StarT-ng: Delivering Seamless Parallel Computing

Derek Chiou[1], Boon S. Ang[1], Arvind[1], Michael J. Beckerle[2], Andy Boughton[1], Robert Greiner[2], James E. Hicks[2] and James C. Hoe[1]

[1] MIT
[2] Motorola

Abstract. StarT-ng is a joint MIT-Motorola project to build a high-performance message passing machine from commercial systems. Each *site* of the machine consists of a PowerPC 620-based Motorola symmetric multiprocessor (SMP) running the AIX 4.1 operating system. Each processor is connected to a low-latency, high-bandwidth network that is directly accessible from user-level code. In addition to fast message passing capabilities, the machine has experimental support for cache-coherent shared memory across sites. When the machine requires memory to be kept globally coherent, one processor on each site is devoted to supporting shared memory. When globally coherent shared memory is not required, that processor can be used for normal computation tasks. StarT-ng will be available at about the same time as the base SMP is introduced in the marketplace. The ability to be both a standard SMP and an aggressive message passing machine with coherent shared memory makes StarT-ng an excellent building block for incrementally expandable parallel machines.

1 Introduction

The past few years have seen the demise of many companies dedicated to making high performance parallel computers. Some members of the computing community have gone as far as saying that parallel processing, in a classic sense, is dead. Although we strongly disagree with this assessment, we do agree that parallel computing is still at an adolescent stage in its development. We believe the problem is two-fold; it is too hard to program parallel computers, and the hardware, especially for *massively* parallel machines, costs too much and does too little. We are trying to solve the first problem by using implicitly parallel functional languages like Id[17, 4] and pH, and multithreaded languages such as Cid[18] and Cilk[5]. This paper, however, concentrates on StarT-ng, our solution to the second issue.

Mainstream computers have become commodities. PC's have been selling in the millions, forcing down prices, speeding up time tables and making rapid strides in performance. The economies of scale required to stay competitive have resulted in consortia to define and design common systems, enabling hardware to be manufactured in huge quantities. Parallel computers, on the other hand, have traditionally used a lot of custom hardware necessitating the use of custom software. By the time the machine is ready to be sold, its processing node is generally one or two generations out-of-date, making the current commercial microprocessors a factor of two or more faster. Since both the development team and customer base are small, problems cannot be found and worked out quickly, often making custom machines and their software unreliable.

Coupling unreliability with the high cost of custom development, the lack of shrink-wrapped software due to the unique nature of parallel machines and the difficulty in writing custom applications, buying a parallel computer is difficult to justify. One would buy such a system only if the application was critical enough to warrant a dedicated *expensive* machine and the associated custom software development and maintenance cost. Massively parallel computers have fallen into the class of traditional supercomputers, rather than being affordable, widely-available high performance computers as originally envisioned.

StarT-ng[3], a joint project between MIT and Motorola, tries to address these problems. StarT-ng is based on a commercial symmetric multiprocessor (SMP) *system* using PowerPC 620 processors. The goal of the project is to deliver very aggressive parallel performance by making small, manageable changes to the base SMP. We have added support for low overhead, high-bandwidth, user-level messaging, and support for globally coherent shared memory. Starting from a commercial system allows us to leverage infrastructure such as the processor, operating system, memory subsystem, and I/O subsystem. By borrowing most of the system technology, we dramatically reduce development time and cost, allowing us to deliver StarT-ng at approximately the time the base SMP is introduced. StarT-ng, though a research machine, is *commercially* competitive in parallel performance. StarT-ng can also run stock sequential applications efficiently. Just as SMP's allow better sharing of resources such as processors, memory and I/O capability between jobs, StarT-ng allows better sharing of a collection of such resources by being able to run sequential, shared-memory and message-passing parallel applications.

While there are many advantages to using an entire system as the building block of a parallel machine, there are many technical challenges as well. Our design not only has to live with the PowerPC 620 microprocessor, which is optimized for sequential execution, it cannot change the base system in any significant way. Observing these tight constraints while providing competitive performance is the topic of this paper.

Organization: The next section makes a slight detour to explore the next generation of microprocessors. Understanding certain processor issues is essential to understanding StarT-ng. In Section 3, we present an overview of the StarT-ng hardware. This is followed in Section 4 with a discussion of message passing support on StarT-ng. Section 5 discusses how shared memory is implemented on StarT-ng. Finally, we compare StarT-ng with some related work in Section 6 before concluding with the current status of the machine.

---

[3] The *T or StarT project has gone through many changes over its lifetime. For a reasonably detailed history of the many generations of *T, see [2]

2 Memory Systems in Modern Microprocessors

The current generation of high-end RISC microprocessors – such as the DEC Alpha EV5, PowerPC 620 and MIPS R10000 — delivers well over 200 Specint92 and 300 Specfp92 or more. This level of throughput is achieved with a combination of fast clock rates and aggressive superscalar execution. As a result, a complete reworking of the memory system to meet the increased memory-bandwidth demand is required.

