The NUMAchine Multiprocessor

Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley,
M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian Z. Zilic,
T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, S. Srbljic

Department of Electrical and Computer Engineering
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4

June 28, 1995

Abstract

NUMAchine is a cache-coherent shared-memory multiprocessor designed to have high-performance, be cost-effective, modular, and easy to program for efficient parallel execution. Processors, caches, and memory are distributed across a number of stations interconnected by a hierarchy of unidirectional bit-parallel rings. The simplicity of the interconnection network permits the use of wide datapaths at each node, and a novel scheme for routing packets between stations enables high-speed operation of the rings in order to reduce latency. The ring hierarchy provides useful features, such as efficient multicasting and order-preserving message transfers, which are exploited by the cache coherence protocol, for low-latency invalidation of shared data. The hardware is designed so that cache coherence traffic is restricted to localized sections of the machine whenever possible. NUMAchine is optimized for applications with good locality, and system software is designed to maximize locality. Results from detailed behavioral simulations to evaluate architectural tradeoffs indicate that a prototype implementation will perform well for a variety of parallel applications.

1 Introduction

Multiprocessors have existed for many years, but they have not achieved the level of success that many experts initially felt would be reached. The lack of stronger acceptance of multiprocessors is due in part to the following reasons: (1) an over-reliance on custom hardware solutions, making it difficult to track the rapid improvements in mainstream workstation technology, (2) a focus on scalability to thousands of processors, involving considerable up-front costs that preclude reasonably-priced small configurations, and (3) a lack of adequate system software, impeding development of application programs that can exploit the performance potential of the machines. These three factors have influenced our approach to multiprocessor architecture, as discussed below.

Multiprocessor systems designed using workstation technology can provide large computing capability at a reasonable cost. Future demand is likely to be the greatest for machines that give good performance and are modular, cost-effective, scalable to a reasonable size, and easy to use efficiently. A key requirement is that a multiprocessor system be viable and affordable in a relatively small configuration, which precludes a large up-front cost. However, it also must be easy to expand the system, necessitating a modular design. While scalability is an important issue, and has strongly influenced research in recent years, it is apparent that demand for huge machines (with thousands of processors) will continue to be low. Commercial interest is likely to be concentrated on designs that are scalable in the range of hundreds of processors.

From a user's perspective, it is desirable that a machine provide high performance and be easy to use, requiring little effort to structure programs for parallel execution. One way to facilitate ease of use is to provide a shared memory programming model with a single flat address space for all processors. This allows parallel programs to communicate by normal memory reads and writes, as opposed to communicating based on software message passing with its attendant overhead. In addition, by providing hardware-based cache-coherence for the shared memory, the task of developing parallel programs is simplified, both because programmers are given a familiar abstraction for accessing memory, and because it is simpler to create compilers that can automatically parallelize programs.

In order for multiprocessor technology to reach a much greater level of commercial success than is presently held, it is crucial that system software for multiprocessors evolve considerably beyond the current state-of-the-art. In order for this to occur, it is necessary that multiprocessor machines become available for use as software research platforms. Such a machine should allow a large degree of flexibility to allow software to control the hardware resources available in the machine.

This report presents the architecture of the NUMAchine multiprocessor and describes a 64-processor prototype that is being constructed. This hardware is part of a larger *NUMAchine project* that includes development of a new operating system, parallelizing compilers, a number of tools for aiding in correctness and parallel performance debugging, and a large set of applications[1]. The overall objectives of the NUMAchine project are to design a multiprocessor system that meets the criteria discussed above and is scalable in the range of hundreds of processors.

The NUMAchine architecture has many interesting features, the most important of which are listed below:

- *Simplicity of interconnection network.* NUMAchine has a hierarchical structure, with bus-based *stations* (containing processors and memory) interconnected by bit-parallel rings. The point-to-point nature of ring-based interconnection provides high-bandwidth communication paths that do not require complex wiring. Moreover, because of the simplicity of this structure, NUMAchine offers excellent modularity and ease of expandability.

- *Cache coherence.* A hardware-supported cache coherence scheme is used, which is efficient, inexpensive to implement, and has little or no impact on the time required by a memory module to service read and write requests. The protocol exploits the ordering and multicast properties of the interconnection network to optimize performance. It is scalable in that the number of state bits required per-cache line grows only logarithmically with the number of processors in the system. This means that the cost of modularity is low, where sufficient SRAM to allow for large configurations (of several hundred processors) involves a negligible extra cost compared to small configurations.

- *Natural multicast mechanism.* The ring hierarchy and a novel routing scheme allow efficient broadcasting. This is fully exploited by our coherence scheme and can also be exploited by software in several ways.

- *Hardware support for enhancing locality.* Each station includes a large *network cache* for data from remote memory modules. The network cache provides a number of attractive advantages that reduce latency of accesses to remote memory, including: 1) serving as a shared tertiary cache for the processors on the station, 2) obviating the need for snooping at stations, 3) combining multiple requests for the same cache line, and 4) serving as a target for broadcasts.

[1]Reports describing various aspects of the NUMAchine project can be obtained via the WWW from:`http://www.eecg.toronto.edu/EECG/RESEARCH/ParallelSys`.

- *Extensive monitoring support*. NUMAchine includes non-intrusive monitoring hardware in all of its main subsystems. Collected data is available to a running application so that adaptive run-time decisions can be made based upon the observed state of the machine.

- *Flexibility*. The NUMAchine hardware exposes many of the low-level capabilities of the hardware to software, allowing system software control over the low-level functions of the hardware. This facilitates experimentation into the interaction between software and hardware in the machine. As an example, software has the ability to bypass the hardware cache coherence mechanisms to take advantage of application-specific semantics and reduce coherence overhead. We have found that the hardware support needed to enable such software control is simple, and does not adversely affect the speed of common-case operations.

The overall NUMAchine project is still in an early phase. The hardware for an initial prototype using MIPS R4400 processors is currently being fabricated. A detailed behavioral simulator is being used to evaluate architectural tradeoffs and the expected performance for a prototype implementation. The final version of the prototype system, targeted for completion in 1996, will consist of 64 processors, connected by a two-level hierarchy of rings. Initial implementations of much of the system software for NUMAchine have been developed on hardware simulators and existing multiprocessor platforms.

The rest of this document provides more details on the NUMAchine architecture (and prototype) and is organized as follows: Section 2 provides an overview of the NUMAchine architecture, Section 4 presents the results of simulations to evaluate the architecture for a variety of parallel applications, Section 5 refers to some examples of related work, and Section 6 concludes.

2 Architectural Overview

NUMAchine is a shared memory multiprocessor with the memory distributed across the stations. A flat physical addressing scheme is used with a specific address range assigned to each station. All processors access all memory locations in the same manner. The time needed by a processor to access a given memory location depends upon the distance between the processor and the memory. Thus, the architecture is of NUMA (Non-Uniform Memory Access) type.

NUMAchine uses a ring-based hierarchical interconnection network. At the lowest level of the hierarchy it has stations that contain several processors. The stations are interconnected by bit-parallel rings, as shown in Figure 1. For simplicity, the figure shows only two levels of rings — *local* rings connected by a *central* ring. Our prototype machine will have 4 processors in each station, 4 stations per local ring and 4 local rings connected by a central ring.

The use of ring-based interconnection networks provides numerous advantages, including: (1) there is a unique path between any two points on the network, so that the ordering of packets is always maintained, (2) information can be sent from one point in the network to one or more other points, providing a natural multicast mechanism, and (3) a simple routing scheme can be used, allowing for high-speed operation of the rings. One of the key design features of NUMAchine is that the above strengths of ring-based networks are fully exploited to provide an efficient implementation of our cache coherence protocol, as described later. Finally, rings engender a modular design that minimizes the cost of small machine configurations, while allowing for relatively large systems.

