LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Message Passing Support on StarT-Voyager

Computation Structures Group Memo 417
December 8, 1998

Boon S. Ang

Derek Chiou

Larry Rudolph

Arvind
Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square, Cambridge, Massachusetts
{**hahaha,derek,rudolph,arvind**}**@lcs.mit.edu**

To appear in the 5th International Conference on High-Performance Computing (HiPC '98) December 17-20, 1998 Madras, INDIA

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

Message Passing Support on StarT-Voyager

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

Abstract

No single message passing mechanism can efficiently support all types of communication that commonly occur 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 high performance over a wide spectrum of communication types and sizes. Hardware and address translation 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 developed at MIT. This paper describes StarT-Voyager's message passing mechanisms and their predicted performance.

1. Introduction

Messages of a variety of types occur commonly in parallel and distributed applications. The most obvious variation is the amount of data in a message. This 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, transfers on the order of many kilobytes of data are also common, especially with coarse-grain parallel applications. The location of the message data is often correlated to message size. For the smallest-sized messages it is the processor registers. As message size increases, the location can be found lower down in the memory hierarchy: medium-sized messages in caches and large-sized messages in main memory. 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.

Various message sizes are also important for shared memory architectures. For example, cache-coherent distributed shared memory (CCDSM) protocols require a variety of message types. Cache-line requests, invalidations and acknowledgments use small, register-based messages, containing only an address and a message type identifier. A cache-line data transfer involve 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 sizes is encountered in general parallel processing, most available parallel machines do not address this diversity. It is usually the job of software to packetize and reassemble large messages, thereby increasing processor overhead and latency. Small messages either waste bandwidth and latency or are aggregated into larger messages in order to amortize the overhead, which increases the granularity of parallelism. Though a single message passing mechanism can emulate message passing of all sizes, it cannot do so efficiently.

This paper describes the message passing mechanisms of the MIT StarT-Voyager machine. StarT-Voyager is designed to efficiently support a wide spectrum of message passing requirements by providing four mechanisms – Basic, Express, Tag-On, and DMA messages. The design also enforces protected communication among multiple users sharing a machine and provides the underlying support for coherent distributed shared memory. Due to space limitations, the latter is reported in a separate paper[3].

The following section discusses StarT-Voyager's architecture. This is followed by a description of the message

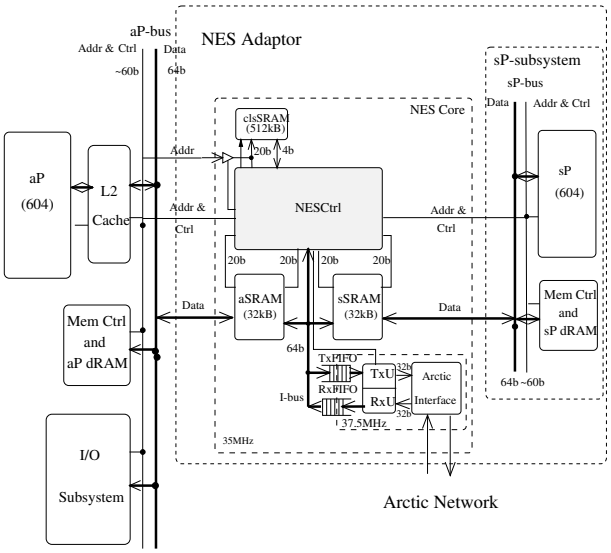

Figure 1. A StarT-Voyager site.

passing mechanisms (Section 3), and the multitasking support in StarT-Voyager (Section 4). Next, Section 5 presents some estimated performance numbers. We conclude in Section 6.

2. Architecture Overview

StarT-Voyager is a parallel system consisting of a collection of commercial SMP's interconnected with a fast network. A Network Endpoint Subsystem (NES) card attached to the coherent memory bus of the SMP provides the interface to the network. Each SMP, referred to as a *site*, is a desktop-class, commercial, dual-PowerPC 604 SMP with a motherboard which accommodates two processor cards. A 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. The remaining 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 [8] network constructed with Arctic router chips[4]; both are designed and implemented at MIT as part of the StarT project. The Arctic network is also accessible with PCI network interface cards desgined and built in two companion projects, StarT-Jr [7] and StarT-X [6].

