LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

StarT-ng: Delivering Seamless Parallel Computing

CSG Memo 371
July 9, 1995

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

To appear in the Proceedings of EURO-PAR'95, Stockholm, Sweden.

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory 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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

StarT-ng: Delivering Seamless Parallel Computing

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

July 9, 1995

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. Every 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 delivered at about the time the base SMP is introduced into the marketplace. The ability to be both a collection of standard SMP and an aggressive message passing machine with coherent shared memory makes StarT-ng a good 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 for the node performance they deliver and supports too little off-the-shelf software. We are trying to solve the first problem by using implicitly parallel functional languages like Id[16, 5] and pH, and multithreaded languages such as Cid[17] and Cilk[6]. This paper, however, concentrates on StarT-ng, our solution to the second issue.

With personal computers (PC's) selling in the millions, mainstream computers have become commodities, resulting in lower computer prices, sped up product time tables, and rapid performance improvements. Parallel computers, on the other hand, have traditionally employed a lot of custom hardware and software. By the time the machine is ready, its processing node is generally a generation or two out-of-date, and a factor of two or more slower than the then current commercial microprocessors. The small customer bases and, therefore, small development teams cannot find and solve problems very quickly, making these custom machines and their software unreliable.

Coupling unreliability with the high cost of custom development, the general lack of shrink-wrapped software and the difficulty in writing custom applications, buying a parallel computer is difficult to justify. One would buy such a system only if one's 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[1], a joint project between MIT and Motorola, tries to address these problems. StarT-ng is based on a commercial symmetric multiprocessor (SMP) *system* that uses 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 same time the base SMP is introduced. StarT-ng, though a research machine, is *commercially* competitive in parallel performance as a message passing machine, and also runs stock sequential and SMP applications efficiently. StarT-ng extends the sharing of processor, memory and I/O resources made possible on a small scale by bus based SMP beyond the scaling constraints of buses.

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. Not only is our design constrained to using the stock PowerPC 620 microprocessor, which is optimized for sequential execution, but it cannot even change the system implementation in any significant way. Our design reuses all of the stock system implementation except for the boards carrying the processors. Observing these tight constraints while providing competitive performance is the topic of this paper.

Organization: In Section 2, we present an overview of the StarT-ng hardware. This is followed in Section 3 with a discussion of message passing support on StarT-ng. Section 4 discusses how shared memory is implemented on StarT-ng. Finally, we compare StarT-ng with some related work in Section 5 before concluding with the current status of the machine.

2 Overview of StarT-ng

A *site* in StarT-ng is a commercial PowerPC 620 SMP augmented with special hardware for message-passing and shared memory. The PowerPC 620 is a 64-bit, 4-way superscalar processor with a dedicated 128-bit wide L2 cache interface and a 128-bit wide L3 path to memory. It employs some of the most sophisticated techniques for pipelining instructions and memory management. It also has a novel feature that allows the processor to communicate with coprocessors over its L2 cache interface.

The StarT-ng SMP has 4 processor card slots that are connected to the main memory by a data crossbar. The crossbar has substantially better throughput than a traditional bus. In the commercial version, each processor card contains 2 processors and their L2 caches. StarT-ng replaces one to four of these processor cards with network-endpoint-subsystem (NES) cards, each containing a single 620 processor, 4 MBytes of L2 cache and a network interface unit (NIU). The NIU allows the 620 to communicate with an MIT-developed Arctic network router chip[7]. 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 at each site has an address capture device (ACD) which allows a designated processor at the site to monitor and respond to bus transactions. When used in this role, a processor is called a service processor(sP); when used to run application code, it is called an application processor (aP). The ACD and sP, collectively called the Shared Memory Unit (SMU), will be used to implement globally shared coherent memory, with coherence controlled at cache-line granularity. The ACD can be disabled when global shared memory is not needed making it completely invisible to the system, allowing all four processors at a site to serve as aPs. Since all

[1]StarT-ng is the latest incarnation of the *T or StarT project. For a history of the different versions of *T, see [4].

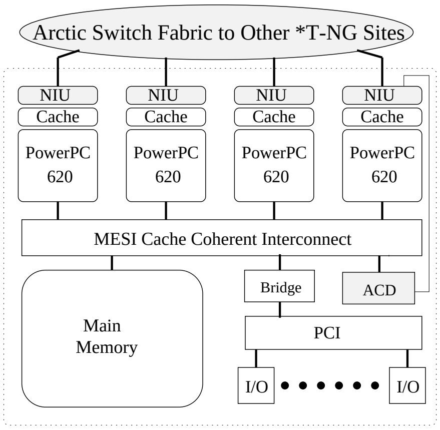

Figure 1: A StarT-ng site: the white areas comprise the base SMP. The grey areas are our additions.