2.1 Multi-level Cache Hierarchies

The simplest way to increase perceived memory bandwidth is to establish a wide, high-speed datapath to the processor's primary cache. In order to do so, modern processors contain small (well under 100 kilobytes) level-one (L1) instruction and data caches on the processor chip itself. These caches have wide, low-latency datapaths to the central processing unit and are pipelined to increase throughput. Set associativity of four or greater is often employed to compensate for the limited size of these L1 caches. However, these L1 caches are too small to get high enough hit rates to effectively mask cache-miss latencies on systems where the system clocks are two to four times slower than the processor clock.

As a remedy, current generations of RISC microprocessors provide a dedicated path to a large off-chip level 2 (L2) cache *in addition to* the the standard L3 path to DRAM. A separate path is provided to eliminate the need for an external cache-controller and to improve performance[4]. All aforementioned processors provide a dedicated L2 path that directly connects to high-speed external cache RAM's. All the necessary cache management circuitry are build into the processor itself. This cache interface is clocked independently from the bus interface and offers shorter access latency. These external L2 caches are usually direct-mapped and are strictly inclusive of the on-chip L1 cache contents.

2.2 Split-phased Memory Bus Cycle

The system memory bus is doubly-taxed by current processor design due to the addition of shared-memory support for symmetric multiprocessing (SMP). Not only is the already increased memory traffic from a single processor multiplied by the degree of multiprocessing, the bus must also accommodate cache-coherence traffic. The traditional *tenured* bus handshake in which the memory requester occupies the bus until the memory responds does not provide sufficient bandwidth for a multi-master bus coupled to relatively long-latency slave devices.

To reduce the bus occupancy of memory requests, modern processors support bus protocols where address and data bus are arbitrated separately, dividing the address-cycle, and data-cycle of a single bus operation into independent events. In a simple pipelinable, split-phased bus, after a an address-cycle has completed, the next address-cycle can start immediately, before the data-cycle of

[4] Of course, support for snooping must be provided for L3 traffic.

the previous address-cycle has even started. In this protocol, the only guarantee is that the order of the data bus events corresponds precisely to the order of address bus events. To relax the protocol further, as is done in a true split-phased bus, data events can complete out-of-order with respect to the address events. An identifying tag must be assigned to each address and data event pair to establish a correlation between a responding data-cycle and its requesting address-cycle. In general, a split-phase bus also needs flow control, so that both master, slave devices and snoopers can avoid queue overruns or other resource exhaustion.

2.3 Weak Ordering of Reads and Writes

In order to improve performance, modern microprocessors allow multiple reads and writes to be in progress at the same time. These reads and writes may start or complete on the bus out of order with respect to the instruction stream. These processors must provide a mechanism to allow the user to enforce ordering of memory operations. When a processor is writing to an I/O device, for example, it is essential for some ordering to be maintained. Some ordering mechanism is required, such as a "local barrier" instruction that ensures previous writes executed on a processor complete before new reads or writes are allowed to execute. When a processor is running in an SMP, it may be important to ensure that a processor's write is seen by all processors before that processor continues. In that case, some "global barrier" mechanism is required. These operations tend to be very expensive since they can sequentialize instruction execution[5]. The PowerPC architecture provides only the global barrier, in the form of the `sync` instruction. Since a global barrier also performs local barrier functionality, a `sync` can be used to in place of a local barrier.

Such reordering of memory operations can be problematic if one desires to use the fast, reordering memory path to access special devices such as a network interface unit. Since most standard I/O units are accessed seldomly, it is acceptable to use synchronizing instructions when accessing them. However, for devices that are both low-overhead and are accessed frequently, such as the StarT-ng NIU, the penalties of using sequentializing instructions are quite noticeable. In order to achieve high performance, it is necessary to find ways to minimize or eliminate synchronization overheads.

3 Overview of StarT-ng

The core of a StarT-ng site is a commercial PowerPC 620 SMP augmented with special hardware for message-passing and for implementing shared memory. The 620 is an aggressive 64-bit, 4-way superscalar processor with a dedicated 128-bit wide L2 path. StarT-ng will have 4 MBytes of L2 cache per processor. The

[5] Sequentializing instruction execution is expensive because the next instruction has to wait for all the functional unit and memory-system pipelines to drain.

StarT-ng SMP site uses a data crossbar rather than a data bus to increase data throughput to the memory. An SMP using the PowerPC 601 and the same crossbar and memory system technology is already on the market. The StarT-ng site has 4 processor card slots, each of which contains 2 processors. StarT-ng will replace one to four of the processor cards with network-endpoint-subsystem (NES) cards each containing a single 620 processor, its L2 cache and a network interface unit (NIU). The NIU allows the 620 to communicate with an MIT-developed Arctic network router chip[6]. The StarT-ng system delivered to MIT will have 4 NES boards per site and will have a total of 8 sites.

One of the NES boards will have an address capture device (ACD)[6] which will allow a designated processor to watch and respond to bus transactions. The ACD will be used to implement globally shared, cache-coherent memory. The ACD can be disabled, making it completely invisible to the system.