The NES, designed for both speed and flexibility, services several memory mapped address regions, each providing a different functionality. Its 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 appears almost identical to the application processor (aP) system. Both the aP and sP have access to the same message passing mechanisms, with each having its own set of resources. Two banks of dual-ported SRAM, the aSRAM and sSRAM, provide storage space for message queues.

The sP subsystem, however, has additional features. It observes all aP bus operations to an NES address region 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. Examples of extensions include coherent shared memory and *non-resident message queues* implementation (see Section 4). Interested readers are referred to Ang et al. [2] for discussions of this flexibility and Ang et al. [3] for NES micro-architecture details.

3. Message Passing Mechanisms

Software overhead, latency, and bandwidth are several considerations for message transmission and reception [5]. StarT-Voyager provides four mechanisms that offer performance tradeoffs between those considerations. In addition to message size and location, the communication performance of a system is highly dependent on low-level implementation details some of which depend on the architecture of today's SMP's and are unlikely to change drastically in the near future. After presenting several common design considerations in the next section, each StarT-Voyager message type is examined individually.

3.1. Design Considerations

Several characteristics of commercial SMP architectures influence the design of a network endpoint subsystem (NES) and the choice of communication mechanisms. For most commercial microprocessors and SMP's, memory mapping is the main interface to the NES. Thus, the optimal operating mode of the main system bus, the cache-line transfer mechanism, and the memory model are critically important. The following discusses how they impact the design.

Virtually all modern microprocessor buses are optimized for cache-line burst transfers. Consider the 60X bus[1] used by the PowerPC 604; a cache-line (32-bytes) of data can be transfered in as few as 6 bus cycles compared to 24

bus cycles required for eight uncached 4-Byte transfers[1]. This difference is accentuated in StarT-Voyager by the 4:1 processor clock to bus clock ratio. Aside from superior bus occupancy, the cache-line burst transfer also uses processor store-buffers more efficiently, reducing processor stalls due to store-buffer overflow.

Burst transfers are typically only available for cacheable memory. With message queues read and writen by both the processor and the NES, cache coherence must be maintained between the NES and the relevant processor cache-lines in order to exploit the burst transfer. There are two alternatives: the processor software may execute explicit instructions to flush specific cache-lines or the NES may issue coherency operations. The latter is referred to as *Reclaim*. There are disadvantages to both explicit flush and to reclaim. A processor-issued `flush` instruction, which completely removes cache-line from cache, and a processor-issued `clean` instruction, which writes-back dirty data but keeps an Exclusive copy, are both expensive to execute requiring anywhere between 10 to 20 processor cycles on the 604. Although reclaim reduces processor overhead, it increases hardware complexity and overall latency. Ultimately, the choice depends on whether processor overhead or latency is more critical to an application (see Mukherjee et al.[10] concerning cacheable network interface).

Cache-to-cache transfers, where a cache with modified data supplies it directly to another cache without first writing it back to main memory is another important feature. It does not affect the logical correctness of a message passing interface design, but influences the latency, bus occupancy and main memory bandwidth utilization of some designs. With this feature, message buffers can logically reside in main memory but be burst transferred directly between the processor cache and the NES. Without cache-to-cache transfers, a cache line is first written back to memory, then fetched from memory to the destination. Since the 604 bus protocol does not support cache-to-cache transfers, StarT-Voyager maintains all the message buffers in fast NES SRAM buffers, avoiding the delays in accessing main memory.

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, a message-launch operation might get reordered to appear on the bus before the corresponding message-compose memory operations. To enforce ordering, processors that implement a weak memory model provide a *barrier* operation (`sync` in PowerPC processors); unfortunately, it is generally expensive, requiring between 12 to 20 cycles on a 604.

With these characteristics of microprocessor memory system in mind, StarT-Voyager's four message passing mechanisms are presented. Each mechanism is statically mapped into a separate physical address region. With appropriate virtual-to-physical address mappings, these services can be accessed directly from user code without violating protection. (Section 4 discusses protection issues.)

[1] 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.