The hierarchical structure in Figure 1 supports high throughput when communicating nodes lie within a localized part of the hierarchy, because many concurrent transfers can take place. Such is the case when there is a high degree of locality in data accesses, so that most transfers are within a station or between stations on the same local ring. The longest transfers traverse all levels of the hierarchy, but these transfer

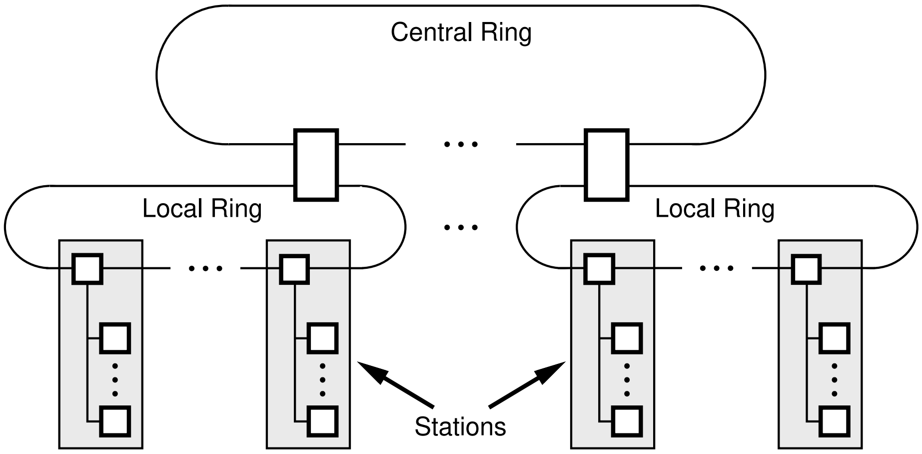


Figure 1: The NUMAchine hierarchy.

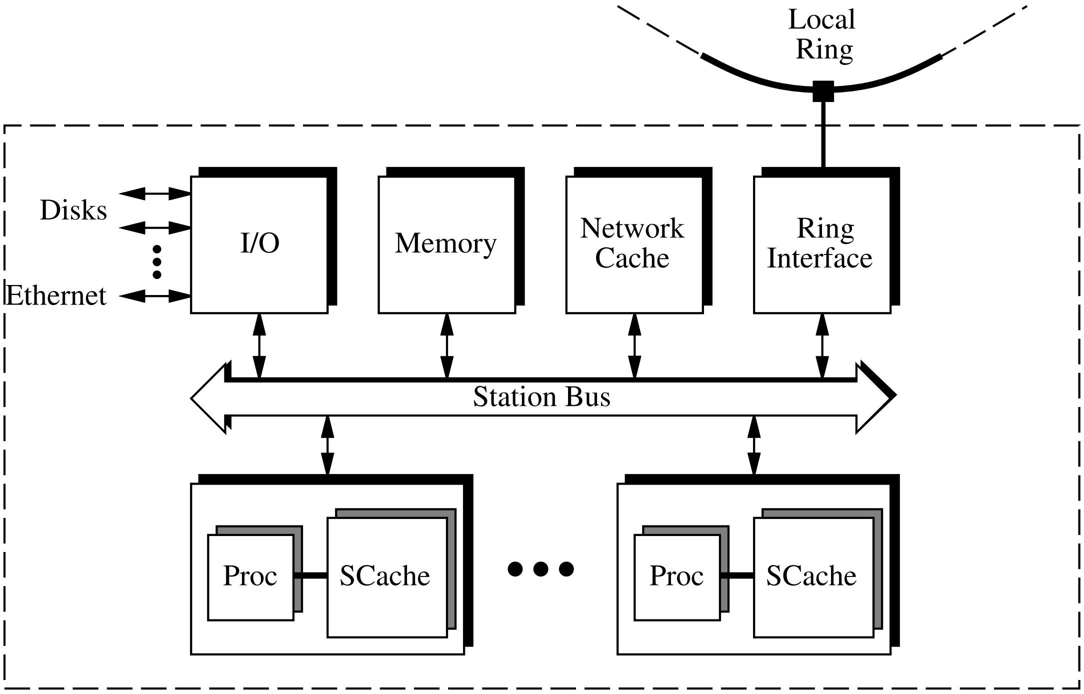


Figure 2: Station Organization.

times are considerably shorter than if all stations were connected by a single ring. An obvious drawback of the hierarchical structure is its limited bisection bandwidth, which means that software that does not exhibit locality may perform poorly. While there are some applications in which locality is inherently low, we believe that with sufficient operating system, compiler, and program development support, data locality can be high for a large class of applications.

Within each station, modules are interconnected by a single bus, as shown in Figure 2. A processor module contains a processor with an on-chip primary cache and an external secondary cache. Each memory module includes DRAM to store data and SRAM to hold status information about each cache line for use by the cache coherence protocol. The network cache is relatively large in size, and unlike the secondary caches, it uses DRAM to store data to allow for larger cache sizes at a reasonable cost. It also includes SRAM to store the tags and status information needed for cache coherence. The local ring interface contains buffers and circuitry needed to handle packets flowing between the station and the ring. The I/O module contains standard interfaces for connecting disks and other I/O devices.

The following subsection provides additional details on various aspects of the NUMAchine architecture, including the memory hierarchy, communications scheme, cache coherence protocol, and the procedure by which flow-control is maintained and deadlock avoided in NUMAchine.

2.1 Memory Hierarchy

The NUMAchine memory hierarchy consists of four levels with respect to a processor within a station. The primary on-chip processor cache is the closest level, followed by the external secondary SRAM cache. The next level consists of the DRAM memory located in the same station. This includes the memory module(s) for the physical address range assigned to the station, and the station's network cache, which is used as a cache for data whose home memory is in a remote station. The final level in the memory hierarchy consists of all memory modules that are in remote stations.

Within each station, processor modules share a centralized memory via the station bus. This arrangement has the advantage of centralizing cache coherence mechanisms within a station, which simplifies the memory system design. Furthermore, separating the processors from the memory permits the processor technology to be improved without affecting the rest of the system.[2]

Each station's network cache serves two related purposes: it caches data whose home memory location is in a remote station, and it confines cache coherence operations (as much as possible, according to the coherence protocol) for the remote data so that they are localized within the station. In addition, the network cache reduces network traffic by serving as a target for multicasts of remote data, and by combining multiple outstanding requests from the station for the same remote cache line. For simplicity, in our prototype machine the network cache is direct-mapped. Its design does not enforce inclusion of the data cached in the station's processor caches, but the size of the network cache, which is at least as large as the combined processor secondary caches, implies that inclusion in the network cache will usually exist.

2.2 Communication Scheme

The NUMAchine rings connect a number of nodes with unidirectional links that operate synchronously using a slotted-ring protocol. Each slot carries one packet and advances from node to node every clock cycle. The ring interface at each node contains a bidirectional link to a station or to another ring. To place a packet onto the ring, the ring interface waits for an empty slot. After removing a packet from the ring, the ring interface sends an empty slot to the next node.

Packets are used to transfer requests and responses between stations. A single transfer may consist of one or more packets, and may be of several types: cached and uncached reads and writes, multicasts, block transfers, invalidation and intervention requests, interrupts, and negative acknowledgements. All data transfers that do not include the contents of a cache line or a block require only a single packet. Cache line and block transfers require multiple packets. Since these packets are not necessarily in consecutive slots, they are assigned an identifier to enable reassembling the cache lines or blocks at the destination station.

The routing of packets through the NUMAchine ring hierarchy begins and ends at stations in the lowest level of the ring hierarchy. The unidirectional ring topology guarantees a unique routing path between

[2] Although the first working NUMAchine implementation will use the MIPS R4400 processor, decoupling the processors and memory in our implementation of the NUMAchine architecture permits switching to the MIPS R10000 processor when it becomes available.