NES boards will actually have ACD's, it may be possible (depending on motherboard specifics) to use more than one as an SMU, dividing up the global space between them.

3 Messaging Support

StarT-ng's user-mode message-passing capabilities are provided by a fat-tree network built from Arctic[7] routers, and accessed through a tightly-coupled hardware network interface unit (NIU) attached to each 620 processor's L2 *coprocessor interface*. The NIU's packet buffers can be memory-mapped into an application's address space, allowing user programs to send and receive messages without kernel intervention by directly manipulating the buffers. Standard communication protocols, such as TCP/IP, PVM, MPI and Active Messages can be easily and efficiently implemented over StarT-ng's networking facilities.

The Arctic routing chip designed at MIT is a 4-by-4 packet-switched router capable of implementing a variety of staged networks. Implemented in .6 micron CMOS gate-array, Arctic is expected to run at 50 MHz, delivering 400 MByte/sec/full-duplex-link at a latency of 6 Arctic cycles per hop. A full fat-tree with 32 end-points delivers close to 6.4 GB/s of bisection bandwidth, and has a maximum of 8 hops between two end-points, resulting in a network latency of less than 1 μs. Based on the approximate PowerPC 620 timings available to us at this time, each 620 processor can achieve a maximum bandwidth of 180MB/s for message receiving or 278 MB/s for message sending.

Arctic supports variable-sized messages of up to 96 bytes of which 8 bytes are routing, control and CRC overhead. It provides two virtual prioritized networks, allowing the implementation of two-priority deadlock-free protocols (often known as separate request-reply) on a single physical network. Arctic also enforces secure space partitioning and employs sophisticated buffer management that allows it to sustain close to its peak bandwidth. Link-level flow control is implemented in hardware. Extensive error checking is designed into Arctic, including Manchester encoding of link-level flow control signals, and 16-bit CRC for every packet. Error rates are, however, low enough so that error recovery is unnecessary under normal operating conditions. Arctic is designed with a set of commercial-quality test, control and error detection and recording features. It was necessary to design our own router because no commercial equivalent, in functionality or performance, was available to us. Further details about Arctic can be found in [7].

3.1 620 Coprocessor Interface

StarT-ng's fast messaging capabilities are built on the L2 coprocessor interface found in the PowerPC 620 processor, which provides a low-latency, high-bandwidth connection to memory-mapped slave devices. Coprocessor device interfaces are required to look exactly like L2 cache sRAM, including having the same read/write timing characteristics. Since the coprocessor is accessed using normal load/store operations, individual pages in the coprocessor region can be accessed either in an uncached, cached with write-through, or cached with write-back fashion, where caching refers to caching in L1.

There are tradeoffs between using uncached, write-back cached, and write-through cached accesses to the coprocessor interface. Accesses to the L2 interface, though partially pipelined, have latencies significantly longer than accesses to the L1 cache. Caching the coprocessor interface allows the L2 access latency to be amortized across an entire cache-line and allows burst transfers. But because the coprocessor devices are slave devices, the L2 interface is not automatically kept coherent. In order to read new data, the 620 must first explicitly flush the previously read cache-line. Write-back cached writes also require flushes to force data to the coprocessor and take advantage of burst transfers to the L2 interface. Write-through cached writes and uncached writes do not require flushes but may not use the L2 interface as efficiently. We intend to experiment with the actual machine to determine the most efficient approach.

A common way to transfer a message consisting of multiple words is to first transfer the data, then indicate commitment of the transaction. If commit is indicated by writing to a coprocessor register, the ordering of writes, as seen by the coprocessor, becomes crucial. The implementation must guarantee that the commit write is not visible to the coprocessor before the data transfer has completed. Though simple in older microprocessors, such a guarantee is complicated in the 620 due to its weak memory ordering, which only ensures that memory operations to the same location occur in program order. No ordering guarantee, however, is provided for operations to different memory locations. Modern microprocessors provide synchronization instructions, which block the execution of subsequent instructions until all the prior memory operations have completed, to solve this problem. Such instructions, however, can be expensive. There are some possible 620-specific techniques which will be tried that may allow us to eliminate many of the otherwise necessary synchronization.

