StarT-Voyager: A Flexible Platform for Exploring Scalable SMP Issues

Boon S. Ang

MIT Laboratory for Computer Science
545 Technology Square, room 205
Cambridge, MA 02139

hahaha@abp.lcs.mit.edu

Derek Chiou
MIT Laboratory for Computer Science
545 Technology Square, room 203
Cambridge, MA 02139
derek@abp.lcs.mit.edu

Daniel L. Rosenband

MIT Laboratory for Computer Science
545 Technology Square, room 217
Cambridge, MA 02139
danlief@abp.lcs.mit.edu

Mike Ehrlich

MIT Laboratory for Computer Science
545 Technology Square, room 243
Cambridge, MA 02139
mikee@abp.lcs.mit.edu

Larry Rudolph

MIT Laboratory for Computer Science
545 Technology Square, room 208
Cambridge, MA 02139
rudolph@abp.lcs.mit.edu

Arvind

MIT Laboratory for Computer Science
545 Technology Square, room 210
Cambridge, MA 02139
arvind@abp.lcs.mit.edu

Abstract:

This paper describes StarT-Voyager, a machine designed as an experimental platform for research in cluster system communication. The heart of StarT-Voyager is a network interface unit (NIU) that connects the memory bus of a PowerPC-based SMP to the MIT Arctic network. The NIU is highly flexible, with its set of functions easily modified by firmware or by programmable hardware, making it possible to compare different communication interfaces and implementation strategies on a common platform. Its flexibility comes from a fast embedded processor and large, fast FPGAs that surround a high-speed protected communication core. Its efficiency comes from a set of primitive operations that are implemented in hardware and are designed to reduce the firmware overhead. Our initial configuration of StarT-Voyager implements four forms of message passing along with S-COMA and NUMA shared memory support. With experimentation on the machine, it can be reconfigured to introduce new mechanisms improving usability and performance.

Keywords:

Parallel systems, network interface unit, flexible, configurable hardware, message passing, shared memory.

1. Introduction

The StarT project was motivated by the desire to investigate high-performance communication mechanisms for symmetric multiprocessor (SMPs) clusters and the implementations of these mechanisms. There is still debate over what are the best set of mechanisms for general parallel application execution. The choice of mechanisms to provide and their implementation can have a profound impact on application performance, and must therefore, be selected after careful evaluation.

How can such evaluation be performed? Building a hard-wired machine capable of evaluating a range of mechanisms and implementations is expensive in terms of both time and resources. It is nearly impossible to accurately compare existing machines since they are implemented with different hardware technologies at different costs and running different software. Emulating the mechanisms on a machine that implements functionality entirely in firmware often provides inaccurate performance data due to bottlenecks that would not exist in hardware. Simulation is either too slow to run real workloads to do real evaluation, or is too abstract to accurately evaluate implementations.

StarT-Voyager is a scalable SMP system that addresses the problems of mechanism selection and evaluation. It addresses these issues by providing a structured, four-tiered architecture that was designed to efficiently implement a wide range of communication mechanisms. The basic design philosophy has been to identify common communication operations, implement them in a generalized form, and export them as primitives to both firmware and reconfigurable hardware extensions. At the lowest level of this architecture is a multiple queue interface that implements protection and service guarantees in a shared network environment. Rather than exposing this core functionality directly to application software as is done in most parallel machines, StarT-Voyager inserts a reconfigurable hardware layer between the two. A firmware engine is provided to provide additional flexibility. StarT-Voyager is able to provide virtually any communication mechanism and implement it in the desired mix of hardware/firmware. It serves as a realistic platform on which real applications and entire system workloads can run.

The rest of this paper proceeds as follows. First, the overall hardware design is reviewed along with a description of the functional units implemented in the NIU hardware. We then describe the default higher-level communication mechanisms that are built on top of the hardware primitives and how they are implemented. This is followed by a description of a set of experiments that illustrate the flexibility of the hardware primitives. The conclusion also provides a comparison with related work.

2. Architecture

This section describes StarT-Voyager's communication layers. StarT-Voyager's communication substrate was designed in a layered fashion, where each layer handles specific parts of communication. By structuring the communication hardware in this way, mechanisms can be implemented efficiently with reasonable amounts of effort. Adding layers introduces very little or no additional overhead since most stages can be pipelined and very few additional stages are required.

The four layers of StarT-Voyager's communication substrate are shown in Figure 1. The four layers are: library code (layer 0), configurable NIU (layer 1), core NIU (layer 2) and the network (layer 3.) Each layer provides specific functionality that we describe below.

- Library code: Library functions generally run within the communicating process. Library functions are provided as a convenience to hide NIU details from the user. For example, we will provide an MPI library that presents the usual MPI interface to the user code but uses the underlying NIU support for the actual communication. However, for many mechanisms no library functions exist; and the process will communicate directly with the NIU using memory operations.
- Programmable NIU: The programmable NIU layer sits on the memory bus of the node and is selectively memory-mapped into the address spaces of processes that use its functionality. It can issue as well as process bus operations. This layer knows the semantics of each address region, allowing it to decide whether to ignore the operation or to handle it. For the operations it handles, it interprets bus operations, either transforming them into commands to the core NIU or handling the operation on its own. For example, an application may send a message by performing a series of bus operations that are translated into a series of commands that compose and launch that message. With sufficient preprocessing on the transmit side and post-processing on the receive side, virtually any communication mechanism such as simple message passing, DMA or cache coherent shared memory, can be implemented.
- Core NIU: The core NIU layer provides the core communication facilities such as simple

message passing transmit and receive coordination and resource management and arbitration. By providing multiple transmit and receive queues, the core NIU is able to provide a protected interface that ensures that messages cannot be sent to an illegal destination and that messages cannot block messages going to another destination. Consequently, different communication abstractions can co-exist on this machine simultaneously. Most other NIUs only provide a small subset of StarT-Voyager's core NIU functionality.