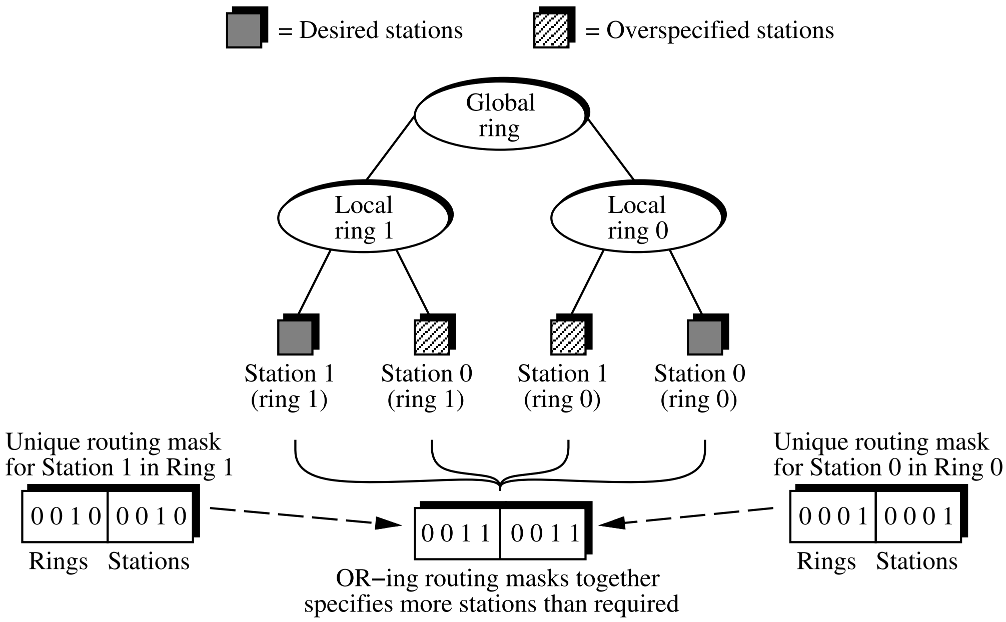


Figure 3: An example of an inexact routing mask.

any two stations. Station addresses are specified in packets by means of *routing masks*. Each *level* in the hierarchy has a corresponding bit field in the routing mask, and the number of bits in each field corresponds to the number of links to the lower level. For example, a two-level system consisting of a central ring connected to 4 local rings, with each local ring connected to 4 stations, requires two 4-bit fields in the routing mask; one field specifies a particular ring, and the other field indicates a specific station on that ring. The routing of packets through the levels of the hierarchy is determined by setting bits in the appropriate fields of the routing mask. Since a single field is used for each level of the hierarchy, the number of bits needed for routing grows logarithmically with the size of the system. In addition to specifying the path of packets through the ring hierarchy, the routing masks are also used in maintaining status information needed for the cache coherence protocol; the routing bits identify the locations which may have a copy of each cache line. The small size of the routing mask limits the storage cost for this status information.

When only one bit is set in each field of the routing mask, it uniquely identifies a single station for point-to-point communication. Multicast communication to more than one station is enabled by OR-ing bit masks for multiple destinations. As a result, more than one bit may be set in each field. Since a single field is used for each *level*, rather than individual fields for each ring at a given level, setting more than one bit per field may specify more stations than actually required. This is illustrated in Figure 3, which shows that when the bitmasks that specify station 0 on ring 0 and station 1 on ring 1 are OR'd, then station 1 on ring 0 and station 0 on ring 1 will also be sent the message. The imprecise nature of the routing bits results in some packets being routed to more stations than necessary, but the extra traffic generated under normal conditions (i.e. where data locality exists) is small and represents a good tradeoff for the savings involved (the significance of the savings is in both the number of bits needed per packet and, more importantly, in the number of coherence status bits needed *per* cache line).

The rules for routing packets in the ring hierarchy using the routing mask are simple. An ascending packet has at least one bit set in the field corresponding to the next higher level, and ring interfaces to higher-level rings always switch these packets up to the next level. Once the highest level specified by the routing mask is reached, the packet must descend. At each ring interface connected to a lower level of the hierarchy, the packet may potentially be switched down to the lower level if the bit corresponding to the

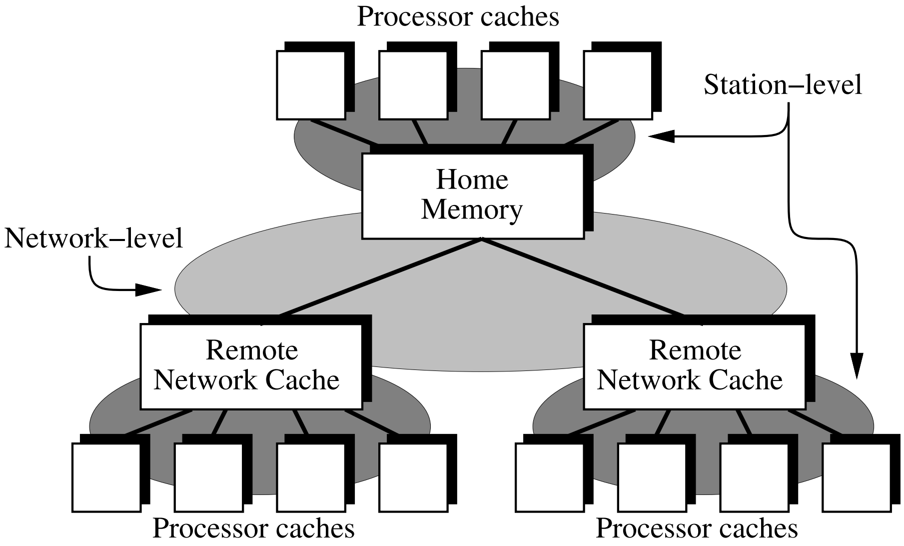


Figure 4: Two-level NUMAchine cache coherence protocol.

downward link is set to one in the routing mask. A copy of the packet may also be passed to the next ring interface at the same level if more than one bit is set in the same field. When a packet is switched downward to a lower level, all bits in the higher-level field are cleared to zero. The simplicity of this scheme permits a high-speed implementation, since only one field of the routing mask is involved in the routing decision at each ring interface.

2.3 Cache Coherence

This section describes details of the NUMAchine cache coherence protocol. Since cache coherence is highly complex, we necessarily cannot describe all possible coherence operations, but enough examples are presented to enable a knowledgeable reader to understand how NUMAchine's cache coherence operates.

NUMAchine's cache coherence scheme is optimized specifically for the NUMAchine architecture. In particular, it leverages the natural multicast mechanism available via the rings, it utilizes the feature that the rings provide a unique path from one node to any other node, and it is designed to localize coherence traffic to within a single level of the ring hierarchy whenever possible. The protocol is enforced simultaneously at two levels, as illustrated in Figure 4. *Network-level* coherence is enforced between the home memory module for a given cache line, and network caches in other stations holding copies of this cache line. *Station-level* coherence for a given cache line is enforced between the processor caches and the home memory on the same station or between the processor caches and the network cache on the same station if the home memory of the cache line is a remote station.

To maintain cache coherence at both the network and station levels, a hierarchical, two-level directory exists. The directory is stored in SRAM located in the memory module and network caches. At the network level, the home memory maintains a full directory of *routing masks* for each cache line. The routing mask can identify a single station or multiple stations as described in Section 2.2. In the directory, the routing mask indicates which stations may have copies of a cache line. At the station level, the directory consists of a simple *bitmask*, or *processor mask*, for each cache line. Since there is only a small number of processors per station, each processor has a dedicated bit in the processor mask. These bits indicate which processors on the station have a copy of the cache line. Processor masks that are stored in a memory module indicate which processors within the local station have a copy of a cache line. The processor masks for copies of

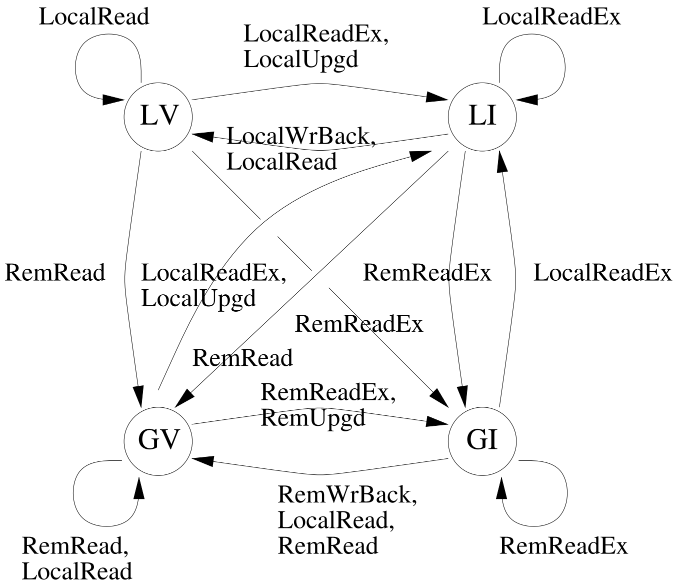


Figure 5: State Transition Tables for memory.

cache lines on remote stations are maintained in their respective network caches.