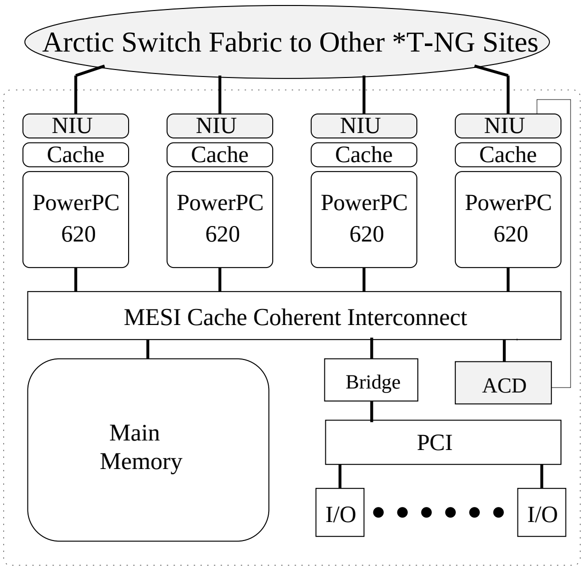


Fig. 1. A StarT-ng site: the white areas comprise the base SMP. The grey areas are our additions.

When a processor is running application code, that processor is called an application processor (aP). When global shared memory support is desired, one 620 processor is dedicated to servicing the ACD and is called a service processor (sP). Common machine configurations are, therefore, four aP's or three aP's and an sP.

[6] All NES boards will have an ACD, but only one will be enabled.

The next two sections describe the message-passing and shared-memory support for StarT-ng.

4 Messaging Support

StarT-ng's user-mode message-passing capabilities are provided through a tightly-coupled hardware network interface directly attached to each 620 processor's L2 cache interface. Each NIU connects the 620 processor to an Arctic[6] packet-switched fat-tree network. The NIU's packet buffers can be memory-mapped into an applications address space. Thus, user programs can send and receive messages, without kernel intervention, by directly manipulating the NIU's packet buffers. In addition to standard networking protocols such as TCP/IP, other popular message-passing layers such as PVM, MPI and Active Messages can be implemented over StarT-ng's networking facilities, allowing StarT-ng to run existing parallel programs that use standard protocols.

This section summarizes the Arctic switch fabric and describes the NIU. The Arctic network router chip and the modular fat-tree switch fabric board is being developed at MIT. while the functional design of the NIU is a joint effort of MIT and Motorola. However, the detailed design and final implementation of the NIU are being carried out by Motorola.

4.1 Arctic Switch Fabric

The Arctic routing chip, a successor to PaRC[11] from the Monsoon project, has been described in other documents[6] and thus will only be summarized here. Its salient features follow.

- 4-by-4 packet-switched router capable of implementing a variety of staged networks.
- 320 MByte/sec/full-duplex-link
- Latency of 6 Arctic cycles per hop.
- Up to 96 byte messages with a maximum payload of 88 bytes per packet.
- Support for two virtual prioritized networks, allowing the implementation of two-priority deadlock-free protocols on a single network.
- Support for secure space partitioning.
- Sophisticated buffer management, allowing it to sustain close to its peak bandwidth.
- Commercial-quality set of test, control and error handling features.
- 16-bit CRC for data and Manchester encoding for flow-control links to detect errors.
- Error rates low enough to ignore under normal operating conditions.
- Hardware router-to-router flow control.

4.2 Network Interface Unit

To fully utilize the Arctic network, a high performance Network Interface Unit (NIU) is required. The StarT-ng NIU is situated on the 620's L2 interface to provide a low-latency, high-bandwidth path between the processor and the Arctic network. The NIU is optimized to support short, frequent messages, common to most fine-grain parallel computation. Using the StarT-ng NIU, the overhead of receiving or transmitting a message by a user-level process to/from the 620 general-purpose registers is conservatively estimated at 100 processor cycles for an 88 byte message and lower for smaller messages, yielding a bandwidth of roughly 120 MByte/sec. These numbers include some synchronization instructions that may possibly be eliminated, potentially doubling the maximum bandwidth and halving the minimum latency.

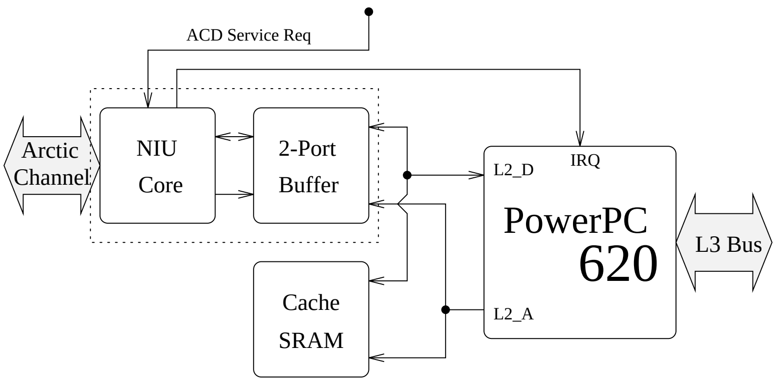


Fig. 2. NIU/620 Configuration

The NIU design incorporates a dual-ported buffer connected directly to the L2 interface of the 620 processor. (see Figure 2.) The other port of the dual-ported buffer connects to the NIU's control logic and datalink transceivers. The 620 interacts with the NIU by reading and writing to specific regions in the buffer. As shown in Figure 3, the dual-ported buffer space is logically partitioned into four data regions and one status/control region. The four data regions are as follows: transmit high priority, transmit low priority, receive high priority, and receive low priority. They are used to transmit and receive two priorities of messages. Each data region occupies two memory pages (8 KBytes) and can thus be individually protected by memory mapping.