- Network layer: The network provides the actual physical transport layer. We require that the network supports at least two priority levels which make deadlock avoidance much easier and more efficient to implement.

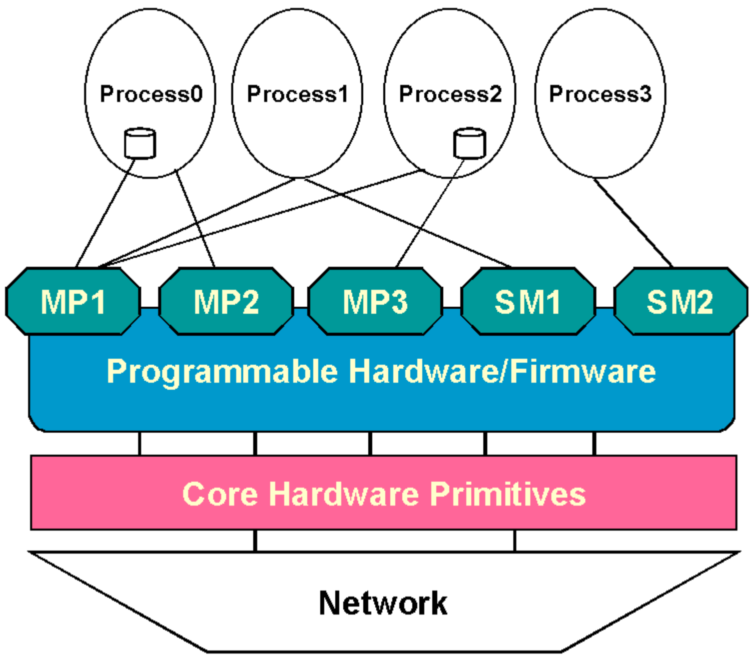


Figure 1: Logical View of Communication Layers of StarT-Voyager. A variety of different message passing (MP) and several shared memory (SM) mechanisms can be supported simultaneously. By simply changing the customizable hardware/firmware interface, old mechanisms can be changed and new ones can be added.

3. Implementation

In this section, we discuss the implementation of StarT-Voyager and point out some of its features. Though we mention which components implement what parts of the communication layers, we discuss the mapping in more detail in Section 4.

The StarT-Voyager system consists of an interconnection network and a set of nodes, with one NIU card per node. Each node consists of an unmodified IBM PowerPC 604e-based two-processor card slot SMP. Each node contains a 166MHz 604e processor and 512KB in-line L2 cache card in one processor slot and a StarT-Voyager network interface unit (NIU) in a second processor slot, Figure 2. The 604e is referred to the application processor (aP). The NIU consists of custom hardware, SRAMs, and a 604 microprocessor that is used as an embedded service processor (sP) to execute firmware. The NIU connects to an MIT Arctic network[1], a 160MB/sec/direction/link fat tree network designed and implemented within our research group. It is important to note that the aP uses all of the original SMP's infrastructure, including the memory controller, DRAM, PCI bridge, and so on.