In addition to the directory, the memory and network caches contain a valid/invalid (V/I) bit per cache line, which indicates whether the copy they have is valid. The network caches also contain a local/global (L/G) bit, which indicates whether the only valid copies of the cache line are on the local station. In the memory module, a separate L/G bit is not needed because this information is provided by the routing mask in the directory.

While three basic states (dirty, shared and invalid) are defined for the secondary cache in the standard way for write-back invalidate protocols, four basic states are defined for a cache line in a memory module or a network cache. The L/G and V/I bits are used to indicate the state of the cache line and can have the following meanings: *local valid* (LV), *local invalid* (LI), *global valid* (GV) and *global invalid* (GI). The LV and LI states indicate that valid copies of the cache line exist only on this station. In the LV state, the memory (or network cache) as well as the secondary caches indicated by the processor mask have a valid copy. In the LI state, only one of the local secondary caches has a copy (which would be dirty), and the particular cache is identified by the processor mask. In GV, the memory (or network cache) has a valid copy of the cache line, and it is being shared by several stations, indicated by the routing mask in the directory. The meaning of the GI state differs slightly for the memory module and for the network cache. In both cases, the GI state means that there is no valid copy on this station. However, the GI state in the memory module also indicates that there exists a remote network cache (identified by the routing mask) with a copy in LV or LI state. Each of the basic states also has a *locked* version. The locked versions are used to prevent accesses to a cache line while the line is undergoing some transition. Any requests for a cache line in a locked state are negatively acknowledged, and the requester will try again.

The NUMAchine cache coherence protocol employs a write-back/invalidate scheme at both levels. The protocol is illustrated using four basic examples: local write, local read, remote read and remote write. The first three of these examples illustrate basic operation of the protocol by indicating how the directories and states are manipulated. The fourth example provides additional details by showing some of the actual steps taken in the hardware. For readers who are interested in the entire protocol, full state transition diagrams for a cache line in memory and for a cache line in a network cache are given in figures 5 and 6.

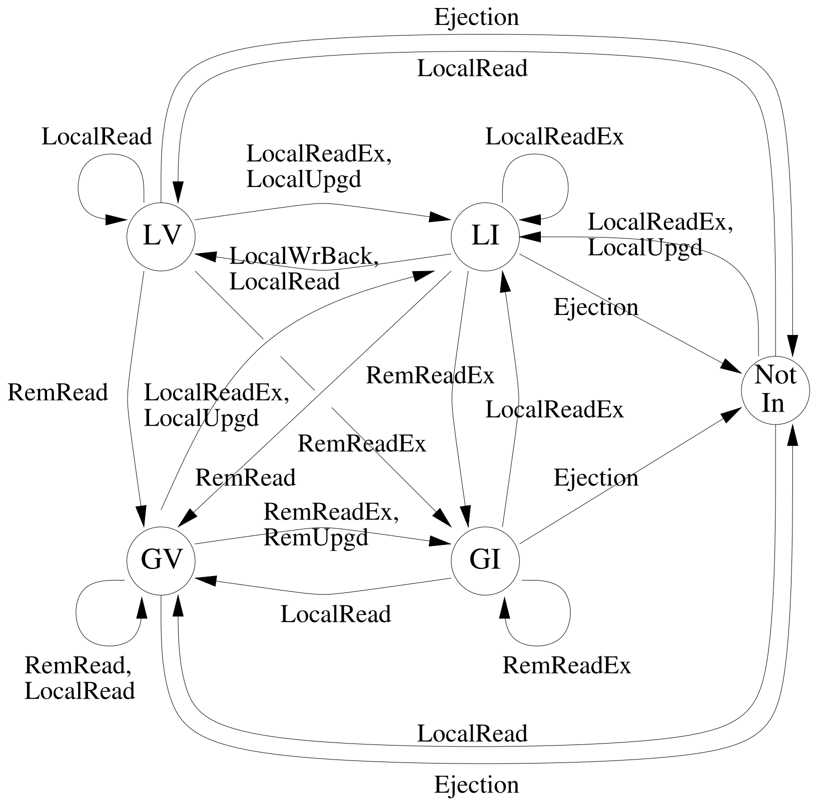


Figure 6: State Transition Tables for network cache.

Let us first consider a local write request by a processor on station Y, for a cache line whose home location is also on station Y. Let us assume that there are valid copies of the cache line on station Y and that the cache line is shared on another station, Z; therefore, the cache line is in the GV state in both the memory on station Y and the network cache on station Z. After the processor issues a write to memory on station Y, the memory controller will send an invalidate request to the remote station Z indicated by the routing mask, and to the local processors indicated by the processor mask in the directory. All the bits in the processor mask are reset except for the bit corresponding to the processor requesting the write. Also, the routing mask bits in the directory are set to indicate the local station. The new state of the cache line will be LI indicating that the memory no longer has a valid copy, but that the copy is in one of the secondary caches on the local station.

Upon receiving an invalidation packet, the remote network cache controller on station Z invalidates copies on the station according to its processor mask (if the cache line has been ejected from the NC, then the invalidation message is broadcast to all four processors), which is then cleared. The state of the cache line is set to GI, indicating that neither the network cache nor any of the secondary caches contain a valid copy of the cache line.

Let us now consider a read by a processor on station Y for the same cache line which is in the LI state in the memory module on station Y. The memory controller determines which processor has the dirty copy, and that processor then forwards a copy of the cache line to the requesting processor and to the memory module. Upon receiving the data, the memory controller writes it to DRAM and ORs the bit corresponding to the requesting processor to the processor mask in the directory. The new state of the cache line will be LV indicating that copies of the cache line are located on this station only. The memory and the processors

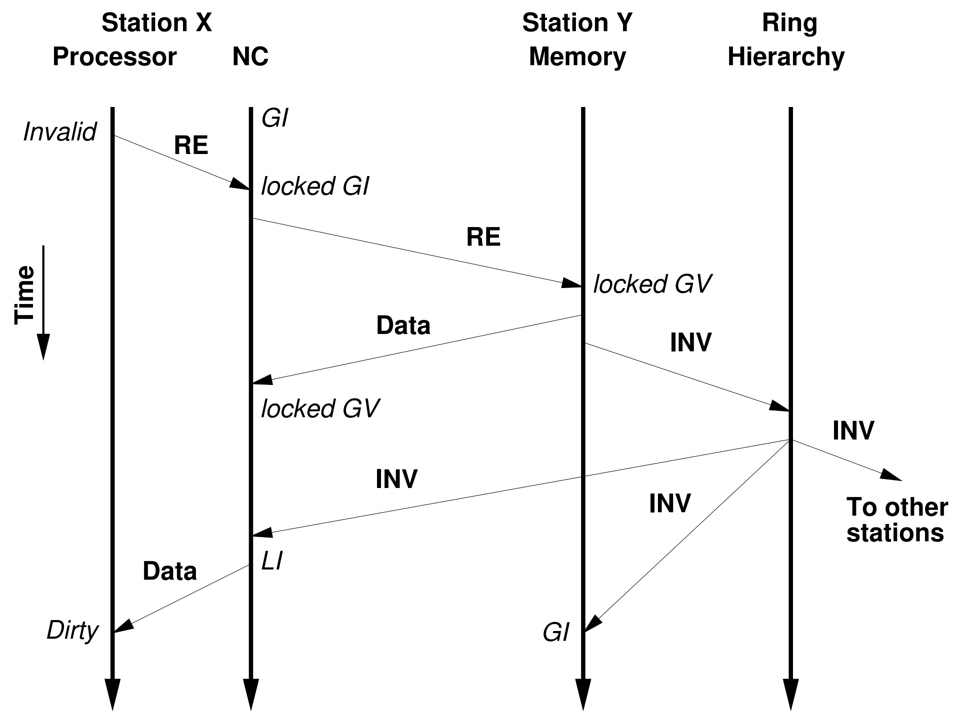


Figure 7: Coherence actions for a remote write.

indicated by the processor mask have valid copies of the cache line.