3.2 Network Interface Unit (NIU) Architecture

The StarT-ng NIU interfaces to the 620 coprocessor interface through a dual-ported SRAM. The 620 interacts with the NIU by reading and writing to specific regions in the buffer. Generally, the 620 will poll the NIU by reading specific memory locations to see if messages have arrived. The user process, however, has the option of configuring the NIU to interrupt the 620 processor when certain conditions, such as the arrival of a certain class of message, occur. 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 message arrival interrupt is an easy way to signal a kernel message arrival.

As shown in Figure 2, the dual-ported buffer space is logically partitioned into four data regions and one status/control region. The status/control region, located on a separate page accessible only to kernel, contains a 32-bit control word and a 32-bit status word through which all relevant NIU internal states can be read and written. This is used by the kernel to initialize the NIU, and

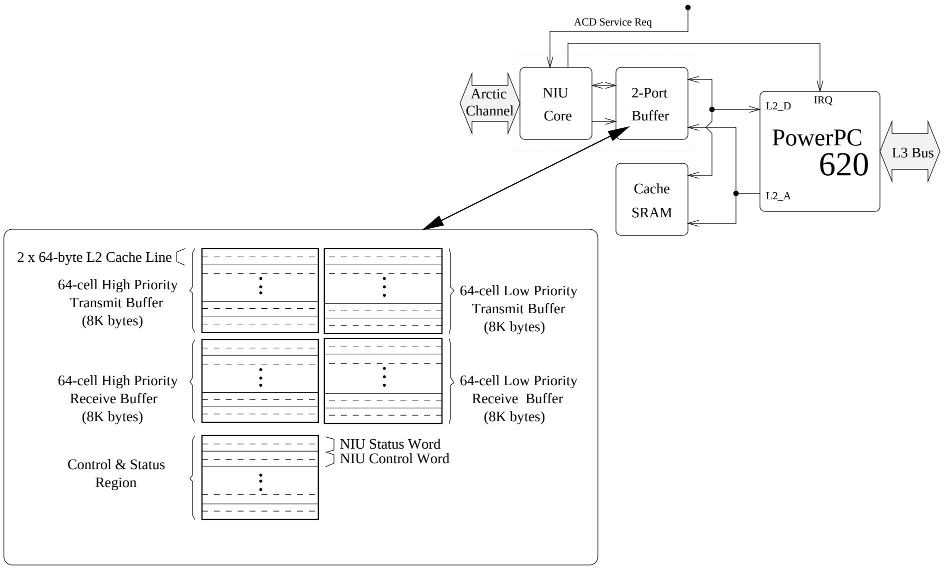

Figure 2: NIU/620 Configuration, with organization of interface buffers shown in box.

perform context switching. The status word also contains the ACD service request signal. Normal user-level message sends and receives do not require access to this region, thus enabling ordinary page access control to protect the NIU, without performance penalty, from user corruption.

The four data regions of the NIU interface allow receiving and transmitting messages at both high and low priorities. Each data region occupies two memory pages (8 KBytes), allowing independent specification of protection and caching. Each transmit and receive data region, subdivided into 64 packet cells of 128 bytes, is jointly managed by the 620 processor and the NIU as a circular queue. For the transmit buffers, the 620 processor acts as the producer of the queue while the NIU serves as the consumer. For the receive buffers, their roles are reversed.

A v-bit in each packet cell indicates whether it contains a valid message. The consumer polls the v-bit at the head of the circular queue. When the v-bit is *valid*, the consumer can proceed to retrieve the message from the cell, after which it frees the cell by resetting the v-bit to *invalid*. Prior to storing a new message into the queue, the producer first checks the v-bit of the cell it wishes to fill to ensure that the cell is free (v-bit invalid). After storing the message, the producer marks the v-bit *valid* to indicate to the consumer that the cell now holds valid data.

To handle timing asynchrony due to crossing of clock domains between the processors and the Arctic network, the NIU must first write the entire message, except the v-bit, into the receive buffer. The entire message actually includes the quad-word containing the v-bit; however, the v-bit is written as invalid. After a sufficient settling time, the v-bit alone is written, to the valid state. When reading messages from the sRAM, the NIU must first read the v-bit and, after it is valid, give sufficient settling time before reading the rest of the quad-word containing the v-bit.