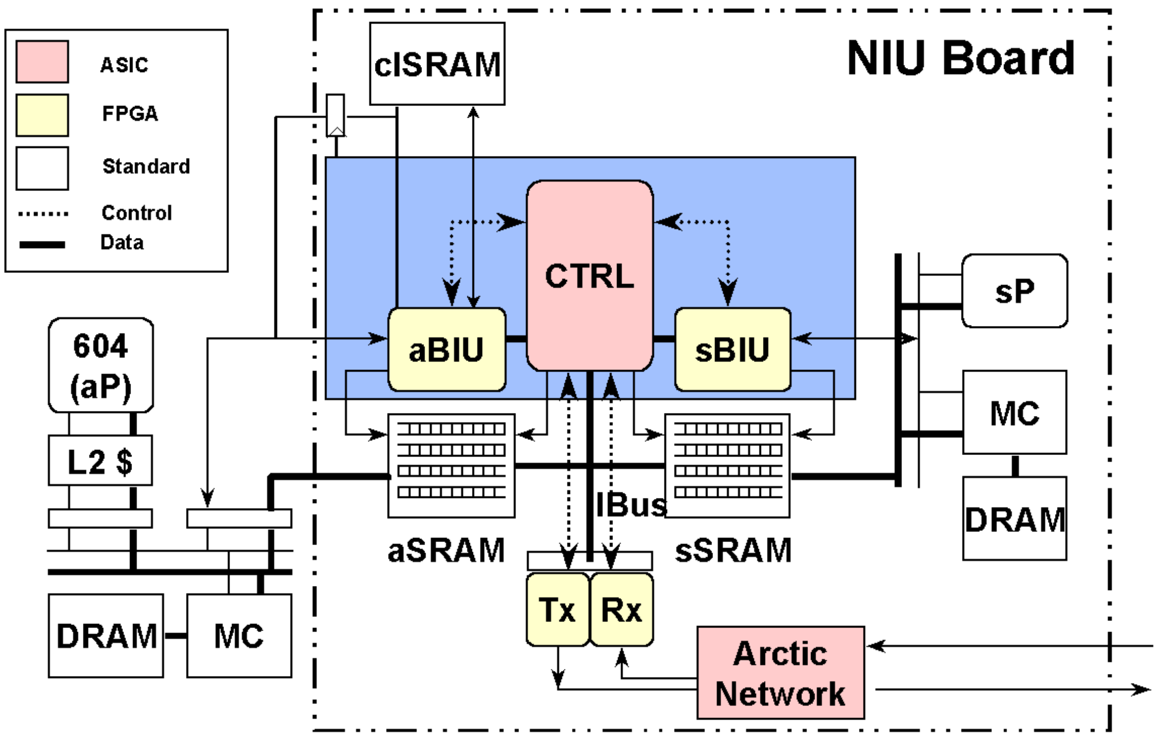


Figure 2: A StarT-Voyager node. The application processor (aP), its level two cache controller, DRAM, and memory controller are all standard. In place of the second application processor is our custom network interface unit (NIU). The NIU contains 3 FPGAs (aBIU, sBIU, TxURxU), 1 ASIC (CTRL), two dual-ported banks of SRAM (aSRAM, sSRAM), a single-ported SRAM (clsSRAM), and an embedded processor (sP). The NIU connects the memory bus of the SMP to a high performance interconnection network (Arctic).

StarT-Voyager implements the core NIU layer (layer 2) in an ASIC (CTRL) and an FPGA on the datapath to/from the network (TxU/RxU). CTRL was too large and had too many performance constraints to implement in an FPGA. The programmable NIU layer (layer 1) is implemented in two large, fast FPGAs surrounding CTRL and a service processor (sP), another 604 processor to run firmware code. The two FPGAs serve as an aP bus interface unit (aBIU) and an sP bus interface unit (sBIU). An additional single-ported SRAM (clsSRAM) is available to the aBIU, and is typically used to keep state associated with cache lines in main memory. All signals to CTRL and most signals from CTRL pass through an FPGA.

The Internal Bus (IBus) is the central communication path of the NIU. CTRL's connection to the IBus forms the bridge across which information can cross from the address/control line paths to the data paths and vice versa. One port of each of the dual-ported SRAMs that provide buffering is attached to one of the 604 data buses and the other port is attached to the IBus. The IBus also connects to TxURxU through a set of hardware FIFOs.

Moving Control Information over Data paths and Data Information over Control paths

An important aspect of the StarT-Voyager NIU is its uniform treatment of bus signals. It has the ability to move what is normally thought of as control signals, such as memory bus address and control signals, to data paths. For example, the aBIU can write the address captured from the aP's bus into one of the SRAM banks via the IBus. This information can then be examined by the local sP and/or sent over the network to another node.

Similarly, information from the data path, such as dynamically generated commands issued through the command queues, can be treated as control information. For instance, data in one of the SRAMs can be used by the aBIU as address and control signals for a bus operation on the aP bus. Such capabilities are not available to most message passing NIUs where addresses are used exclusively to specify buffer locations or memory mapped control registers. The ability to move information between control and data paths in a general way enables both shared memory abilities, and forms of message passing that use the address as part of the header/data of the message or vice-versa.

Underlying Queue Support

The low-level management of the transmission and reception of messages across the network is also under the charge of CTRL. While buffer space for message queues in StarT-Voyager is provided by the dual-ported SRAMs, control state for these queues (for example producer/consumer pointers) resides inside the CTRL ASIC.

Queue pointer updates trigger CTRL actions. In the case of transmit queues, producer pointer updates indicate that messages have been composed and are ready to be launched. Receive queue consumer pointer updates indicate that messages have been received by the processor and that its SRAM buffer space can be reused. CTRL choreographs both the transmit and receive processes. CTRL updates the appropriate transmit queue consumer pointer after it has launched a message, or the appropriate receive queue producer pointer after a new message has arrived.

CTRL also supports two local command queues and a remote command queue. Through the local command queues, the sP issues commands to its aBIU, CTRL, or sends messages to other nodes across the network. Commands to aBIU and CTRL can be seen as messages that request services. Examples include performing a bus operation on the aP bus or performing an aSRAM to sSRAM copy. The remote command queue is similar, except that commands in this queue originate from nodes across the network. Commands in these queues are processed in FIFO order to simplify the coordination for a sequence of commands with mutual dependence. One last queue in the NIU is used by the aBIU to communicate with the sBIU.

4. Mapping Communication Layers onto StarT-Voyager

In this section, we discuss how the communication layers described in Section 2 are implemented on StarT-Voyager. We start from the core NIU layer (layer 2) and work our way up.

Core NIU (Layer 2)

As mentioned earlier, layer 2 of the StarT-Voyager communication stack consists of a protected message passing queue abstraction. There are two categories of queues: network queues and command queues that are essentially local message queues. We discuss the primitives supported by the Core NIU and, for the most part, implemented by CTRL. After presenting this list, the interface and functionality of each unit is explained.

- **Multiple transmit and receive queues:** Direct support for 16 transmit and 16 receive queues.
- **Ordered firmware command queues:** Two command queues provide firmware the ability to send messages across the network, send messages to the aBIU, issue block operations, and to configure NIU state. With the exception of block operations, commands are guaranteed to be issued and completed in order. These queues are especially useful for shared memory.
- **Remote command queue:** Remote sites can issue commands to CTRL and the aBIU.
- **Transmit queue prioritization:** Arbitration between multiple transmit queues using a dynamically reconfigurable system register that specifies queue priorities.
- **Protection:** Messages cannot be launched into the network without the appropriate permissions. If protection is violated, the queue is shutdown and firmware/OS is notified by an interrupt.
- **Destination/resource translation:** By default, messages specify virtual destinations that are translated into a physical site, a logical receive queue and a network priority. The OS or firmware can disable translation on a per-queue basis.
- **Receive queue caching:** A large receive queue namespace is supported out of which a small number are cached in hardware queues within the NIU. Others are sent to a miss queue that is serviced by firmware.
- **Block read and transmit:** Block operations on address ranges of up to a full page are provided.