Next we consider the case where a shared read request issued by a processor on station X arrives at a memory module on station Y, where the cache line is in the GI state. In this example, we assume that the cache line is dirty on another station Z. We also assume that on station Z, the network cache entry for this cache line is in LI state. The home memory module sends a read request message (identifying the requesting processor on station X) to station Z using the routing mask. Using the information in its processor mask, the network cache on station Z obtains the dirty copy from the secondary cache, causing the state to change to GV in the network cache. The dirty data is forwarded to station X and a copy is also sent to the home memory module (in separate transmission). When the data arrives at station X, a copy is written to both the network cache and the requesting processor. In the network cache the state of the cache line is changed to GV and the processor mask is set to indicate the requesting processor. When the data arrives at the home memory module, it is written into DRAM. The existing routing mask in the memory is OR'ed with the bits corresponding to Stations X and Y, and the state of the cache line is changed to GV.

As a final example, we consider a write request by a processor on station X for a cache line whose home location is on station Y. In this final example we would also like to describe the locking mechanism that allows cache coherence and provides support for different consistency models. Figure 7 illustrates the necessary actions. Let us assume that the network cache state on station X is GI (i.e., there is no valid copy in the NC or in any of the processor on the station), and that the cache line is in GV state in the home memory. The processor's request goes first to the network cache on station X. The network cache locks this location and sends a write request packet to station Y (a *write request* means that the memory module should provide the data and give write permission). When the request reaches the home memory module, the data is sent to station X and all other copies are invalidated. The invalidation scheme is implemented as previously suggested in [11]. The home memory location is locked when the invalidate request packet is issued. The invalidate packet reaches the highest level of (sub)hierarchy needed to multicast it to stations with copies; it is then distributed according to the routing mask, which identifies all stations with valid copies, plus station X. When the invalidate packet returns to station Y (where it originated), the memory location is unlocked and placed in GI state, and the routing mask is updated to indicate station X as the only station with a copy. It is important to note that the invalidation requests do not have to be acknowledged by the caches that invalidate their copies of the cache line.

When the cache line reaches station X, the network cache writes it into its DRAM and waits for the invalidate packet to arrive. It is guaranteed that the data packet will arrive *before* the invalidate message, because the memory module sends the data first and the ring hierarchy preserves ordering of message. Upon arrival of the invalidate packet, the network cache sends the data from its DRAM to the requesting processor and puts the cache line into LI state. Also, the processor mask is set to indicate which processor on the station has the copy.

Some further aspects of the NUMAchine coherence protocol are summarized below.

- The protocol exploits the natural support for multicasting in the ring hierarchy for low overhead invalidation of shared data in remote stations. The routing mask in the network-level directory of the home memory specifies the stations that must receive an invalidation message. From the home station, a single invalidation message with the routing mask ascends the ring hierarchy to the first level from which all stations specified by the routing mask can be reached. After passing a sequencing point for ordering invalidation messages at that level, copies of the invalidation message then descend to all stations specified by the routing masks, including the home station and the station which issued the write request causing the invalidation. The sequencing point in each ring is the connection to a higher-level ring, except in the central, where one of the interfaces is designated as the sequencing point. The sequencing points and unique paths in the ring topology guarantee a global ordering of invalidations for different cache lines performed by different processors. These properties enable an efficient implementation of sequential consistency.

- The protocol supports a sequential consistency model by exploiting the order-preserving properties of the ring hierarchy [10]. The arrival of an invalidation message at the station that issued the corresponding write request serves as an acknowledgement and permits the write to proceed. As shown in Figure 7 the network cache receives the data and remains locked for that cache line until the invalidation returns. Upon receiving this invalidation, the data is sent to the requesting processor. Sequential consistency is ensured because the data response sent to the processor is globally ordered with all invalidations to other cache lines. The delay between receiving the data response and the invalidation can be very short if the data response and the invalidation message traverse the same path in the ring hierarchy. The delay can also be relatively long if the data comes from a nearby processor and the invalidation message must go to the top of the hierarchy. Simulation results show that the percentage of cases with large delays is insignificant to system performance; when compared to a system in which no locking mechanism for the purpose of consistency is used, only a 2% difference in overall performance was noted.

- The protocol combines routing masks for multiple stations to maintain a full directory with a cost that is bounded logarithmically by the number of processors in the system. This makes it possible to maintain a full directory entirely in hardware. When a station has exclusive access to a cache line, then the routing mask in the directory unambiguously indicates this station. When multiple stations have shared copies, the routing mask in the directory may specify more stations than actually have shared copies, due to the inexact nature of the routing masks. The protocol is designed to handle this imprecision when specifying multiple stations; the set of possible states in the network caches reduce the impact of this ambiguity on performance. For example, if an invalidation arrives at a network cache for a cache line in the GI state due to an ambiguous routing mask, then the invalidation will not be sent to any of the local processors.

- The protocol exploits the presence of the network cache to confine coherence operations within a remote station for both shared and exclusive data. In particular, the network cache avoids the latency of retrieving shared data from the remote home memory, which would otherwise be the case because contemporary microprocessor coherence protocols typically do not support cache-to-cache transfers of shared data (i.e., in response to a request for a cache line, the processor will respond with data only if the cache line is dirty). The LV and LI states in the network cache avoid sending write requests to the home memory because these states indicate that the only valid copies of data are located on this station.
- The protocol avoids unnecessary data transfers between stations for write permission requests whenever possible. When a processor requests write permission for a cache line for which it already has a shared copy, the home memory normally responds only with an acknowledgement, thereby avoiding the communication overhead of sending the data. However, the directory in the memory module may indicate that the shared copy was invalidated before the request arrived (the remote station had not yet seen the invalidation message when it sent the write permission request), in which case the home memory forwards the data to the requesting station, in order to avoid the latency of issuing a new request for the data. On the other hand, the directory may be ambiguous as to whether or not the requesting station still has a valid copy, due to the inexact nature of the routing masks. In this case, the protocol *optimistically* assumes that data is still valid at the remote station and responds only with an acknowledgement. While the latency increases if the assumption proves to be wrong, overall performance improves if the assumption is correct. Simulation results in Section 4 indicate that the assumption is correct in most cases.

In summary, the basic coherence mechanism for the NUMAchine cache coherence protocol is write-back/invalidate. This mechanism is the best choice for today's applications which are designed to exhibit as much locality in accessing data as possible. Since we efficiently enforce the strictest model of memory consistency (sequential consistency), our implementation also enables us to efficiently support any other memory consistency model that is supported by the processor/secondary cache subsystem. In order to make the protocol efficient, communication overhead must be minimized, which we have successfully achieved. At the same time, we have kept latency low for all cases except for the optimistic one described above which makes a decision in favor of low communication overhead at the expense of a slight increase in latency. Since latency can be tolerated using techniques such as prefetching, we have concentrated more strongly on reducing the communications overhead. Finally, the NUMAchine cache coherence protocol is conducive to low-cost implementation in hardware, because the amount of memory required for the directories is small and the logic circuitry needed to manipulate those directories is reasonable.

2.4 Deadlock Avoidance and Flow Control

NUMAchine ensures that deadlock will not occur by dealing with messages that elicit responses from the target stations differently from those that do not elicit a response; we refer to the former as *nonsinkable* and the latter as *sinkable*. Sinkable messages include read responses, write-backs, multicasts, and invalidation commands, while nonsinkable messages include all types of read requests (including interventions). To avoid deadlock, certain functional modules in NUMAchine use separate queues to hold sinkable and nonsinkable messages.[3]

[3] A single queue would suffice if one could guarantee that a nonsinkable message is received only when it can be fully processed without blocking subsequent sinkable messages.

The following rules govern the handling of sinkable and nonsinkable messages once they have entered the network:

- the ordering of sinkable messages is maintained,
- a downward path always exists for sinkable messages, and
- sinkable messages are given priority over nonsinkable messages.