The status/control region, located on a separate page, contains a 32-bit control word and a 32-bit status word. The control word allows the 620 to set or query the NIU state, while the status word contains the NIU transmitter and receiver status bits, the ACD service request signal and any state requested by the 620 through the control word. Since the user does not need to access the control region of the NIU for user-level messaging, this region of the NIU interface will be mapped to a kernel page, protecting the NIU from user corruption. All relevant NIU internal states can be read and written by the kernel through the control interface, allowing the network to be context-switched.

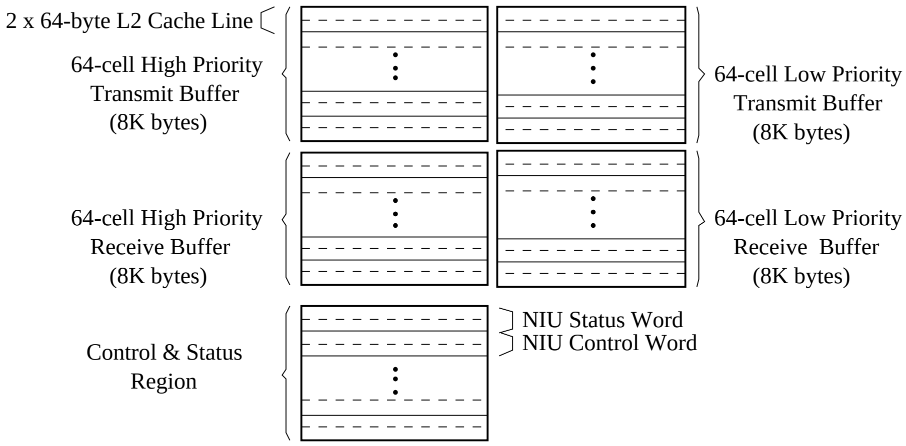


Fig. 3. NIU/620 Interface Buffers

Each transmit or receive data region is further subdivided into 64 packet cells. Each packet cell is 128 bytes large (two 620 cache-lines), enough to hold a maximum sized (96 byte) Arctic packet. Each data region is jointly managed by the 620 processor and the NIU as a circular queue. For the two transmit buffers, the 620 processor acts as the producer of the queue while the NIU serves as the consumer. For the receive buffers, the 620 processor and the NIU reverse their roles as consumer and producer.

Each packet cell contains a v-bit to indicate whether it contains a valid message. Prior to storing a new message into the queue, the producer must first poll the v-bit at the tail of the circular queue to ensure the cell is not still holding a valid message. If the v-bit is invalid, the producer stores the message. After storing the message, the producer then marks the v-bit *valid* to indicate to the consumer that the cell now holds valid data. From the other side, the consumer polls the v-bit from the head of the circular queue. When the v-bit becomes *valid*, the consumer can proceed to retrieve the message from the cell. After retrieving the message, the consumer needs to reset the v-bit to *invalid* to free the cell for reuse. With the use of the v-bit, there is no explicit exchange of queue indices between the 620 processor and the NIU to manage the circular queues.

For receiving a message, aside from polling on the v-bit at the head of the receive queue, the user process also has the option of configuring the NIU to interrupt the 620 based on various conditions such as the arrival of a certain class of message. This feature allows the user program to avoid the overhead of polling the network if it is known that messages arrive very infrequently. If the high priority network is devoted to the kernel, enabling the high-priority arrival interrupt would be an easy way to signal a kernel message has arrived.

The transmit cells takes on a rather awkward format (see Figure 4) due to the order in which cache-line-writes go to the L2 interface. Since the L2 interface is only 128-bits wide (a quad-word) and cache-lines are 512 bits big, cache-line-writes occur in four quad-word writes and always occur from the first quad-word to the last quad-word. In order to avoid any race conditions, the v-bit is located in the fourth quad-word of a transmit packet cell. The packet header (including length and route), plus the first 64-bit data of the outbound message occupy the rest of the fourth quad-word. The remainder of the outbound message data fills the cell successively starting from the first quad-word of the cell, skipping over the fourth. In other words, the second through the seventh 64-bit message words will occupy the first three quad-words of the cell, while the eighth and successive data words will start on the fifth quad-word of the cell.

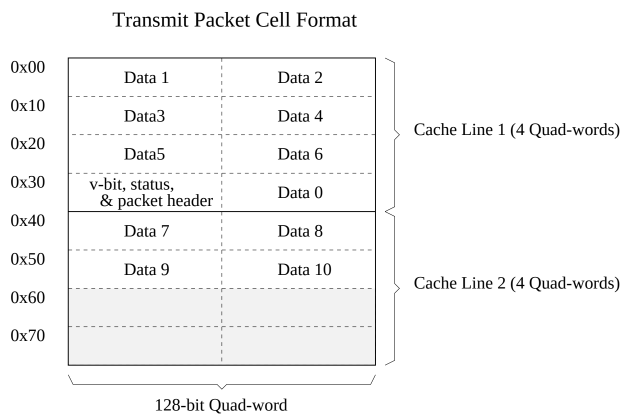


Fig. 4. Transmit Cell Packet Format

The v-bit, packet status plus header, and the first 64-bit data word of an inbound message is packed into the first 128-bit quad-word of a receive packet cell (see Figure 5), rather than follow the transmit format. The remainder of the inbound message data are stored contiguously on the successive word locations. The placement of the v-bit in the first quad-word is dictated by the way reads are done on the 620 L2 interface.[7]

[7] The NIU and the 620 have to agree on the order in which the quad-words are written and read, to ensure that the v-bit is valid only when the data in a cell is valid.