CTRL Interfaces

CTRL implements most of the core hardware functionality that is mentioned above. It has been designed to export these core functions to the BIUs through several interfaces. The BIUs in turn export the functions to both the aP and sP, but are also able to create more complicated operations by using several of the core functions to satisfy a single processor request. This allows for a very flexible and efficient interface between firmware and the core hardware. Firmware can access all hardware functions inside CTRL. However, as critical sequences of operations are identified, the BIU implementations can be extended to support these more complicated operations. In this case the BIUs rather than the firmware will generate the sequence of operations that CTRL needs to perform, thereby reducing processor overhead and improving the core hardware utilization.

There are several major interfaces between each of the BIUs and CTRL that allow easy access to all of CTRL's functionality. Both BIUs can request CTRL to write data to SRAM, and both BIUs can update and read CTRL's internal state (including queue pointers and system registers). Surprisingly, these two interfaces provide access to most of the core functions. Messages can be composed by writing data through CTRL to the SRAMs, messages can be launched by updating the appropriate queue pointer inside CTRL, and queue priorities, permissions and many other configuration registers can be set through writes to the system registers in CTRL.

A small extension to the interface that lets sBIU write to the SRAMs through CTRL also allows sBIU to issue commands to a pair of command queues that CTRL manages. These command queues allow firmware or sBIU to issue a sequence of operations that transmit messages over the network, copy data from one SRAM to another, and also issue bus operations to the aP memory bus. Each queue processes its commands in-order, making the queues very useful for shared-memory protocol processing.

The central data-path that connects CTRL to the SRAMs and the network is the IBus. Almost all data that flows through the NIU (ranging from received messages, to aP to sP communication) will cross the IBus at least once. CTRL manages the IBus. Considerable effort has been put into optimizing its usage, as it is a critical resource in the system.

There are many other interfaces that enhance performance and are required for correct operation. One of these is an interface that allows CTRL to issue bus operations to the aP memory bus (through aBIU). Others simply return status information to the BIUs.

CTRL's Role in Core NIU Functionality

The core functionality was designed with the idea in mind that it could be used under many different cicrumstances. Hardware in the BIUs may be requesting a service, firmware running a shared-memory protocol may be using the hardware directly, or even parts of application code may be accessing some of the core functinality directly. In order to support a large variety of uses we have added many options to each of the core functions. This will improve performance, make the core functions easier to use, and will also enforce protection. In the next paragraphs we describe a few of the options and features we believe are important and demonstrate some of the flexibility that is provided by the system. There are many more features and options that are supported, but unfortunately it is beyond the scope of this paper to describe them in greater detail.

One important feature when transmitting messages is destination translation. It can be used to enforce protection or simply to make routing and destination queue selection easier. CTRL implements the destination translation by first applying an AND/OR mask to the virtual destination that is specified in the message header (the mask bits are specified as part of the transmit queue state). The result of the AND/OR mask is used as an index into a translation table in one of the SRAMs. The translation table entry specifies the physical route, logical destination queue number and a few other parameters. Thus, if software does not want to worry about routing and logical destination queue numbers, it simply needs to specify the virtual destination and use the translation mechanism. CTRL also supports raw messages in which the header is fully specified (no translation needed). However, raw messages will probably not be used frequently as protection cannot be enforced when they are being used.

Translation at the receive end is also supported by CTRL. In this case, CTRL translates the logical queue number into a physical queue number. The translation is performed using a process similar to cache-tag lookup. If the queue is not resident (cached) in hardware, then it will be sent to the miss/overflow queue. Firmware will then process the message in the miss/overflow queue and write it to its non-resident (DRAM) location. Selectively caching queues enables the NIU to support a large number of logical destinations efficiently, while using only a small amount of resources. It also makes the machine more suitable for a multi-tasking environment.

There also are many options that can be enabled during message transmit and receive. For instance, message arrival can raise an interrupt if its receive queue has been configured accordingly. Another example deals with messages heading for a full receive queue. In this case, options include, dropping the packet, holding on to it until space frees up in the receive queue (which can lead to deadlocking the network), or diverting it into the overflow queue. Many such options are supported. They will allow us to optimize the software, and will also provide many interesting opportunities for future experiments.

An example of some blocks we have put into CTRL to optimize performance are two block operation units. These and other features we support could have been emulated in firmware, but we thought they would lead to significant performance improvements. The two block operation units are used for block aP bus operations, an for block transmit operations. Block aP bus operations can request that a region of aP DRAM, up to one aligned page, be read into aSRAM. CTRL implements this function by issuing a number of bus operations to the aBIU. The block transmit command divides a block of data in either SRAM bank into packets, adds appropriate headers and bus operations and sends them across the network. The bus operations, when enqueued in the destination's remote command queue, can be used to copy the sent data into the destinations aP DRAM. These two block operations can be chained to implement very efficient DMA transfers.

Programmable NIU (Layer 1)

The aBIU, sBIU and sP together implement Layer 1. The two FPGAs provide the programmable hardware support while the sP executes firmware. The sP relies on the BIUs to provide it with access to the NIU.

Both the aBIU and sBIU sit between their respective 604 buses (address and control signals) and CTRL. In the common mode of operation each BIU observes every bus operation (address and control signals) and activates different finite state machines based on the observed bus operations. The BIUs can ignore bus operations, handle the bus operation completely, forward a processed form of the bus operation to firmware, execute a series of commands to CTRL, or forward the operation to the other BIU for handling. As an additional input, for every bus operation occurring on the aP bus, the clsSRAM is read and the data is passed to aBIU. clsSRAM state can be written from the aBIU. Currently, this happens in response to commands issued by sP, but state machines implemented in the FPGA hardware could also initiate it.