For implementation purposes, an additional requirement is that the number of nonsinkable messages in the system is bounded. This is guaranteed because the number of nonsinkable messages that are issued from a station is limited by the local ring interface (up to 16 in our prototype). This bound implies that the size of some of the nonsinkable queues in NUMAchine grows as a linear function of the number of stations. Although this is not a scalable approach, it does not impose any practical limitations for the target system sizes. (An alternative scalable strategy is to negatively acknowledge nonsinkable messages when a queue becomes full, turning a nonsinkable message into a sinkable one.) For example, a queue size of 32KBytes per ring interface would be sufficient to handle the nonsinkable messages in a system with one thousand processors.

In addition to the deadlock avoidance scheme described above, NUMAchine has a simple flow control mechanism optimized for expected bandwidth requirements. Each ring interface has an input queue large enough to handle short term traffic bursts. Whenever the input queue is close to capacity, the operation of the ring that is feeding the buffer is temporarily halted; other rings are unaffected and continue to operate. Meanwhile, packets at the head of the full queue are processed until it is empty enough to start up the ring again. The flow control ensures that the order of sinkable requests is maintained, and it can be shown that this allows for important common case optimizations to the coherence protocol. However, it can result in poor performance under extreme conditions, such as when many processors simultaneously flush modified data from their caches to remote memory.

3 Prototype Design Details

The NUMAchine prototype (currently under construction) consists of a 64-processor system using two levels of rings, with the 4x4 geometry shown in Figure 8. The initial CPU boards will utilize 150 MHz MIPS R4400 processors, with 1 MB of L2 cache. The Network Cache on each station will contain at least 4 MB of memory.

3.1 System Modules

In the following subsections, the various system modules in the prototype are discussed, followed by a description of the hardware support for monitoring. All modules are described at the block diagram level; we do not provide details of the actual hardware circuitry.

3.1.1 Processor Module

Figure 9 provides a block diagram of a processor module. It contains a MIPS R4400 processor (this will likely be changed to the MIPS R10000 processor when it becomes available), and a 1-MByte secondary cache. The R4400 requires that the user provide (complex) circuitry to handle signalling between the processor and the rest of the system; this circuitry is called the *external agent* in the figure. The external agent handles formatting of all data and commands in both directions: those that are generated by the processor, and those being sent to the processor. The normal path for information flowing between the

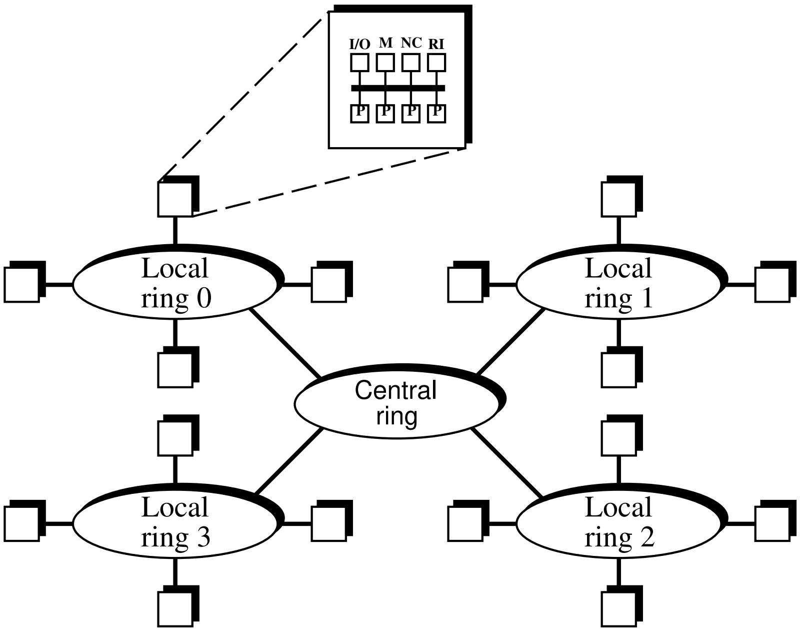


Figure 8: Geometry for prototype machine. Each station contains 4 processors.

processor and the NUMAchine bus is through the FIFOs shown in the figure. The FIFOs are included in the design because they allow for efficient communication over the bus, since the FIFOs allow the processor module to be ready to receive data even if the R4400 itself is not ready for an external request. The *bypass register* in Figure 9 allows the outgoing FIFO to be bypassed for certain operations, but we will not explain the details of its usage here.

The *Bus Interface* block handles flow of data between the FIFOs and the NUMAchine bus (which uses the mechanical and electrical specifications of the Future Bus standard, but employs custom-designed control). Bus Interface also performs arbitration when the processor module wishes to communicate with any other module over the NUMAchine bus. The other blocks in the processor module are for anxillary purposes, as explained below:

- *Local Bus Interface*. This allows for connection of miscellaneous devices that will be used for initial debugging. The local bus interface connects the external agent to a bus that is local to the processor card only. Connected to this bus are a ROM (for providing initialization code for the R4400 to execute), and an interface, called *Gizmo Interface*, for connecting to a microcomputer board. The microcomputer board is employed as a simple and convenient way of providing access to I/O devices, like UARTs and ethernet. Note that the Gizmo Interface will be used only during initial debugging, since each NUMAchine station will have a dedicated card for I/O devices.

- *Monitoring, Interrupt and Barrier Registers*. Various types of monitoring circuits for measuring performance with respect to the processor module are provided in this block. Also, for convenience, two registers that are not part of the monitoring circuitry are situated in this module. The *Interrupt Register* is used to send an interrupt to the R4400; it allows a number of modules in the NUMAchine prototype to interrupt the R4400 processor.