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. The dual-ported sRAM and the v-bit scheme provide a bridge across the processor and network clock domains, handling all the meta-stability and race concerns.

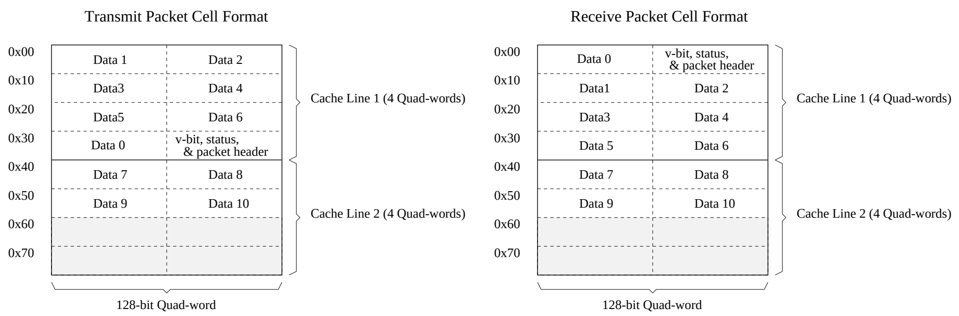

Figure 3: Transmit cell and receive cell packet formats.

3.3 Transmit and Receive Cell Formats

The v-bit handshake between the NIU and the 620 requires that the v-bit be written last by the producer, and read first by the consumer relative to the data that it guards. When the 620 accesses the NIU through a cached interface, data transfer between the 620 and the dual-ported sRAM occurs in multiple cycles in an order dictated by the 620. In order to make sure that the v-bit is written last to the sRAM, the transmit cells take on the awkward format shown on the left side of Figure 3, where the v-bit is in the last quad-word (128 bits) of the first cache line.

To further optimize the performance for uncached and write-through interfaces, the v-bit is placed in the last double-word of the last quad-word. This takes advantage of 620's *store-gather* capability, where two 64 bits stores to contiguous, ascending memory locations that occur one after another are packed into a single 128 bit transfer over the L2 interface. The v-bit and header are placed into the first cache-line of the transmit cell since smaller messages will only use one cache-line of the cell. For the receive cell, the v-bit is in the *first* quad-word of a receive packet cell (see right side of Figure 3), because the transfer of a cache line to the 620 starts by reading the first quad-word.

The StarT-ng NIU is optimized to support short, frequent messages, common in fine-grain parallel computation. The processor overhead of transmitting a 96-byte message (including an eight-byte header) by a user-level process using data already in its 620's registers is estimated at 42 cycles, assuming uncached access to the transmit buffers and buffer pointer already in register. Reading a 96-byte message takes 65 cycles under the same assumptions.

4 Shared Memory Support on StarT-ng

In addition to being a message passing machine, StarT-ng includes experimental support for building cache coherent shared memory. The main goals of this work are to explore: (i) the OS and virtual memory management (VMM) issues of a cache coherent distributed shared memory (CCDSM) system, (ii) hardware organization necessary to prevent deadlocks, and (iii) suitable memory models for programming. The emphasis in this research is on the necessary mechanisms to implement CCDSM correctly, rather than on the efficiency of the whole system.

4.1 Shared Memory Implementation

StarT-ng's cache-line coherent shared memory is implemented completely in software, allowing flexibility in the choice of coherence protocols. We plan to start with a simple directory-based,

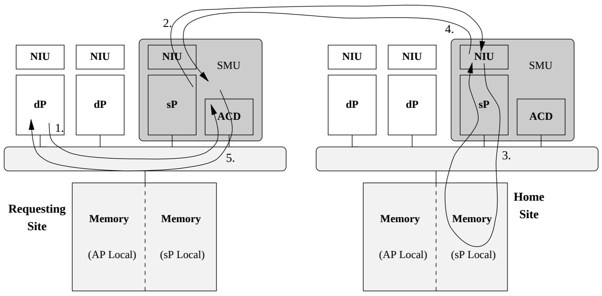

Figure 4: Servicing a global memory cache miss, assuming a clean copy is available at the home site and no remote coherence action is needed.

fixed home-site approach.