3.2. Basic Messages

The basic message passing mechanism of StarT-Voyager consists of enqueuing and dequeuing of basic messages into separate transmit and receive queues. Messages are sent to queues and not to processors. A basic message has two components: a 32-bit header specifying the logical destination queue and a variable payload of between four and twenty-two 4-byte words.

The processor performs four steps to send a basic message (Figure 2, top half). The Basic Message transmit code first checks to see if there is sufficient buffer space to send the message (Step 1). That figure also shows several messages that were composed earlier, and are waiting to be transmitted. When there is buffer space, the message is stored into the next available buffer location (Step 2); the buffer is maintained as a circular queue of a fixed but configurable size. The transmit and receive queue buffers are mapped into cached regions of memory. Unless NES Reclaim mode is used, the processor must issue `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.

The application processor overhead can be reduced by using the NES Reclaim facility where the NES issues `clean` bus operations to maintain coherence between the processor cache and the NES buffers. In this case, the pointer update will cause the NES to reclaim the message, then launch it.

Though the transmit and receive queues are mapped to cached regions, producer and consumer pointers are mapped to *uncached* regions to ensure that the most up-to-date copies are seen both by the application and the NES. To minimize the frequency of reading these pointers from the NES, software maintains a copy of the producer and consumer pointers (P, C in top half of figure). The copy of the consumer pointer need only be updated when it indicates that the queue is full; space may have freed up since by then. The NES may move the consumer pointer at any time if it launches any messages, as illustrated in our example between steps 1 and 2.

Message reception by polling is expected to be the common case, although an application can request for an in-

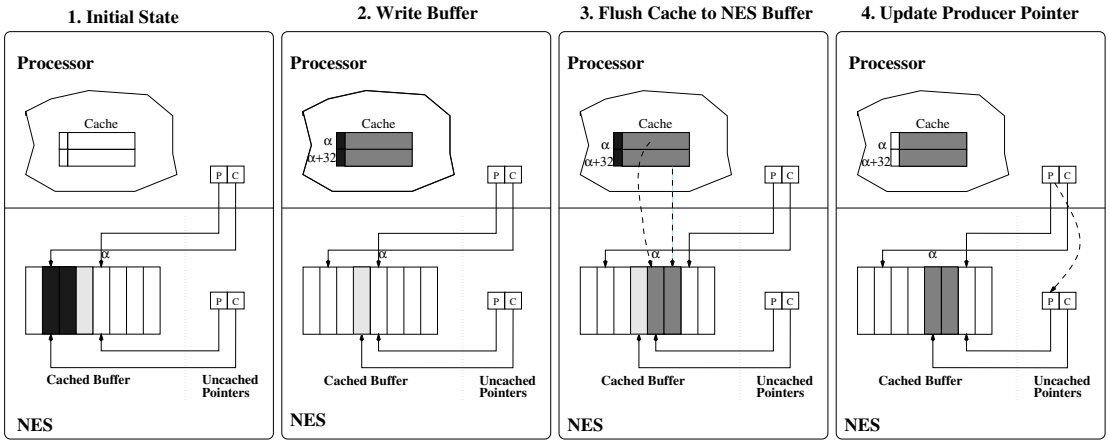

Figure 2. Sending a Basic Message

terrupt 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 the presence of messages. Messages are read directly from the message buffer region. Coherence maintenance is again needed so that the application does not read a cached copy of an old message. As before, this can be done either explicitly by the processor with `flush` instructions or by NES Reclaim.

3.3. Express Messages

Synchronization and simple requests/replies are common in many applications, demanding short, low overhead messages. One minimally-sized, but useful message type consists of a single word of data plus several additional bits that can serve as a tag or identifier. The minimal overhead required to transfer the entire express message between the application and the NES is a single, uncached, memory access. The Express Message Mechanism, which avoids many of the overheads of Basic Messages, is introduced to meet such demands.

To maximize the amount of data transported by a message while keeping each compose and launch to a single, uncached write, the transmit queue name, message destination, and five bits of data are packed into the *address* of an uncached write. Along with the actual 32 bits of the data written, the message body thus contains a total of 37 bits of data. When it observes a store to the appropriate address range, the NES automatically transforms the address into a message header and appends the data from the uncached write to form an Arctic message. Figure 3 shows a simplified format for sending and receiving Express Messages. Additional address bits can be used to convey more information but consumes larger (virtual and physical) address space with potential detrimental effect on TLB if the information encoded into the address bits do not exhibit "good locality"[2].