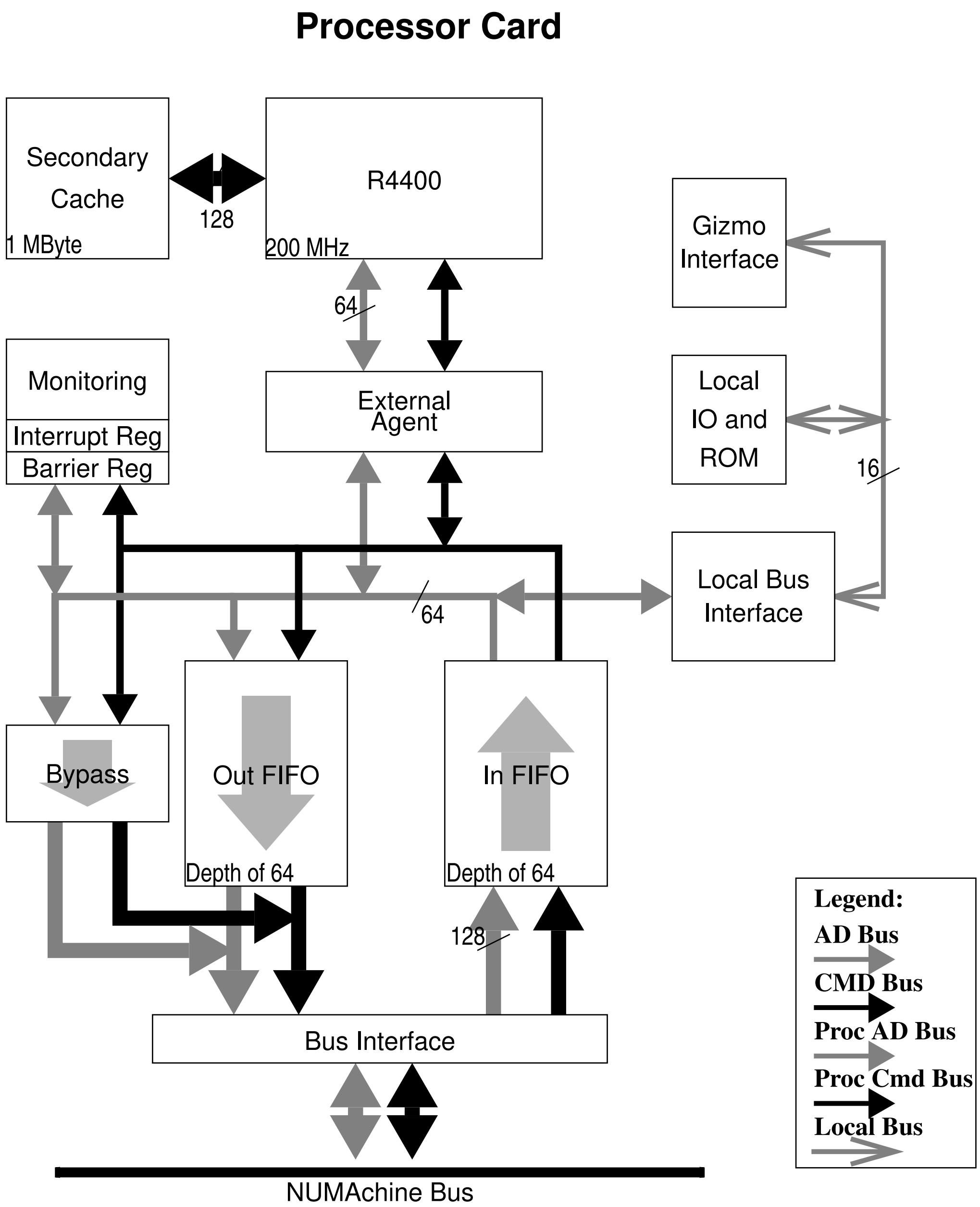


Figure 9: A NUMAchine Processor Module.

3.1.2 Memory Module

A block diagram of a NUMAchine memory module appears in Figure 10. Data and commands enter and leave the memory module through FIFOs, in a similar manner as described above for a processor module. The main control circuitry in the memory module is called the *Master Controller*, which controls reading and writing to the FIFOs (on the side opposite from the bus) and which controls the other functional blocks in the memory module. The DRAM block contains up to 2 GBytes of memory and a DRAM controller; the memory is split into two banks and is interleaved. The DRAM controller supports accesses by cache lines,

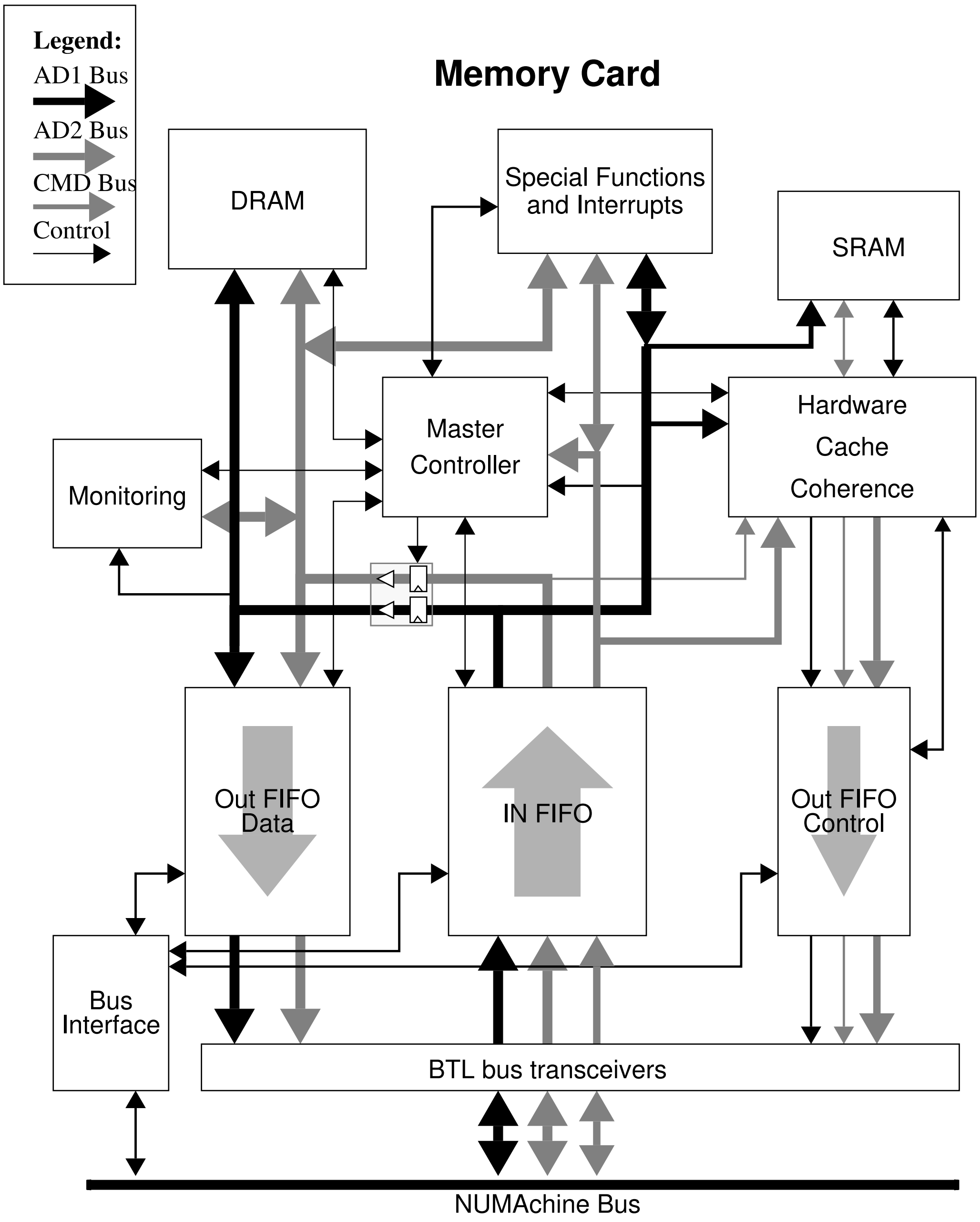


Figure 10: A NUMAchine Memory Module.

and also allows access to individual bytes, words, etc.

The *Hardware Cache Coherence* block maintains the cache coherence directories in SRAM. Cache Coherence actions take place in parallel with DRAM activity and are synchronized (via the Master Controller) whenever necessary. The cache coherence block implements all of the coherence actions and state transitions for cache-line status bits needed for the NUMAchine cache coherence protocol, as described in Section 2.3.

The *Special Functions and Interrupts* block provides operations in addition to normal memory access commands. Examples of special functions are block transfers of data from DRAM, kill operations for a

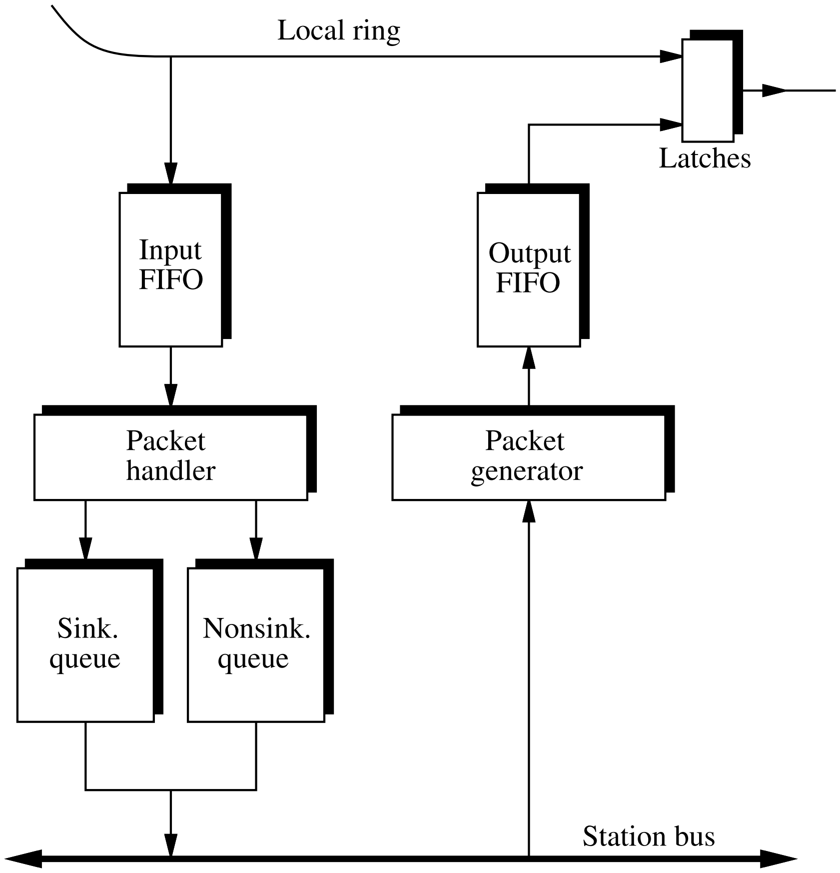


Figure 11: Local Ring Interface.

range of cache lines, writes directly to SRAM (which allows system software to bypass the hardware cache coherence actions), etc. This block also contains circuitry for forming interrupt packets, so that the memory module can send an interrupt to a processor, either because of an error condition, or to indicate completion of a special function.