Figure 4 shows how a cache miss of a global location is serviced. The cache miss results in a bus operation that is claimed by the local SMU (step 1) acting like memory. A high order bit of the physical address space is used to distinguish between global and local address spaces (more on address spaces later), allowing the SMU to detect operations to a global location by examining the address. The physical address of a global location is further divided into two parts: a field indicating the address's home, and the remainder indicating the actual cache-line address. The home site field enables the SMU to forward the request to the home-site SMU (step 2) which maintains directory information and initiates the appropriate coherence actions. In our example, no further coherence action is needed. Thus, after updating the directory information and reading the cache-line from the local dRAM (step 3), the SMU returns the requested data (step 4) to the requesting site, where the SMU returns the data to the requesting processor (step 5). This example is, of course, a specific case. In general, coherence action request messages may have to be sent out to invalidate remote caches or flush a dirty cache-line to reclaim ownership. It is important to note that the details of the directory-based protocol are flexible since they are implemented in software/firmware by the SMU.

In StarT-ng, the SMU uses a 620 at each site as a service processor (sP) to provide the processing power. Using a 620 provides flexible and inexpensive implementation, since it is fully programmable and can share system resources. An ACD is provided to allow the sP to observe, initiate and respond to bus transactions. In our current design, the sP reads and writes the ACD over the L3 snoopy bus itself. Faster designs, which allow the sP to communicate to the ACD through the coprocessor interface, were examined but not chosen for the initial implementation to reduce design complexity.

The user applications on the aP's see two regions of virtual memory which translate to two distinct regions of physical memory: (i) aP local memory which is accessed through the local memory controller without the SMU's involvement; and (ii) global memory, which is mapped to the SMU. This distinction is made because global memory accesses that go through the ACD are slow, while many objects in parallel programs, such as program text and stack frames, are local.

The sP also sees two regions of memory: (i) sP local memory and (ii) the ACD command interface. All global address space handled through the ACD is eventually mapped to some sP's local address space. Although both the aP and sP local memory reside in dRAM accessed through the local memory controller, they must either be completely disjoint or shared in a non-cached fashion in order to avoid deadlocks.

Deadlocks were a serious concern in the design of StarT-ng's shared memory. Deadlock-free

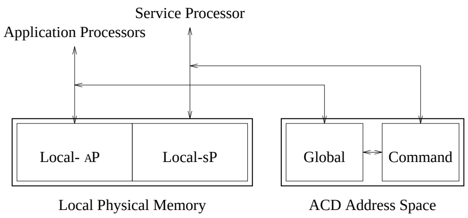

Figure 5: Physical Address Space Organization

implementation requires that the ACD be able to *selectively* flow-control requests due to lack of buffers, including software-based buffers in the sP. In particular, new cache-line read requests must be separated from write-back requests to avoid the possibility of reads consuming all buffering resources and causing deadlocks. Coherence-initiated cache-line flushes must be issued by the ACD and not the sP in order to avoid deadlocks. Deadlock issues are discussed in other papers[3, 2].

4.2 Access to Local-Global Memory

A *local-global* access is an access to a global location that has its home on the same site as the requesting processor. Our current design requires all global accesses, including local-global accesses, to be processed by the SMU in order to do the correct directory checks and maintenance. The SMU path overhead, however, is undesirable for local-global accesses, since the desired memory is local and often does not require remote coherence action. An all-software improvement, which we will try, would be to integrate network shared memory[15] and the cache-line level shared memory supported by the SMU (see Section 4.3). In the next paragraph, we discuss other local-global optimizations to StarT-ng which could not be implemented due to resource limitations.

One improvement has the SMU instruct the memory controller to deliver the desired data directly to the requesting processor once the directory check passes, bypassing the SMU during the return path. Yet another optimization modifies the memory controller to allow it to initiate dRAM access for local-global access but returns the cache-line only after the SMU determines and signals the memory controller that it is safe to do so. When data should come from a remote, dirty site, the SMU squashes the data read by the memory controller, and takes over the responsibility for returning the data. This scheme can be implemented without changing the memory controller by moving the filtering mechanism to the NES cards. Overall, however, it is probably more efficient to implement the SMU in the memory controller itself, which results in a FLASH-like design.

4.3 Operating System and Virtual Memory Management on StarT-ng

A major difference between StarT-ng and other CCDSM machines is in the OS and VMM. Some current CCDSM machines, such as Alewife, do not support virtual memory while others, like Dash, implement VMM with an SMP-like OS that has a single OS image and a single set of page tables for the entire machine. Each site of StarT-ng runs its own copy of an enhanced commercial SMP OS with its own site-local page tables. A message-passing-based paging layer is added to achieve inter-site global virtual memory.

