4 Comparison with Other Machines

Figure 9 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 9 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, 15, 11, 21, 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.

Processor overhead for message passing communication is very low on StarT-Voyager, and this scales down to the smallest message with only one word of data. Among the other machines, the lowest overhead for small message, both in terms of number of processor cycles, and wall-clock time, is achieved on a CM-5, taking about 50 (33MHz) processor cycles (1.5μs) to either send or receive a four 4-word message. StarT-Voyager is able to better CM-5 both in processor count and wall-clock time. With a Basic Message, the wall-clock time overhead of .17μs is almost 9 times better than CM-5's. In terms of processor clock, which is a better comparison since CM-5 has a relatively slow processor, StarT-Voyager's Basic Message takes only 26 processor cycles, half that of CM-5.

StarT-Voyager's latency is also significantly better than other machines, only 1/6th that of CM-5, which has the lowest latency them. Communication latencies for machines with memory bus network interfaces – Meiko CS-2, Paragon and T3D, are similar, in the vicinity of 10μs. I/O bus based interfaces are slower, the fastest, achieved by Active Messages over Myrinet, has a latency of 15μs.

StarT-Voyager delivers very high message passing bandwidth, roughly an order-of-magnitude improvement over most existing machines. Among the other machines, only Paragon achieves a similar bandwidth of 145 MByte/sec, but requires messages of almost 4-KBytes to reach *half* of that bandwidth. In contrast, StarT-Voyager achieves *the peak bandwidth* with packets carrying only 88 Byte of data payload. Most of the other machines achieve peak bandwidth of only 10-20 MByte/sec, with SP-2 reaching 35MByte/sec.

6 Design Modifications for Multiprocessor Host System

When a host SMP has multiple aPs and its coherent memory bus supports cache-to-cache transfer of data, both characteristics of emerging high-end SMPs, the StarT-Voyager design can be improved. New issues in this environment are: (i) atomicity of concurrent accesses to a shared message queue by multiple threads; and (ii) the need to support more resident queues because different jobs using separate message queues may be executing on different aPs simultaneously.

Several threads executing in parallel can share a message queue only if allocation of transmit queue buffers and received messages is done atomically. In addition, each thread should be able to independently launch a message, or free a message buffer. StarT-Voyager's existing Basic Message interface does not meet these requirements. To achieve atomic allocation with that interface, a mutex lock is needed for each queue, adding significant overhead. The design's use of FIFO queues requires message launch to follow the buffer allocation order for a transmit queue, and buffer deallocation to follow the message arrival order for a receive queue. This requires coordination between threads and introduces dependence. For example, should a thread obtain a transmit buffer and not utilize it for a while, another thread which obtains a transmit buffer after it will not be able to launch its message.

In contrast, the Express Message interface, which relies on a single uncached read or write to perform a complete operation on a message queue, meets the atomicity and independence requirements. We can use this technique to develop a new interface for Basic Messages which associates with each message queue two FIFOs of buffer pointers: one for pointers to free buffers, another for pointers to full buffers. Consumers pop buffer pointers off the Full Buffer FIFO, dereferencing it to get to the message. To free a buffer, a consumer pushes its pointer into the Free Buffer FIFO. In a similar fashion, producers obtain buffers by popping buffer pointers off the Free Buffer FIFO. After writing message data into the buffer space, the pointer to the buffer is pushed into the Full Buffer FIFO. A distinguished null-pointer value is returned when popping from an empty FIFO. This design achieves both atomicity and independence. It comes at the price of: (i) an indirection to get a send or receive buffer, (ii) more complex hardware and extra memory for the buffer pointer FIFOs, and (iii) no opportunity for aggregating accesses to the NES. This interface can be extended to support Express Messages too by extending the transmit Full Buffer FIFO to include a small amount of data. In order to minimize the number of bits to specify the buffer address, they are specified as a offset from a known per-message queue base address. This combined interface allows short sized messages to be sent with the efficiency of Express Messages, while presenting an interface for medium sized messages that works with concurrent accesses from multiple threads. StarT-Voyager incorporates certain elements of this interface in Tag-On Messages with the difference that NES support for Tag-On Messages does not include a hardware FIFO to read free buffers; however, it can be emulated by sP software.