Unfortunately address bits cannot be used to convey message data to the processor when receiving an Express Message. 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 and subsequently moved into GPRs via (cache) memory[3]. Alternatively, two 32 bit loads into GPRs can be issued to receive the message.

Express Message queue entries are not accessed directly, rather the NES provides a FIFO push/pop interface to transmit and receive. In response to a memory write access indicating an Express Message enqueue operation, the NES provides the necessary buffer address to put the 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. 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[11], the Empty Message can be treated as a legitimate message with a "no action" message han-

[2]Alternate translation mechanisms such as PowerPC's block-address translation mechanism[9] may be employed to mitigate this problem but the use of such mechanisms depends on both processor architecture and OS support.

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

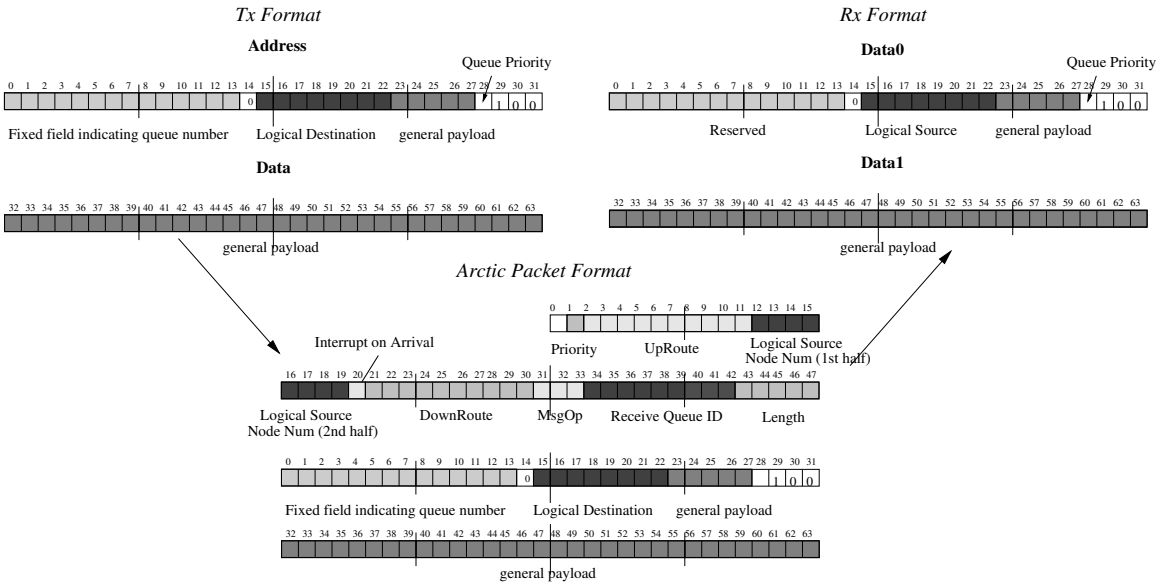

Figure 3. Express Message Formats

dler. Unlike the receive queues, transmit queues employ producer and consumer pointers between software and the NES to coordinate availability of message buffer space.

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 application can unilaterally achieve the movement, *i.e.* 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. 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. The sP at the source and the sP at the destination exchange messages and perform the necessary name translation and protection checks before setting up DMA hardware to perform the transfer. Finally, the packetization, reassembly, and actual data transfers occur in hardware.

The application overhead is reduced by decoupling any page pinning from the transfer request and assigning the responsibility of page management to the sP. SMP operating systems provide support for parallel access to the virtual memory management data structures, enabling the sP to do the management.

3.5. Tag-On Messages

The Tag-On Message mechanism extends the Express Message mechanism to allow additional data to be appended to an out-going message. Designed to eliminate a copy if message data was already in NES SRAM's, it is especially useful for implementing coherent shared memory protocols, and for multi-casting medium-sized messages. As composed by an application, a Tag-On Message is an Express Message with the addition of several previously unused address bits to specify an SRAM location where the additional message data can be found. At the destination NES, a Tag-On Message is partitioned and placed into two separate queues. The first part is its header which is delivered as an Express Message. 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.