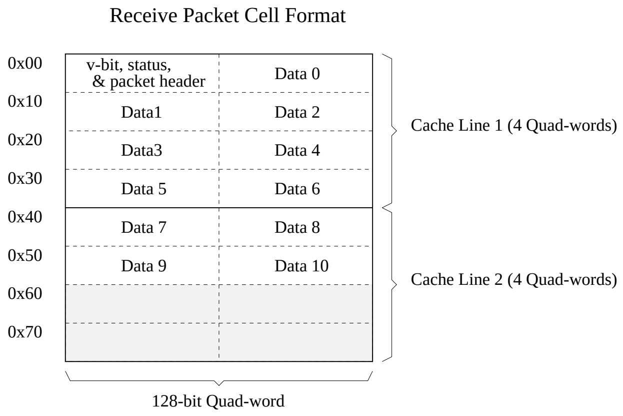


Fig. 5. Receive Cell Packet Format

5 Shared Memory Support on StarT-ng

In order to fully utilize the large, well-supported caches in the 620 and to explore new grounds in architecture, compilation and scheduling, we added coherent cache-line shared memory support to StarT-ng[8]. A design requiring minimal hardware, however, was desired to reduce design and implementation time. An ideal design would allow the use of a 620 processor on each site to support shared memory cache-coherence protocols and additional features such as an L3 software cache. Coherent shared memory implemented using a 620 would be flexible, since the 620 could be arbitrarily programmed, already has access to system resources such as the network and DRAM, and would add no cost to the system.

Unfortunately, the 620 or, for that matter, any commercial processor, does not allow software to observe and respond to bus transactions directly. It was noted, however, that if some device captured bus transactions and propagated responses, a 620 processor could read those transactions and respond to them through that device. Hardware with this functionality, the ACD, is incorporated into StarT-ng to provide the necessary infrastructure to experiment with shared memory.

The sP provides most of the services necessary to implement shared memory. It provides processing power to run coherency protocols, access to the network, and access to site resources such as DRAM. The ACD is simply a way for the service processor to observe and respond to bus transactions and perform necessary coherency operations. This model relies on the fact that the 620 bus

[8] The decision to incorporate support for globally coherent caches after switching from the 88110MP to the PowerPC 620 is a fundamental difference between the original *T project and StarT-ng[2].

is split-phased. The split-phase bus allows the sP to service the ACD over the same memory bus that other processors use to access memory, even though outstanding requests are pending.

The ACD acts as a memory device for the global region of memory. It "captures", that is, buffers and acknowledges, all bus transactions destined for the global memory area. Two capture buffers are provided, one to capture read operations, the other to capture write operations. Having two buffers is necessary to avoid deadlock. One data buffer is provided for reply cache-lines and one address buffer is provided for flushes. The sP needs the capability to issue flushes through the ACD to avoid deadlocks. The reply and flush buffers are written by the sP to initiate a response or invalidation operation respectively[9]. The ACD hardware was designed to be simple to implement and fits easily into 35,000 gates.

Using a standard processor to support global memory has obvious tradeoffs. Clearly, the performance of such a scheme is not as good as if there were special-purpose hardware dedicated to the task. As discussed in Section 5.2, the expected latency of a cache-miss is a factor of four worse than that of Stanford's FLASH, an aggressive shared-memory machine. It is important to note, however, that this penalty is only paid when there is a cache-miss — if miss rates are low, the amortized penalty is also low.

The advantage of StarT-ng's approach, however, lies in its flexibility and ease of implementation. The sP runs arbitrary software, allowing us to experiment with cache-coherency protocols very easily. It can implement such features as a software cache, which improves sharing of remote data among the different processors at a site and which allows features such as update-based protocols, variable-sized cache-line fetches, and other sophisticated streaming techniques. The sP could treat different areas of global memory differently — it's possible to support uncached global memory in this fashion. Even a cache-only memory architecture[9] could be implemented. Also, if shared memory support is not needed for a program, the sP becomes a normal computation processor. The possibilities for experimentation are endless. By coupling the simple ACD with a 620 on an SMP system, we implement shared memory in a very low-cost and flexible way, allowing us to experiment with it and tune it for our purposes.

Since global memory accesses that go through the ACD are slow, the ACD is avoided whenever possible. aP's see two regions of memory, their normal local memory, which is handled in the way normal SMP memory is handled, and global, which is mapped to the ACD. The service processor also sees two regions of memory, local and the ACD command interface. The aP and sP local memory is completely disjoint — the DRAM backing the aP and sP local memory is partitioned. One cannot access the other's memory directly. All global address space handled through the ACD is eventually mapped to some sP's local space[10].