Firmware on the sP is capable of controlling all aspects of NIU operation. The sP has access to both an immediate command interface to CTRL and to both local command queues. The immediate command interface allows the sP to read and update CTRL state. The command queues allow the sP to issue commands to the aBIU, sBIU, CTRL and TxURxU.

The reconfigurability of Layer 1 and parts of Layer 2 through the TxURxU enables hardware functionality to be implemented and exported to sP firmware. For example, in the default design, the sP can observe, respond to and initiate aP bus operations. This functionality is implemented by the aBIU, the sBIU and CTRL through the local command queues.

When sending messages to other nodes, the sP can bypass all translation and message data reformatting which normally occurs in TxU. Thus the sP can take over the function normally performed in the TxU. Though the sP will generally run trusted code, full protection and translation can be enforced for sP generated traffic as well making it suitable to run untrusted code. It is interesting to note that, except for the local command queues, the aP has potential access to all functionality that the sP does. Unless the aP is running trusted code, however, it is unlikely that the functionality will be exposed to the aP.

5. Default Communication Mechanisms

To demonstrate the efficiency and flexibility of the NIU design, we present the following communication mechanisms that have been implemented on top of the primitive operations.

- Message passing
  - Basic: A basic message has a variable length data section of up to 88 bytes. The transmit and receive buffers are mapped to the application code cache. Application code manipulates uncached pointers to transmit and receive buffers. The implementation merely exports the underlying message passing primitive to the user.
  - Express: An express message consists of a five-byte payload. The transmit and receive queues are uncached so that a single uncached store can compose and launch a message. A single uncached load reads the received message from the NIU and frees the buffer space it occupied. Part of the address of a transmit store encodes the logical destination and a byte of data. The BIU generates the appropriate SRAM address to implement a FIFO in a circular buffer. Entries in this FIFO include not only data captured from aP's data bus, but also the address information captured by the BIU, and then written into SRAM via the IBus. The BIU also updates the CTRL pointer after the message has been fully composed. The reverse is done for receive.
  - Tagon: A tagon message is a basic or express message with an additional 1.5 or 2.5 cache-lines of SRAM data attached to it. A pointer in the message description specifies the data in SRAM. This is available for both Express and Basic messages and implemented by CTRL.
  - DMA: An arbitrarily large region of memory can be copied from a local DRAM to a remote DRAM across the network. It is implemented by firmware making use of the primitive block operations.
- Shared memory
  - S-COMA: A simple, cache only memory access mechanism (SCOMA) allows a region of DRAM to be used as a level 3 (L3) cache. The single ported SRAM (clsSRAM) is

used to maintain cache-line state bits that are checked by the aBIU. If the check fails, the bus operation is passed to firmware for servicing. Data supplied by a remote node for a pending read can be received via the remote command queue to avoid firmware execution on the return.
    - NUMA: A non-uniform memory access (NUMA) mechanism is implemented by having the aBIU pass aP bus operations accessing a specific region of DRAM to the sP through a queue implemented between the two BIUs.
- The message passing mechanisms are straightforward to implement. The Basic message mechanism is very similar to the underlying message passing support provided by CTRL. Regions of the dual-ported SRAM are mapped into the user's address space. In order to send a message, the user composes the message including the header into the cached SRAM space, then updates the transmit queue's producer pointer. The BIU handles the writes into the SRAM completely on its own, providing the SRAM the correct address and control signals to capture the data when it appears. The BIU processes the pointer update bus operation (all information for the pointer update is encoded in the *address* of the operation), and passes the pointer update to CTRL, indicating that the message can be launched.
- Receiving a message starts with CTRL updating the receive producer pointer after a message has been received. The user polls the receive pointer with loads that the BIU transforms into reads against the SRAM (the pointers are shadowed in the SRAM by CTRL when it transmits and receives.) The user then reads the message from SRAM. The user then informs the NIU that it has finished receiving the message by updating the receive consumer pointer for that queue with a store that is transformed into a pointer update by the BIU and passed to CTRL.

Express messages are implemented in a similar fashion. However, the header of a transmitted express message is taken from the address part of the store that composed/launched the message. The BIU processes the address and writes it to the SRAM via the IBus using a command to CTRL. The BIU then updates the queue pointer, initiating the transmit.

For Express message receive, the user issues loads to a set address. The BIU understands that the load is an Express message receive, determines whether a message is available to be received and passes the correct data to the load if a receive is possible. It then updates CTRL's copy of the receive queue consumer pointer. If a message is not available, a canonical empty message is returned instead.

For Tagon, CTRL watches for specific bit in the message header that indicates that a Tagon is requested. In that case, other bits in the header provides the SRAM address of the additional data. CTRL reads this additional data onto the IBus after the normal message header and data, inserting that data into the transmitted message.

DMA is a combination of blocked operations. The user sends a message to the sP requesting a DMA. The sP breaks up the DMA into as many blocked operations as are necessary to respect the page limit and boundary limitations, and issues the appropriate read/transmit block operation combinations. If the DMA is a remote read, the sP will send a message to the remote sP to initiate the operation.

NUMA, in the default implementation, is implemented by passing all bus operations within a 1GB address range to the sP in a special queue implemented by the BIUs. CTRL does not get involved at all other than to write to the SRAM's via the IBus on the aBIU's request. The sP firmware does whatever is necessary to ensure coherency, including sending messages to other sPs. The aBIU supports a configurable table that decides whether an operation is actually passed to the sP, allowing the filtering of operations that are not useful for coherence, and which operations are retried until the sP explicitly stops the retries, allowing the sP to satisfy load operations.