The message body of a Tag-On Message has three decoupled data fields: a 5 bit tag, a 32 bit word, and up on three cache-lines of data. The header and message data is not located in continuous addresses but separated into two separate queues. This is useful in coherence protocol when shipping a cache-line of data from one site to another. The 32 bit word field corresponds to the address of the cache-line and the 5 bit tag field identifies the message type. In StarT-Voyager, the sP first issues a command to move a cache-line of data from main memory of an application into its SRAM. A Tag-On Message then ships the data to the requester. A cache-line data may also be pushed into the NES without

the sP asking for it, for example the aP's cache may initiate 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 sent directly.

Tag-On Messages are also useful for multi-casting. To multi-cast some data, an application first moves it into NES SRAM, and then issues a separate, low overhead Tag-On Message for each destination. Thus, the bulk of the message data is moved over the system memory bus only once at the source site.

3.6. OnePoll

In order to minimize the overhead of polling from multiple receive queues, StarT-Voyager introduces a novel mechanism, called *OnePoll*, 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 within some of its address bits the queues from which to poll. The result of the read is 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 name and its queue pointers is returned. If there are no messages in any of the polled queues, a special Empty Express Message is returned.

OnePoll is useful to user applications, most of which are expected to have four receive queues: Basic and Express/Tag-On, each with two priorities. The sP has nine queues to poll; clearly the OnePoll mechanism dramatically reduces the sP's polling costs.

4. Protection and Multiuser Concerns

StarT-Voyager was designed to support a multiuser, multitasking, loosely gang-scheduled environment. To achieve this goal, it employs a protection scheme which supports multiple privately owned, simultaneously active message queues at each site. A number of translation mechanisms are used to restrict accesses to each queue.

4.1 Communication Protection Scheme

Each StarT-Voyager site logically supports up to 512 pairs of active transmit/receive queues. These queues are similar to main memory DRAM pages in that they are allocated by the system to parallel jobs, and once allocated, the queues are privately owned until they are returned back to the OS. Local access to a 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. Remote access, *i.e.* the right to send message to a remote receive queue, is restricted by subjecting the destination of each message to a translation. The application code uses a logical destination name to specify the destination of each message. Associated with each transmit queue is a Destination Table (see Figure 4) set up by system code which maps this logical name to a physical site and queue name[4].

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

Figure 4. Destination Table: each row contains the logical destination to ⟨site, RxQ, source⟩ mapping for a transmit queue.

The NES also attaches to each outgoing message a source identifier which the destination process can use in its reply message. In StarT-Voyager, such a specifier is also kept in the Destination Table (see Figure 4); this design allows the same process to have different source identifiers for different destinations. It can be useful in distributed application communications, where the structure of communication domains may be modified dynamically, making it difficult for a process to use the same source identifier without resorting to a sparse name space.

4.2. Job Scheduling Flexibility

Under StarT-Voyager's protection model, each message identifies its receive queue, which remains active even when its owning process is not currently running. In contrast to traditional MPP's protection model which typically supports only a single set of network queues, this protection scheme does not require the network state to be swapped out during a parallel job context switch. In fact, StarT-Voyager's protection model removes the requirement to gang schedule in order to enforce communication protection. A parallel job can be suspended and "swapped out" by suspending and swapping each of its constituent processes independently, *without global coordination*. This model liberates job scheduling to concentrate on resource availability and data dependence as the factors governing scheduling policy.

Logical destination translation also enables processes to transparently migrate between sites and physical queues. Several processes of the same job can also be mapped to the same site, each with its own set of message passing queues.

[4] 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 currently implement this additional hardware.

This is especially useful if a parallel job must be moved to a smaller space partition, say to adapt to the failure of a site.

4.3. Message Queues Implementation

The number of queues at each site is far too large to be supported economically with expensive SRAM hardware. Consequently, each NES supports 16 pairs of hardware transmit/receive queues (8 Express/Tag-On and 8 Basic), which are used as a software-managed cache for the 512 pairs of transmit and receive queues. The logical queues mapped to hardware queues are called *resident* while the others are called *non-resident*. The sP implements the transition of a queue between resident and non-resident resources. With the sP's assistance, Non-resident Message queues continue to transmit and receive messages without having to be moved into resident resources. Non-resident queues' buffer space is mapped into the aP's DRAM while the producer-consumer pointers are mapped into an address space serviced by the sP. The latter allows pointer operations to trigger sP processing.