The VMM implementation has two layers: local and global. The first layer is the standard SMP VMM, handling local memory, but is augmented to also cache information about global memory. Initially, all global pages are protected against any access. When an access to such a page is first

Type of Miss	StarT-ng (proc cycles)	FLASH (proc cycles)
L3 Hit	157	54
Local Clean	199	54
Local Dirty Remote	575	198
Remote Clean	520	202
Remote Dirty at Home	575	202
Remote Dirty Remote	955	250

Figure 6: Expected miss penalties excluding network latency in processor cycles. StarT-ng has a 133 MHz clock cycle while FLASH has a 200 MHz clock. The numbers for FLASH are taken from [10].

made on a site, the access is trapped and processed by the second layer which can either bring the page into local dRAM (sP local space), or provide a physical address translation if the page is already in another site's memory. In either case, the translation has to be set correctly so that the generated address falls into the AP global address space, and the home-site ID is in the appropriate field. The first layer VMM caches this until it is changed by the second layer.

This VMM approach will enable techniques similar to NVM[15, 9] that support coarse grain sharing and replication of pages by mapping global virtual pages into the aP's local physical memory, allowing accesses to those pages to bypass the SMU. StarT-ng therefore offers the flexibility of keeping coherence for global data at either page, or cache-line levels. At any time, each page has to be using only one scheme, but the selection can be changed dynamically, and independently for each page.

From the OS perspective StarT-ng looks like a high-speed network interconnection of multiple autonomous systems. This multiple OS image approach has significant advantages in fault tolerance; an OS crash at one site will not necessarily crash the other sites, killing only applications which depend on the crashed site. Another advantage is the fact that the SMP OS requires very minimal, if any, modifications. A third advantage is that TLB and other VMM-specific bus operations do not need to be broadcast across the entire machine whenever they occur. Finally, the use of site-local page tables offers software a choice of the granularity at which coherence is maintained. If desirable, it is easy to maintain page-level coherence, rather than the usual cache-line-level coherence, for selected pages.

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

This section presents estimated service times for global cache misses in StarT-ng. As noted earlier, the primary goal of shared memory support is *not* performance. The previous sections noted areas which could be improved but were not done for the StarT-ng implementation because of resource limitations. Not surprisingly, StarT-ng's shared memory performance is not particularly strong. Due to StarT-ng's very large caches (4 MB) and the improved locality due to its SMP nature, we hope cache miss rates will be low enough to make the coherent shared memory performance acceptable.

The penalties of cache-misses are shown in Figure 6. The times are given in processor cycles, and are approximate and conservative for StarT-ng. The penalties do not include network latencies, which is an orthogonal implementation issue. 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 are between 3 and 4 times longer than FLASH.

The actual impact of these numbers on performance depends on the miss rates and the percentage of memory operations in a program. When all other parameters are the same, a factor x increase in miss penalty requires a factor x decrease in miss rate to maintain the same overall run-time. Thus, if all parameters are the same, we would require a miss rate of between 3 and 4 lower than FLASH's to get to the same level of performance. The parameters are, however, not all the same. StarT-ng is based on SMP's which reduces the number of sites so that a larger fraction of references should be local-global. We also plan to make aggressive use of network virtual memory (mapping global pages to local pages) to further increase locality and reduce SMU utilization.

Large objects can be prefetched or steamed into a large software cache (L3 cache) maintained by the sP, further improving locality. Because the system continues to provide coherence maintenance, the user code can safely provide hints for prefetching based only on approximate information.

Another way to circumvent miss penalties is to switch threads when a cache-miss occurs. When the cache-line is returned, the thread is restarted where it left off. The penalty of a cache-miss is simply the time to swap out the thread and swap it in later, plus the cost of checking for cache-misses. Such a scheme is required to cache memory locations with synchronizing semantics such as I-structures[5].

Without special hardware support to detect and handle cache-misses, StarT-ng must implement this scheme in software. We plan to use a *miss pattern* as the returned data to indicate a cache miss. The application code tests all global loads to determine if a cache miss has occurred. The SMU returns the miss pattern after an access fails to hit in the L3 cache. If the miss pattern is encountered, the application thread sends a message to obtain the cache-line, then swaps itself out and schedules the next thread. The cache-line request includes information on how to restart the thread. The requested data is returned directly to the suspended thread and the cache-line to the sP for insertion into the L3 cache, ensuring that the scheme will work even if real data is equivalent to the miss pattern. An upper bound on the overhead of access to global memory would be around 300 cycles. Speculation and superscalar execution should remove most, if not all of the miss-checking overhead.