S-COMA is a superset of the NUMA support. The clsSRAM bits are read for every aP bus operation and are passed to the aBIU. If an aP bus operation is to a specific region of memory and initiated by the aP, the aBIU will use the clsSRAM bits as cache-line state. Four bits are passed to the aBIU, allowing for multiple coherence protocols simultaneously or very complex coherence protocols. The aBIU determines what action, if any, should be taken with respect to the bus operation. Two bits encode the possible reactions: one bit indicates whether the operation should be retried and the other bit specifies whether the operation should be passed to the sP. These bits are in a table indexed by the bus operation and the clsSRAM bits (NUMA support is similar but only requires indexing on the bus operation.)

Extending Default Mechanisms

There are several ways to easily extend the default mechanisms. For example, to reduce the involvement of sP in shared memory cache-miss processing, the aBIU can be modified to send a message to the home site directly, rather than composing a message to the queue serviced by the local sP firmware.

As another example, StarT-Voyager could emulate Shrimp’s[2] and Memory Channel’s[4] reflective memory communication support. The default StarT-Voyager hardware is sufficient for the sP to implement this functionality. Further enhancements to the aBIU can implement this completely in hardware.

"Diff-ing" hardware can be added in the TxURxU FPGA for update-based shared memory protocols that support multiple writers. Diff-ing is common to software-based shared memory implementations although it is expensive both because comparison is usually done for an entire page, and because it is extra overhead. StarT-Voyager’s clsSRAM can be used to track modifications at the cache-line granularity, thus reducing the amount of diff-ing required. To support diff-ing in hardware, both the new and old data are supplied to the TxURxU so that it can perform the diff and send the appropriate message. Additional support from the aBIU, sBIU and/or firmware is required to provide the old data.

Such modifications enable the performance of different implementations of communication mechanisms to be compared *while keeping all other parameters constant*. Experiments of this type have traditionally been very hard to perform because of the static nature of hardware and the occupancy impact of firmware implementations. To our knowledge, StarT-Voyager is the first machine that allows such experimentation within a single real environment, allowing far more accurate evaluation and comparison of mechanisms.

6. Experiments

This section presents a set of experiments to illustrate the kind of research that can be conducted on this platform. The experiments investigate different ways of implementing block memory transfer, i.e. copying data from contiguous memory locations in one site to contiguous locations in another site. Once the transfer is complete, a message is put into the receiving jobs' regular message queue; the receiver, upon reading this message, can then begin using the transferred data. (This is similar to am_store in Active Message.)

We evaluated three implementations of this feature on StarT-Voyager. Two other implementations are under investigation but results are unavailable at the time of writing. The first three methods differ in the degree of aP and the sP involvement in the transfer.

- **Block Transfer Approach 1:** The sender aP reads data from memory, packetizes them into Basic messages and sends them to the receiver aP, which copies the data into memory.
- **Block Transfer Approach 2:** The aP issues a request to the local sP, which takes over the responsibility of reading, packetizing, and sending out the packets. These packets are received by the destination sP, which moves the data into its final memory locations. This approach shifts the overhead of managing the transfer from the aPs to the sPs. Furthermore, neither processor reads the data directly, leading to lower sP occupancy than aP occupancy under the first approach. Instead, command queue commands allow the data to be transferred directly between aP DRAM and aSRAM, and TagOn messages pick up the data and ship it across the network.
- **Block Transfer Approach 3:** The third approach relies on block operations performed by hardware functional units in the NES to handle the read, packetize and send, receive and write operations. Both the aP and the sP have very low overhead, and the operation essentially happens in the background.

The other two approaches are variants of the second and third approaches. They make use of S-COMA support to optimistically shorten the latency of the block transfer. The dominant latency component of a block transfer is the time between the arrival of the first word of data to the arrival of the last word of data. If the reader can start reading *before* the data has been fully transferred and can be stalled if it attempts to read data that has not yet been transferred, it is likely that much of the data transfer latency can be hidden. Of course, by rewriting the application to make requests for smaller regions of memory, such latencies are not a significant issue. StarT-Voyager's S-COMA support enables such latency masking techniques without a rewrite of the application. In this case, the receiver is optimistically notified that the transfer is complete. Should it attempt to access any data that has not yet arrived, the S-COMA (clsSRAM) cache-line state check hardware will retry the transaction until the data arrives.

- **Block Transfer Approach 4:** This is similar to approach 2, except that the notification of transfer completion is given early, after a quarter of the data has arrived. The sP incurs the additional overhead of first setting the clsSRAM state bits to retry read transactions, and as data arrives, changing them to allow further read transactions to complete.
- **Block Transfer Approach 5:** The last approach is essentially approach 3 with modifications to the aBIU FPGA so that it not only supports moving block transfer data into aP DRAM, but also updates clsSRAM state bits for the relevant cache-line after the data move. The block operation unit can be used to set the clsSRAM bits to their initial retry state.

While the last two approaches can reduce latency tremendously in the good cases, it can also degrade performance if the receiver attempts to use data that does not arrive for a long time. Whereas the receiver could be doing some other work to tolerate latency without optimistic notification of transfer completion, retry by S-COMA cache-line state check hardware prevents the aP from doing any useful work at all. One option to dealing with this problem is having the receiving site's sP send a separate request for the desired cache-line on the hope that it will arrive sooner than the copy being sent in the block transfer.