The sP multiplexes the non-resident transmit queues to a single hardware transmit queue, and performs destination queue name translation in firmware. It also does the demultiplexing of messages destined to non-resident queues from a single shared receive queue called the *miss queue*. When a message packet arrives at an NES, it determines if the destination queue specified in the message header is resident by looking it up in a table of receive queue cache-tags. If it is and and there is enough space in that queue for the message, the NES enqueues the message. This entire process occurs in hardware. Otherwise, the NES enqueues the message in the miss queue.

5. Projected Performance

This section presents estimates of resident queue performance using three performance metrics: processor overhead, communication latency, and peak bandwidth. The results are generated using C code sequences compiled into assembly code, manually optimized and scheduled for a PowerPC 604 processor. Only one general purpose register is assumed to be globally used for message queue information such as counters and buffer addresses. 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 the low-level simulator used by hardware designers. Bus latency of accesses to the NES is counted. The numbers assume that the aPs and sPs will run at 140MHz, the system bus and most of the NES will run at 35MHz, and the Arctic network will run at 37.5MHz[5].

The main observation is that there are several performance metrics and the choice of message passing mechanism depends on which metric is relevant or most important.

5.1. Processor overhead

Processor overhead is the number of cycles required by a processor to compose and launch or to receive a message. There is one routine to send and one to receive. The number of cycles is the time spent in each routine. The processor pipeline and bus interface buffers are assumed to be empty at the start of routine execution. The receive overhead is divided into two parts: overhead and load latency. Overhead is the receive overhead that is unavoidable due to buffer management. Load latency is the time required to read data from the NES card. This could be considered to also be processor overhead if the data is used immediately[6]. This overhead can be reduced with software prefetching; in the very best case, it contributes nothing to processor overhead. Figure 5 shows the processor overhead for sending and for receiving all four types of messages[7].

The processor overhead for Basic Messages shown in Figure fig: po-no-reclaim 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. 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.

The overhead reported for DMA does not include that of pinning pages and is applicable to the situation where pinned pages are used repeatedly. DMA has no software receive component and thus no corresponding overhead.

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

[5] we opted to implement part of NESCtrl in FPGA to provide greater experimental flexibility at the prices of lower clock speed. With FPGAs, we can get the machine built, run a series of experiments, and then modified the hardware design in response to those experiments. The tradeoff is a lower clock speed: the NES will now run at at 35MHz, and the aPs and sPs at 140MHz.

[6] 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.

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

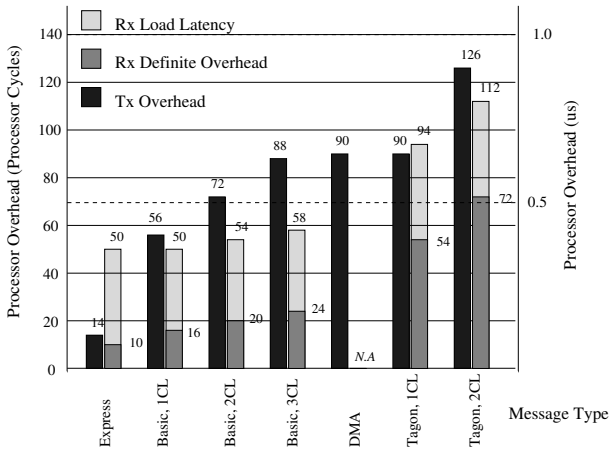

Figure 5. Processor overhead for the different message passing mechanisms, assuming NES Reclaim is used to maintain message buffer coherence for Basic Messages but not for Tagon.