5 Related Work

StarT-ng is heavily influenced by dataflow architectures. Its message passing architecture emphasizes low-latency delivery of *small* messages rather than high-bandwidth transfer of large messages, although its bandwidth is very competitive. Achieving low overhead sending of small messages is a more difficult objective to achieve but allows finer granularity parallel execution. Machines that have influenced us in this area are the original *T project[19], MIT's Monsoon[18], ETL's EM-4[22], the J-Machine[8] and the M-Machine[11].

StarT-ng's software approach to cache-coherency is shared by other projects as well. The Wisconsin Wind Tunnel[20] (WWT) uses minimal hardware support[2] to implement shared memory. Network Virtual Memory[15, 14] (NVM) takes advantage of virtual-memory management hardware to maintain coherency at page-granularity.

StarT-ng is remarkably similar in some ways to Typhoon[21], an architecture developed at the University of Wisconsin. Typhoon, however, is not SMP based and proposes a much larger degree of custom hardware for its message passing and shared memory support than StarT-ng.

[2] 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.

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 operating systems.

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 than the SMU, and achieves better global cache-miss performance, but requires much more design effort both in the special hardware and in the system software to use it. FLASH's shared memory design is conceptually cleaner since it avoids unnecessarily recrossing the L3 and thus eliminates some potential deadlock situations.

The Stanford DASH[13, 12] 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 local-global memory to proceed like a purely local access, unless coherency action has to be carried out on remote sites. However, the protocol is fixed in hardware.

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 control than NVM, and works with much more aggressive processors than the Wisconsin Wind Tunnel. We believe that this system will be an extremely competitive message passing machine which will also enable research into global shared memory issues.

6 Current Status and Conclusions

StarT-ng's delivery schedule is partitioned into 3 phases. Phase 1, to be delivered to MIT at the beginning of 1996, will consist of a machine with 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, such as 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 to its maximum rated speed and the NIU clock to 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 an improvement over 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 underway. StarT-ng should be a cost-effective, realistic platform for research as well as commercial parallel computing.

References

[1] A. Agarwal et al. 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 and D. Chiou. Finding Deadlocks in Cache-Coherent Distributed Shared Memory Machines. In *Proceedings of the 1995 MIT Student Workshop on Scalable Computing, Wellesley, MA*, August 1995. (To appear).

[3] B. S. Ang et al. Issues in Building a Cache-Coherent Distributed Shared Memory Machines using Commercial SMPs. CSG Memo 365, Laboratory for Computer Science, MIT, Cambridge MA, December 1994.

[4] B. S. Ang et al. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. In *Advanced Topics in Dataflow Computing and Multithreading, IEEE Press*, 1995.

[5] Arvind et al. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[6] R. D. Blumofe et al. 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`.

[7] 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.

[8] W. J. Dally et al. Architecture of a Message-Driven Processor. *IEEE Micro*, 12(2):23–39, 1992.

[9] S. Dwarkadas et al. 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.

[10] M. Heinrich et al. 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] S. W. Keckler et al. Processor Coupling: Integrating Compile Time and Runtime Scheduling. In *Proceedings of The 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 202–213, 1992.

[12] D. Lenoski et al. 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.

[13] D. Lenoski et al. 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.

[14] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986. (Also as YALE/DCS/RR-492).

[15] K. Li et al. Shared Virtual Memory Accommodating Hetergeneity. CS-TR 210-89, Princeton University, Department of Computer Science, February 1989.

[16] R. S. Nikhil. Id Reference Manual, Version 90.1. CSG Memo 284-2, Laboratory for Computer Science, MIT, Cambridge MA, September 1990.

[17] 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.

[18] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* The MIT Press, 1991. Research Monograph in Parallel and Distributed Computing.

[19] G. M. Papadopoulos et al. *T: Integrated Building Blocks for Parallel Computing. In *Proceedings of Supercomputing '93, Portland, Oregon*, pages 624–635, November 1993.

[20] S. K. Reinhardt et al. The Wisconson Wind Tunnel: Virtual Prototyping of Parallel Computers. In *ACM SIGMETRICS*, May 1993.

[21] S. K. Reinhardt et al. 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.

[22] S. Sakai et al. An Architecture of a Dataflow Single Chip Processor. *Proceedings of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 46–53, 1989.