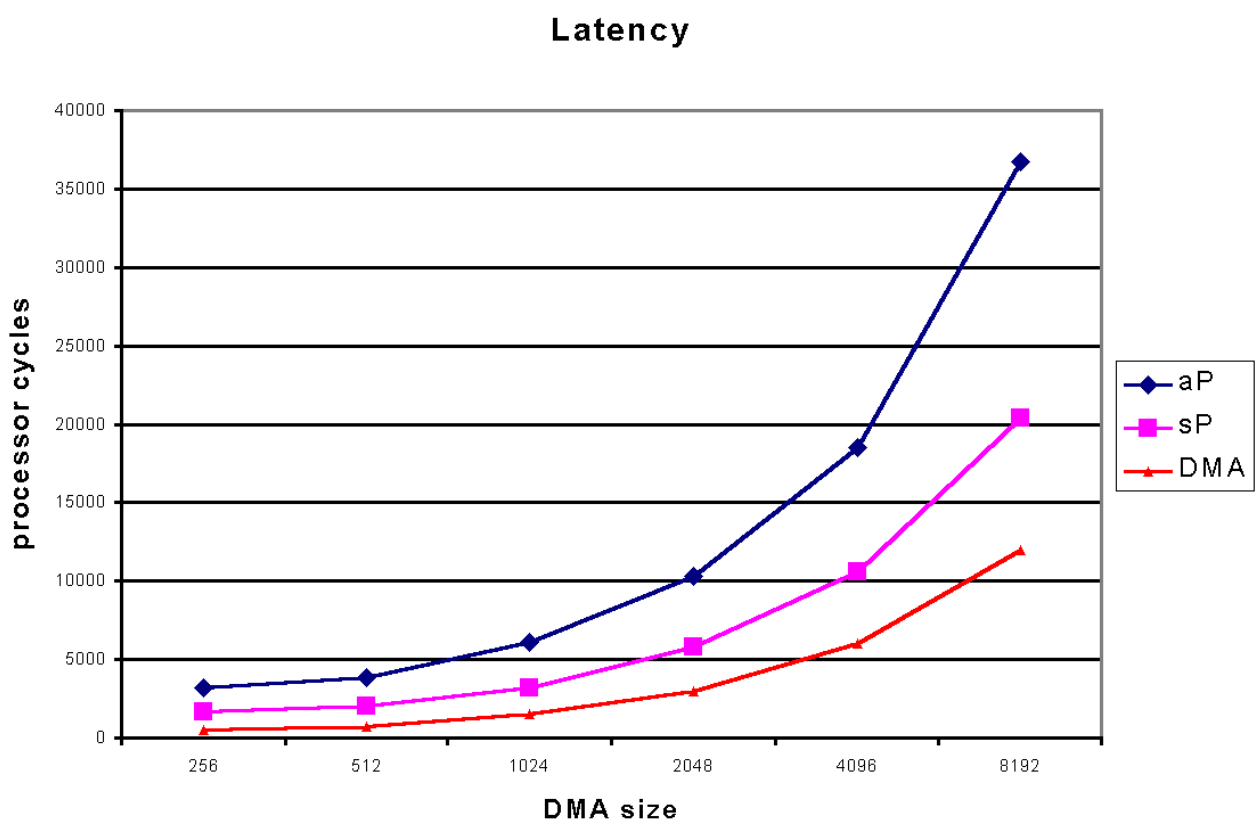
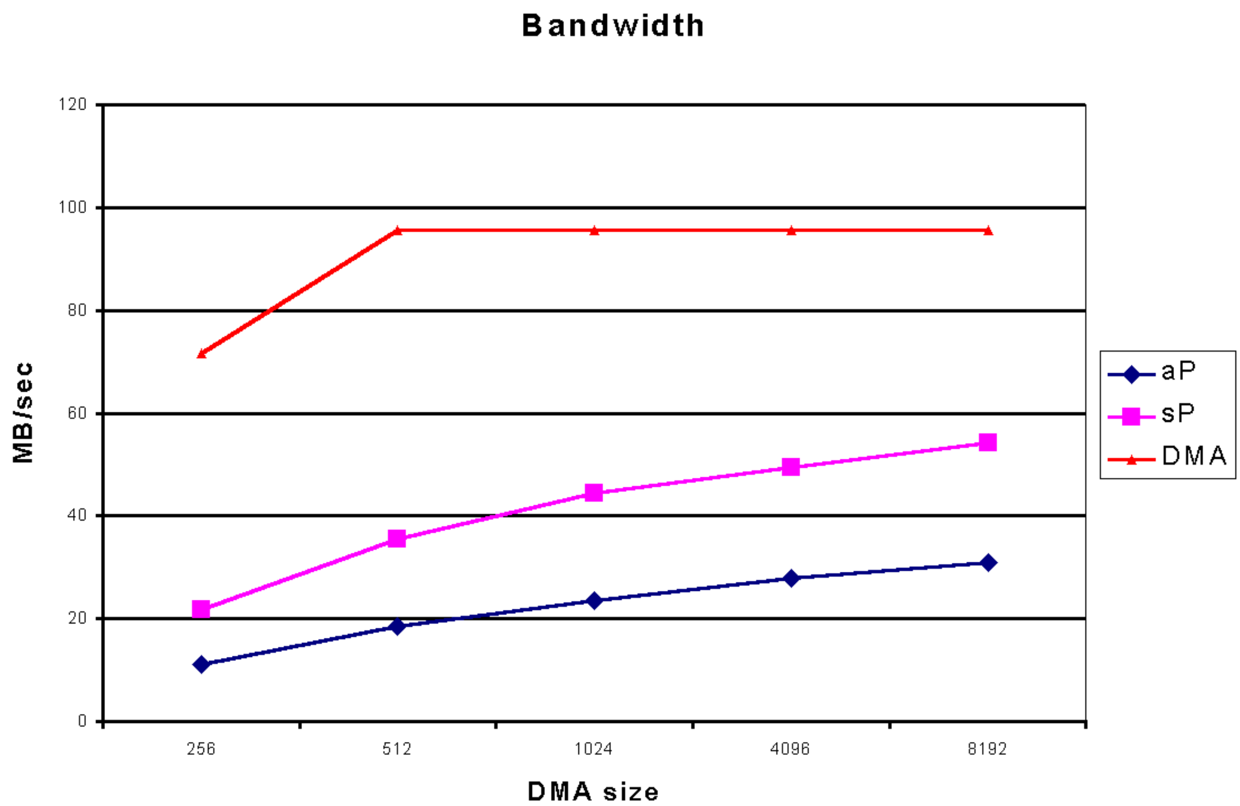