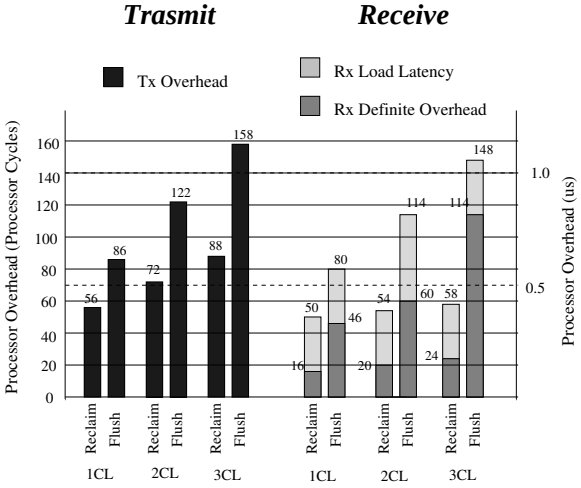

Figure 6. Comparison of Basic Message processor overhead with message buffer space coherence maintained by software (Flush), and by NES hardware (Reclaim).

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.

Total latency is lowest for Express Message with a 2μs latency for nearest neighbor communication. Except for DMA, all message types take under 6.5 μs even for messages that travel the furthest distances. The DMA latency for 256-byte transfer is also shown. The reasonably low DMA latency, together with the low processor overhead to initiate DMA (presented in the previous section) makes DMA effective for relatively modest-sized transfers. When a message travels many hops in the network, Arctic network's latency dominates the latency for all non-DMA messages. With future router chips expected to have higher degrees and faster clock rates, the network latency should decrease, making the other latency components that 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 continuously. Furthermore, a site is of either sending or receiving messages but not both. It takes into account resource contention between successive invocations 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 15 MBytes/sec bandwidth. The highest bandwidth of over 110 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 performance is limited by the Arctic Network's bandwidth.

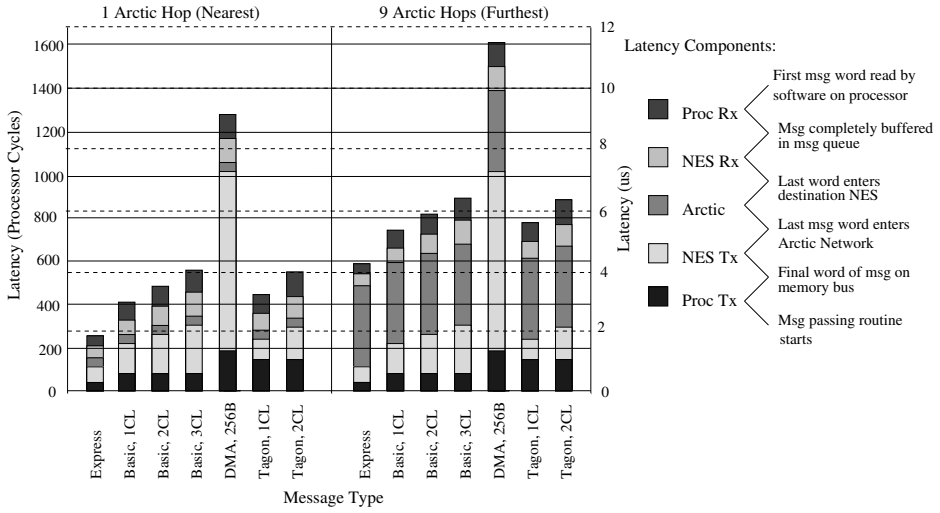

Figure 7. One-way communication latency for StarT-Voyager's message passing mechanisms.

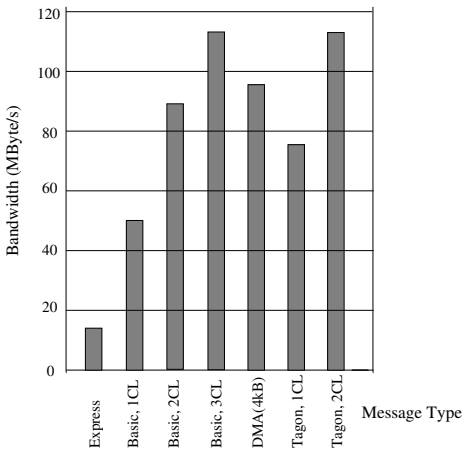

Figure 8. Comparison of peak communication bandwidth that each message passing mechanism can deliver.