In the rest of this section, we first elaborate on the global shared memory

[9] Other ACD features, specific to the 620, are not discussed in this paper.

[10] In this paper, we assume the sP also acts as the home site protocol processor.

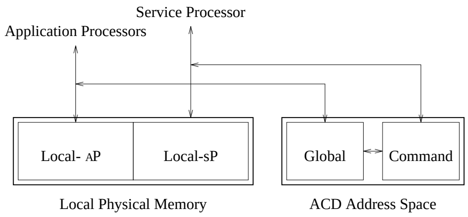


Fig. 6. Memory Organization

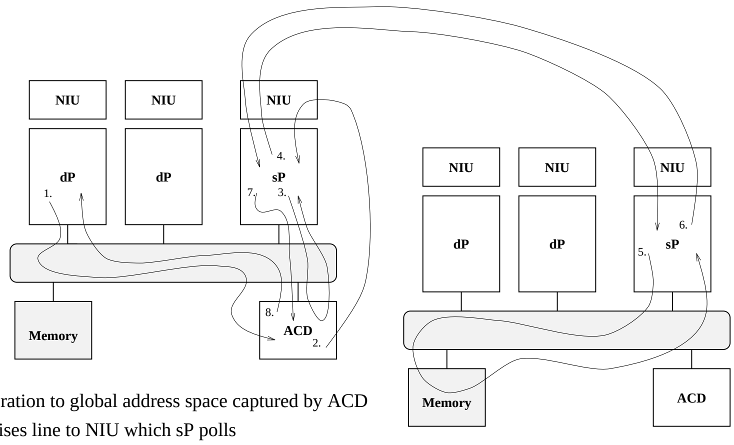
implementation on StarT-ng in Section 5.1. This is followed in Section 5.2 by a discussion of the expected performance of this implementation.

5.1 How Shared Memory Works on StarT-ng

We will illustrate how global cache-coherency is maintained through a general example. The ACD captures an L3 memory operation on a global address by storing away all relevant details, and then notifies the service processor that the outstanding request has been captured by raising an "external event" line on the service processor's NIU, causing a bit to be raised in the NIU status word. Alternatives to the sP polling the NIU are interrupts and polling the ACD over the L3.

The service processor handles the captured request by reading the captured information from the ACD over the L3 bus. After the service processor has read the outstanding request, it frees the capture buffer then performs the actions proscribed by the protocol such as checking the L3 cache, sending a message to the home site, and so on. This sequence of actions is illustrated in Figure 7. Eventually, the requested data is returned to the requesting service processor, which then writes the data to the ACD which puts the data on the L3 bus completing the bus operation.

Deadlocks are possible in any shared memory system. Simple SMP systems solve the problem by having a single point of sequentialization (the address bus), carefully allocating buffers for specific functions and providing "retry" mechanisms to force transactions to back-off if necessary. Distributed schemes have no efficient single point of sequentialization and often cannot know to retry transactions until it is too late to do so. The StarT-ng scheme is even more complicated than a normal distributed scheme since it uses a peer to the application processors, sitting on the same memory bus and using the same MESI protocols and buffering, to run the protocols and service the ACD. Avoiding deadlocks in StarT-ng's shared memory system has been examined in detail in another paper[3].



1. Bus operation to global address space captured by ACD
2. ACD raises line to NIU which sP polls
3. sP services ACD via L3 and retrieves catpured operation
4. sP at requester site forwards captured request to sP at home site
5. sP at home site fetches requested data from local-global memory
6. sP at home site returns requested data to the sP at requestor site
7. sP at requester site furnishes ACD with requsted data
8. ACD supplies requested data to complete the bus operation.

Fig. 7. A simple global cache miss

5.2 Expected Performance of StarT-ng's Shared Memory System

Since StarT-ng's support for cache-coherent shared memory involves a lot of software, a cache miss incurs a significant penalty. In this section, we will assess that penalty and analyze whether that penalty is tolerable for more than just experimentation.

The cache miss penalty depends on whether the desired cache-line hits in the L3 cache. Since the L3 cache is managed by software, we expect that aggressive prefetching techniques, such as bringing over an entire object at a time, can be employed. Also, if prefetching from the processor is done at the correct time, much of the desired data that is being prefetched could wind up in the sP's L1 or L2 cache, speeding up the software access time.

The penalties of cache-misses are shown in Figure 8. The times are given in processor cycles and are very approximate and conservative. The penalties assume that a remote home location is 7 network hops away and that a remote dirty location is 5 network hops away. The corresponding penalties for the Stanford FLASH, as reported in [10] are given for comparison. To the first order, the miss penalties for StarT-ng, assuming the blocking case, are roughly 4 times longer.

To consider the actual impact on performance, one has to take into consideration miss rates and the percentage of memory operations in a program. First, let us consider miss rates. In the same FLASH paper, the miss rates for selected

Type of Miss	StarT-ng (proc cycles)	FLASH (proc cycles)
L3 Hit	157	54
Local Clean	199	54
Local Dirty Remote	825	286
Remote Clean	770	222
Remote Dirty at Home	825	290
Remote Dirty Remote	1455	382

Fig. 8. Miss penalties in processor cycles. The numbers for FLASH are taken from [10].