To support more resident queues, one method is to scale up in a brute force manner, either increasing the number of queues supported in each NES, or use multiple NES's. Changes to the micro-architecture can, however, result in better utilization of a smaller amount of SRAM. Instead of using SRAM as straight forward queues, it is used as staging buffers for transmission of messages, and as a cache for received messages. The final storage space for all queues is main memory DRAM. Out-going messages are read by the NES into the staging buffers, before it is send out into the network, while an arriving message is placed into the Received Message Cache unless the latter is full, in which case the message is written into DRAM. Cache management policy can be customized to the structured usage pattern by adopting strategies like deallocating data in the Received Message Cache after it has been read and the queue buffer space freed by application software. Destination translation tables can also be placed in DRAM, with only cached copies in the NES. This design blurs the distinction between resident and non-resident resources. Assuming that the coherent memory bus supports cache-to-cache transfers, the design can achieve the same data-transfer path as the straight forward queues-in-SRAM design of StarT-Voyager. Though such a design uses NES buffers more efficiently, it is also more complex than the current StarT-Voyager design.

7 Conclusions

StarT-Voyager is designed to be a scalable, general-purpose parallel system which can effectively support a mixed workload of sequential, distributed and parallel applications in a multitasking environment. It uses unmodified commercial SMPs as building blocks to keep development cost low, and capitalizes on the rapid performance improvement of commercial systems. StarT-Voyager combines the best features of traditional MPPs, hybrid message passing/shared memory machines and more recent work on NOWs, and introduces new mechanisms in areas where existing machines have fell short.

While adopting the concept of direct user-level access to message passing hardware from MPPs like the CM-5, StarT-Voyager goes much further in (i) providing sufficient support to cover a wider range of communication needs efficiently, and (ii) developing a simple but flexible protection scheme. These are traditional MPPs' Achilles heels which have confined them to a niche market. From NOWs, StarT-Voyager borrowed the idea of employing commercial systems as building blocks, and the vision of a general purpose parallel system that is not only suitable for executing parallel programs, but is also efficient for running traditional sequential and client-server distributed applications. StarT-Voyager is much more aggressive in pushing for higher performance under this approach, and includes hardware supported cache-coherent shared memory[13].

Although StarT-Voyager provides many mechanisms, most of them require only incremental additions to the basic hardware. The different message types all rely on the same hardware structure of FIFO queues with producer and consumer pointers. This is exposed to applications in the Basic Message interface. To support Express Message, the NESCtrl chip adds a veneer of hardware that manages the producer and consumer pointers so that the queue now appears as hardware FIFOs to applications. The DMA engines' ability to read and write aP DRAM are already present in order to support cache-coherent distributed shared memory[13]. Through careful interface design and factoring out common subcomponents, StarT-Voyager is able to offer a range of capabilities without excessive hardware design complextiy. Compared to more custom machines like the Stanford Flash[26], StarT-Voyager has far lower development costs while delivering similar functionalities. StarT-Voyager may have lower coherent shared memory performance than Flash because its sP is less tightly coupled to the aP memory system than Flash's MAGIC chip which is situated at the memory controller location. Nevertheless, because StarT-Voyager handles common cases completely in hardware, the sP is far less loaded than Flash's Protocol Processor, giving StarT-Voyager some performance edge.

Credits and Acknowledgments Boon S. Ang and Derek Chiou are the principle architects of StarT-Voyager. Mike Ehrlich is leading the NES hardware implementation team which consists of Andy Boughton, Chris Conley, Jack Costanza, Dan Rosenband, Handong Wu and Brad Bartley. Andy Boughton designed Arctic and is leading the Arctic implementation team which consists of Jack Costanza and Dan Rosenband. Larry Rudolph is heading the software team. Xiaowei Shen is designing and implementing coherency protocols. We would like to thank Bob Greiner who worked with us on earlier designs and taught us much about micro-architecture design.

References

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

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

[3] M. S. Allen, M. Alexander, C. Wright, and J. Chang. Designing the PowerPC 60X Bus. *IEEE Micro*, pages 42 – 51, Oct. 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, C. Hawblitzel, and T. von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP. In *Proceedings of Supercomputing '96*, 1996. (to appear).

[11] 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).

[12] 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.

[13] D. Chiou, B. S. Ang, L. Rudolph, and Arvind. Coherent Shared Memory Support on StarT-Voyager. CSG Memo 388, MIT Laboratory for Computer Science, July 1996.

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

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

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

[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] 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,).

[40] 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.

[41] M. R. Swanson, R. Kuramkote, L. B. Stoller, and T. Tateyama. Message Passing Support in the Avalanche Widget. UUCS 96-002, Department of Computer Science, University of Utah, Mar. 1996.

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

[43] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Paralle and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado*, Dec. 1995.

[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.