5.4. Discussion

StarT-Voyager's message passing mechanisms have different performance characteristics.

Those with the best bandwidth and latency performance generally incur the highest processor overhead. For example, non-reclaimed basic messages have lower latency and slightly higher bandwidth than reclaimed basic messages. For the latter, software on the processor flushes the composed messages to the NES explicitly so that the NES can launch the message immediately upon receiving indication that the message has been composed. In contrast, for reclaimed messages, the message ready signal to the NES starts the reclaim operations which must complete before the transmit starts. In addition, for the 604 bus protocol, the processor flushing the message explicitly is more efficient than having an external bus device issue a flush bus operation on the message. This is because the 604 will retry the flush bus operation while it writes out the message.

Reclaimed messages, however, require much less overhead than non-reclaimed messages. Reclaimed messages avoid the need for the processor to perform the flush, which can take between 15 and 25 processor cycles to complete. These cycles generally cannot be hidden, since a memory barrier instruction often follows. A reclaimed message transmit or receive allows the processor to do more work when it is not communicating since it is less busy actually sending and receiving the message. If the computation is compute bound, achieving lower processor overhead is the overriding concern.

Basic messages can be more efficient than tagon because basic message signaling to the NES can be aggregated.

Tagon messages, however, perform very well for multicast operations, because the multicasted data needs to travel over the memory bus only once.

6. Conclusions

StarT-Voyager is designed to be a scalable, general-purpose parallel system that effectively supports 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: (i) providing sufficient support to cover a wider range of communication needs efficiently, and (ii) developing a simple but flexible protection scheme.

Different message passing mechanisms are useful for different types of communication beyond the obvious latency/bandwidth/processor overhead tradeoffs. For example, when a cache-line that resides in a shared memory region is pushed out of the cache, the NES captures the cache-line in a circular buffer. Depending on the circumstances, that cache-line may have to be sent to another site or may be written back to local DRAM. In the former case, the Tag-On mechanism should be used to avoid copying. Tag-On composes the header and launch the message in a single bus operation, minimizing latency which is important for coherency operations.

On the other hand, if a processor is spawning a number of remote procedure calls where all the arguments are different but known, the Basic Message Mechanism is superior. The spawning processor can aggregate the pointer update, reducing overhead. In addition, it is more convenient for both message sender and receiver to manage the buffer if the entire message is in a single continuous region of memory.

The needs of a cache coherence protocol engine are different from that of an application processor. For a protocol engine, it is important to quickly process the request and then move on to another task. Overhead not only delays the current action, but prevents other actions from being processed as well. On the other hand, in application code, processor overhead is more tolerable since the superscalar architecture can often find other instructions to execute while waiting for a memory operation to complete.

The issues addressed in StarT-Voyager are general issues. Processors will always have hierarchies of storage: registers, caches, memory, disk, and so on. In order to maximize communication performance, the characteristics of the each hierarchy level and how it communicates with its neighbors must be taken into account. Communication requirements do tend to follow the capabilities of the hierarchy level that store the message data. For example, data that is latency sensitive generally resides in registers, while data that is bandwidth sensitive generally resides in memory. Although the exact performance tradeoffs may differ from machine to machine, we believe that it is possible to dramatically improve overall system performance by providing several communication mechanisms and tailoring each to fit the communication requirements of size and location.

Acknowledgements: 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.

References

[1] M. S. Allen, M. Alexander, C. Wright, and J. Chang. Designing the PowerPC 60X Bus. *IEEE Micro*, pages 42 – 51, Oct. 1994.

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

[3] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. The StarT-Voyager Parallel System. In *Proceedings of PACT'98, Paris, France*, Oct. 1998.

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

[5] D. Culler et al. 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.

[6] J. C. Hoe. StarT-X: A One-Man-Year Exercise in Network Interface Engineering. In *Proceedings of Hot Interconnects VI*, Aug. 1998.

[7] J. C. Hoe and M. Ehrlich. StarT-Jr: A parallel system from commodity technology. In *Proceedings of the 7th Transputer/Occam International Conference*, Nov. 1996.

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

[9] 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.

[10] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd ISCA*, May 1996.

[11] 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 ISCA, Gold Coast, Australia*, pages 256 – 266, 1992.