SPLASH programs are obtained from a simulator for FLASH which assumes a 1 MByte cache. The miss rates, except for MP3D, range from 0.06% to 1.00%.

In terms of overall performance, it is clear that unless the miss rate is under 1%, the performance of StarT-ng's shared memory system is quite poor. If the miss rate is less than 0.1%, which occurs for 3 of the 7 benchmarks with a 1 MByte cache, our performance is quite good. We believe that integrating network virtual memory to reduce miss rates and using split-phase caching techniques to bound miss penalties will dramatically drop the miss penalty, allowing StarT-ng to run many more applications efficiently.

5.3 Improving StarT-ng's Shared Memory Performance

Since most of the shared memory support is built in software, the options for performance enhancement are limitless. The software L3 cache has already been mentioned. Here are a couple more possibilities.

Integrating Network Virtual Memory into StarT-ng: In order to fully exploit the large amount of compiler research aimed at automatically placing local-global data, the path to access local-global data should be made faster. One way to do that is to introduce temporary "localization" of a global page via the virtual memory system by using network virtual memory[16, 8]. This technique is however inadequate when there is finer grain sharing, where the ownership is better divided among different sites at the cache line level. For pages that fall within this category, we will rely on the cache line level cache coherency mechanism provided by the ACD and sP. Schemes to dynamically switch a page between coherency granularities will be investigated.

Reducing Application Processor Idle Time through Multithreading: Another way to avoid the long delays incurred by servicing a global cache-miss is to switch threads when a cache-miss occurs. When the cache-line is returned, the thread is restarted where it left off. Such a scheme is also required to cache memory locations with synchronizing semantics such as I-structures[4]. In such a scheme, the penalty of a cache-miss is simply the time to context switch out the thread and switch it back in later. There is also a potential cost of checking to see if there was a cache-miss. With special hardware support, this last cost can be eliminated.

Unfortunately, StarT-ng does not have special hardware support to detect and handle cache-misses. In order to implement this scheme, an efficient method to detect a cache-miss needs to be implemented in software. Our proposed scheme uses a *miss pattern*, that is, a pattern which is very unlikely to occur in real data. When a cache-miss occurs and the request is propagated to the memory bus, the ACD captures it and passes it to the service processor. If the desired cache-line is not in the L3 cache, the service processor returns a cache-line filled with the miss pattern through the ACD to the requesting processor. All accesses to memory where a miss pattern can be returned must be checked against the miss pattern. If the miss pattern is encountered, the thread must be swapped out pending the arrival of the desired data. Speculation and superscalar execution can probably remove most, if not all of the cycle overhead involved in this check.

When the requesting processor sees the miss pattern, it sends a request for the cache-line. The request includes information on how to restart the thread, save its current thread state away and schedule the next thread. Switching threads will be performed by code generated by the compiler. An upper bound on the amount of time to access global memory would be around 300 cycles. The request will generate 2 reply messages. The sP will receive the cache-line to insert into the L3 cache while the requesting processor gets back the exact word that was requested.

6 Related Work

StarT-ng is heavily influenced by dataflow architectures. Its message passing architecture is optimized for low-latency messaging rather than high-bandwidth messaging (although its bandwidth is very competitive.) Though it is a more difficult objective to achieve, low-latency messaging is essential for our compiler research. Machines that have influenced us in this area are the original *T project[20], MIT's Monsoon[19], ETL's EM-4[23], the J-Machine[7] and the M-Machine[12].

StarT-ng's software approach to cache-coherency is shared by other projects as well. The Wisconsin Wind Tunnel[21] (WWT) uses minimal hardware support[11] to implement shared memory. Network Virtual Memory[16, 15] (NVM) takes advantage of virtual-memory management hardware to maintain coherency on a page-granularity basis.

StarT-ng is remarkably similar in some ways to Typhoon[22], a paper architecture developed at Wisconsin. Typhoon, however, is not SMP based and

[11] They hijack the ECC bits and handlers rather than adding any additional hardware. Unfortunately, this strategy cannot be supported on more aggressive processor architectures which do not provide precise ECC exceptions.

proposes a much larger degree of custom hardware for its message passing and shared memory support than StarT-ng does.

Alewife[1] and FLASH[10] use varying degrees of software in their coherency processing. Alewife has hardware support for maintaining coherence, but traps to software for exceptional cases not supported in hardware. Each Alewife site consists of a modified SPARC 2 processor and a fully custom memory controller, the CMMU. Unlike StarT-ng, Alewife cannot use standard commercial software such as the operating system.

Cache-coherency on FLASH, like on StarT-ng, is maintained completely in software. That software, however, runs on a special piece of hardware, the Magic chip, which replaces the standard memory controller. The Magic chip is much more aggressive, technology-wise, than StarT-ng, and achieves better performance when a cache-miss occurs, but requires much more design effort, not only in the special hardware but in the system to use it. FLASH's shared memory design is conceptually cleaner since it avoids unnecessarily recrossing the L3 and thus eliminates some of the deadlock situations.