Figure 3: Latencies of Approaches 1 through 3



3: Bandwidths of Approaches 1 through 3

The latency and bandwidth of the first three approaches are shown in Figures 3 and 4 (we did not have sufficient time to produce numbers for the last two approaches.) It is important to note that these numbers have not been optimized; the aP and sP code are very general and unoptimized, increasing latency and decreasing bandwidth. These numbers should be seen as a demonstration of what StarT-Voyager is capable of doing in terms of evaluation, rather than StarT-Voyager's performance.

Approach 1 has the worst performance, because the data needs to be moved over the aP bus twice on each side. In addition, the aP incurs overheads to copy the data from one region of memory to the other.

Approach 2 performs better because data moves over the aP bus only once on each side and the sP never has to actually copy the data. Approach 2, however, has a significant impact on sP occupancy.

Approach 3 has the best performance in terms of bandwidth. The block operations can read and transmit at almost maximum hardware speeds. Occupancy of both the aP and sP is minimal to nil.

These experiments also indicate that firmware engine occupancy is extremely important and can strongly color experimental results. Using S-COMA state is impractical for a firmware engine but will work well with hardware supported DMA. The ability to implement functionality in hardware is an important one and is required to fairly evaluate communication mechanisms.

This example shows that the StarT-Voyager design is extremely flexible, allowing a function to be implemented in multiple ways for straightforward comparison of the merits of each approach. Because we can choose to implement portions of the design in FPGAs instead of in firmware, our approach provides a good idea of the hardware cost, in terms of area and impact on cycle time. This particular example also examines the advantage of using the cache coherency support of a memory bus, via StarT-Voyager's S-COMA support, and the synergy between message passing and shared memory support in this design.

7. Conclusions

StarT-Voyager is an experimental platform designed for cluster system research and is especially ideal for comparing different communication mechanisms "in-vivo". By incorporating both hardware and firmware flexibility, StarT-Voyager is able to natively support a wide range of communication mechanisms, avoiding the problem of apple/orange comparisons when comparing different machines supporting different mechanisms. The construction of the StarT-Voyager NIU demonstrates that this additional functionality does not significantly increase the hardware complexity.

We argue for more functionality to be implemented in network interface hardware than is included in most current network interfaces. By providing simple protection, translation and multiple queues, a wide range of communication mechanisms can easily be implemented. In addition, providing these primitives allows for more general parallel computing and more flexible job-scheduling in multitasking of the parallel system.

StarT-Voyager is similar in spirit to several other machines. The Stanford Flash[7] and Sun S3.mp[8] provide programmable firmware engines which allow for full emulation of any memory-mapped functionality. However, neither machine has the capability to implement additional functionality in hardware once the machine is constructed. In addition, both rely on the firmware engine to service even local DRAM references, making occupancy such a large issue that it becomes very difficult to fairly evaluate an arbitrary mechanism.

Myrinet[3] and StarT-Jr[5] also provides a firmware engine, but sit on the I/O bus and therefore cannot support shared memory. They also have no ability to add further hardware support. Memory Channel[4] and StarT-X[6], other I/O bus network interfaces, are implemented completely with FPGAs and off-the-shelf parts but do not have firmware support. The Princeton Shrimp-II[2] provides no exported programmability of the network interface. The Typhoon architecture proposed by Reinhardt et. al. [9] is similar to StarT-Voyager in that they both use an embedded processor to provide programmability, including allowing it to observe and affect the outcome of coherent memory bus operations. Typhoon, however, does not address the use of message passing and shared memory simultaneously and, therefore, does not address the same protection issues we do.

StarT-Voyager provides a very flexible platform for investigating communication mechanisms and their implementations. Because it will be an actual running system, the investigations will not be confined to single program simulations, but system workload level studies.

Acknowledgments

Acknowledgments

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.

References

1. George A. Boughton. Arctic Switch Fabric. In *Proceedings of the 1997 Parallel Computing, Routing, and Communication Workshop*, Atlanta, GA, June 1997.
2. 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.
3. 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.
4. R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, pages 12 - 18, Feb. 1996.
5. James C. Hoe and Mike Ehrlich. StarT-JR: A Parallel System from Commodity Technology. In the *Proceedings of the 7th Transputer/Occam International Conference*, Tokyo, Japan, November 1996.
6. James C. Hoe. StarT-X: A One-Man-Year Exercise in Network Interface Engineering. To appear in Hot Interconnects VI, Stanford, California, August 1998.
7. 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.
8. A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin. S3.mp: Current Status and Future Directions. *Workshop on Shared Memory Multiprocessors, ISCA 94,* 1994.
9. S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Il, pages 325 - 336, Apr. 1994.