The Stanford DASH[14, 13] is similar to StarT-ng in that it uses SMPs as building blocks for parallel machines. It adds custom shared memory boards to provide cache-coherent shared memory across multiple SMP sites. Unlike StarT-ng, all the protocol processing is performed by hardware on the shared memory board. All the directory memory also resides on this board. DASH's shared memory implementation, unlike StarT-ng's, allows access to resident global memory to proceed without going through the shared memory board, unless coherency action has to be carried out on remote sites. However, the protocol is fixed in hardware, and dedicated memory is used to maintain directory information.

StarT-ng's approach to shared memory uses much simpler hardware than any of the hardware supported shared memory schemes that we have encountered, allows finer-grained coherency than NVM, and works with much more aggressive processors than the Wisconsin Wind Tunnel. We believe that this system will allow us to conduct interesting research into real problems using globally shared memory and running message passing codes with very competitive performance.

7 Current Status and Conclusions

StarT-ng's delivery schedule is broken into 3 phases. Phase 1 will be delivered to MIT at the end of 1995 and consist of 8 sites, each containing 4 NES boards. The Phase 1 NES boards will communicate with their NIU's at either a third or a half of the processor clock rate. The ACD will run at a fourth of the processor clock rate, forcing the memory bus to run at that speed when the ACD is enabled (and at L2 speeds otherwise). The ACD and NIU will be built from off-the-shelf parts (*e.g.*, FPGA's and dual-ported SRAM's). The 620 clock rate may have to be reduced slightly to accommodate the ACD and NIU.

Phase 2 will raise the 620 processor and bus to their maximum rated speed and the NIU clock at one half of the 620 clock. The ACD clock-speed remain a factor of 4 slower than the 620 necessitating the 620 L3 bus to run at that speed when the ACD is turned on. Phase 2 is due in the middle of 1996.

Phase 3 will boost ACD clock rate to one half of the 620 clock rate, making the SMP sites of the machine run at full commercial speeds even when the ACD is turned on. Phase 3 is currently planned for delivery perhaps in September of 1996, but will be influenced by results from experiments conducted on the phase 1 machine.

StarT-ng is a step away from networks of workstations, capturing most of their advantages while significantly out-performing them. StarT-ng will deliver very aggressive message passing performance and will provide mechanisms to experiment with cache-coherent shared memory. Intensive effort to develop simulators, compilers, operating system support and coherency protocols are already underway. StarT-ng should be a cost-effective, realistic platform for research as well as commercial parallel computing.

References

1. A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocesors*. Kluwer Academic Publishers, 1991.
2. B. S. Ang, Arvind, and D. Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. In *Proceedings of the ISCA '92 Dataflow Workshop, Hamilton Island, Australia (to be published)*, May 1992.
3. B. S. Ang, D. Chiou, and Arvind. Issues in Building a Cache-Coherent Distributed Shared Memory Machine using Commercial SMPs. CSG Memo 365, Laboratory for Computer Science, MIT, Cambridge MA, December 1994.
4. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.
5. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, A. Shaw, and Y. Zhou. Cilk: An efficient multithreaded runtime system. Submitted for publication, December 1994. Available via anonymous FTP from `theory.lcs.mit.edu` in `/pub/cilk/cilkpaper.ps.Z`.
6. G. A. Boughton. Arctic Routing Chip. In *Parallel Computer Routing and Communication: Proceedings of the First International Workshop, PCRCW '94*, volume 853 of *Lecture Notes in Computer Science*, pages 310–317. Springer-Verlag, May 1994.
7. W. J. Dally and et al. Architecture of a Message-Driven Processor. *IEEE Micro*, 12(2):23–39, 1992.
8. S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *4th. ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 1993.
9. E. Hagersten. *Toward Scalable Cache Only Memory Architectures*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, October 1992.
10. M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Charachorloo, D. Nakashira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architecture Support for Programming Languages and Operating Systems, San Jose, CA*, pages 274 – 285, October 1994.
11. C. F. Joerg. Design and implementation of a packet switched routing chip. Technical Report MIT/LCS/TR-482, Laboratory of Computer Science, Massachusetts Institute of Technology, December 1990.
12. S. W. Keckler and W. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213. ACM, 1992. ISCA '92-Gold Coast, Australia.
13. D. Lenoski, J. Laudon, K. Charachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocesor. In *Proceedings of The 17th Annual International Symposium on Computer Architecture, Seattle, WA*, pages 148–159, 1990.
14. 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, May 1992.
15. K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 86. (Also as YALE/DCS/RR-492).
16. K. Li, M. Stumm, D. Wortman, and S. Zhou. Shared Virtual Memory Accommodating Hetergeneity. CS-TR 210-89, Princeton University, Department of Computer Science, February 1989.
17. R. S. Nikhil. Id Reference Manual, Version 90.1. CSG Memo 284-2, Laboratory for Computer Science, MIT, Cambridge MA, September 1990.
18. R. S. Nikhil. Cid: A Parallel "Shared-memory" C for Distributed-memory Machines. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing, Ithaca, NY*, Lecture Notes in Computer Science. Springer-Verlag, August 1994.
19. G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. The MIT Press, 1991. Research Monograph in Parallel and Distributed Computing.
20. 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, November 1993.
21. S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconson Wind Tunnel: Virtual Prototyping of Parallel Computers. In *ACM SIGMETRICS*, May 1993.
22. 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, April 1994.
23. S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. *Proceedings of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 46–53, 1989.