3.1.3 Ring Interfaces

Two types of ring interfaces are needed in the NUMAchine architecture. The *local ring interface* provides the functionality needed to transmit messages to and from a given station and its associated local ring. This includes the capability for formatting of outgoing packets and interpretation of incoming packets. The *inter-ring interface* is much simpler, since it merely acts as a switch between two levels of rings in the hierarchy.

Local Ring Interface A block diagram of the local ring interface is depicted in Figure 11. Its upward path (to the ring) consists of a packet generator, an output FIFO, and a latch. The *packet generator* transforms incoming bus transactions into one or more ring packets and places them into the output FIFO. If the message must be split into multiple packets, then a distinct tag is assigned to each outgoing packet to enable re-assembly at the destination. Packets in the output FIFO are placed onto the ring as slots become available.

The downward path consists of an input FIFO, a packet handler, a sinkable queue, and a nonsinkable queue. The input FIFO is used as a buffer between the high speed ring and the packet handler. Since a slotted-ring protocol is used to transfer packets on rings, a cache line is not necessarily transferred in packets that occupy consecutive slots. As a result, the *packet handler's* primary task is to reassemble messages, such as cache lines, based on the tags assigned when the ring packets were sent.

Inter-Ring Interface Both upward and downward paths of the inter-ring interface are implemented with simple FIFO buffers. They are needed because a packet that has to go from one ring to another can do so only when an empty slot is available on the target ring. These buffers must be large enough to accommodate bursts where many consecutive packets on one ring have to go to the next level ring. In simulations of our prototype machine these buffers never contain more than 60 packets.

Routing decisions in the inter-ring interface are very simple in our communications protocol. Because of this simplicity it is feasible to operate the higher-level rings at higher speed, which might be a pragmatic approach if bisection bandwidth were to prove to be an issue in large systems.

3.1.4 Network Cache

The network cache (NC) is shared by all processors in a station and is used to cache data originating from other stations. The cache lines are maintained in DRAM so that very large network caches can be implemented at reasonable cost. The NC should be at least as large as the combined capacities of the secondary caches on the station, and can be made larger. SRAM is used to maintain status and control information of each cache line so that it can be accessed quickly.

The NC serves a number of useful purposes. It acts as a shared tertiary cache for the station, as a target for broadcasts, and as a target for prefetching when the processor does not support prefetching directly into its primary or secondary caches. It also performs a function akin to snooping, which is usually found in bus-based systems. In this section, uses of the NC are described. Section 4 will show the effectiveness of the NC, based on simulations of our NUMAchine prototype.

A read request to non-local memory is always directed to the local NC. If the cache line is present in the NC, then the NC responds with the data. If the NC knows that the cache line is dirty in a secondary cache on the station, it causes the data to be transferred to the requester. Otherwise, the NC sends a read request to the home memory module. When the data arrives from the remote station, it is forwarded to the requesting processor and a copy is kept in the NC. Subsequent read requests for the same cache line by another processor on the station are satisfied by the network cache, avoiding remote memory accesses. In effect, the NC replicates shared data from remote memories into the station. This feature is referred to as the *migration effect* of the NC.

The NC retains shared data that is overwritten in a processor's secondary cache, if the new cache line does not map into the same location in the NC as the cache line that is overwritten. Also, dirty data ejected from a processor's secondary cache due to limited capacity or conflicts is written back into the network cache, but not necessarily to the remote memory. If such data is needed again, it will be available from the network cache. This feature of the NC is referred to as the *caching effect*.

The NC "combines" concurrent read requests to the same remote memory location into a single request that propagates through the network to the remote home memory module. This occurs as a direct consequence of locking the location reserved for the desired cache line; subsequent requests for the locked line are negatively acknowledged, forcing the processors to try again. After the response to the initial request arrives, subsequent requests are satisfied by the NC. In this respect, the NC reduces network traffic and alleviates contention at the remote memory module. This feature is referred to as the *combining effect* of the NC.

The NC localizes coherence traffic for cache lines used only within a station but whose home location is in a remote memory module. Such lines exist in either LV or LI states in the NC, and all coherence actions for these lines involve only the NC and not the remote home memory module. For example, assume that a dirty cache line exists in a secondary cache and that its state in the network cache is LI. If another processor on the same station reads this cache line, then the NC determines from its processor mask which processor has the dirty copy and that processor sends the data to both the requesting processor and the NC. The state in the NC now becomes LV. If one of these two processors later requests exclusive access to the same cache line, the line becomes dirty again, and the NC invalidates the other copy. The state of the line in the NC becomes LI. All this is done locally, without having to send any messages to the home memory, which maintains the cache line in the GI state. This feature is referred to as the *coherence localization effect* of the NC.

The network cache is a convenient target for broadcasts (multicasts). Data produced by one processor, and needed by other processors can be broadcast, to avoid hot-spot contention at memory modules and in the interconnection network. Other possibilities for broadcast targets are less attractive: broadcasting into secondary caches requires complicated hardware on each processor and can eject data in use by the processor; broadcasting into memory modules is impractical for addressing reasons.

The NC can also be used for prefetching data if the processor does not support prefetching directly. Prefetching can be implemented easily as a "write" request to a special memory address which causes the hardware to initiate the prefetch [16]. The disadvantage of prefetching into the network cache is that the data is not placed as close as possible to the requesting processor.

The use of the NC obviates the need for snooping on the station bus, saving cost and reducing hardware complexity. A processor can obtain remote data from the NC instead of obtaining it from another processor's secondary cache through snooping. In fact, the NC provides complete snooping functionality. It responds with shared data directly, and causes forwarding of dirty data as explained above. This functionality may not be easy to achieve using snooping, because many modern processors make it difficult to implement a mechanism that allows shared data to be obtained from a secondary cache.

3.2 Hardware/Software Interaction

We have chosen to give software access to the low-level capabilities of our hardware. This low-level control, in conjunction with the natural multicast capability of our interconnect, allows system software to provide applications with a rich set of features. We first describe some of the low-level control that is provided to system software, and then briefly describe some of the capabilities this control gives to applications and system software.

- System software can (bypassing the coherence protocol of the hardware) read and write the tags, state information, and data of: any memory module in the system, any network cache, and the local secondary cache. These accesses can be performed atomically with respect to coherence actions by the hardware and with respect to other such accesses by software.[4] The data of the caches can be accessed either by index or by address.

- A processor can request any memory in the system to: invalidate shared copies of any of its cache lines, kill dirty copies, and obtain (at memory) a clean exclusive copy. Similarly, a processor can request any network cache to: invalidate any shared copies of cache lines local to the station, kill a dirty local copy, prefetch a cache line that will soon be accessed, or write a dirty cache line back to (remote) memory.

- Some of the above operations (e.g. requests to set the state of a cache line, invalidate a cache line, and write-back a dirty cache line) can be done using a block operation that affects all cache lines in a

[4] Atomic access is performed using `read_with_lock` and `write_with_unlock` operations. The lock acquired is the per-cache line lock used by the coherence protocol, allowing system software to manipulate cache lines even while they are being accessed by application software.

range of physical memory. For example, a single request to a network cache can be used to invalidate the locally cached copies of data belonging to a sequence of physical pages. (The initiating processor receives an interrupt when the requested operation has completed.) Also, a block prefetch request can be made to the network cache, which will asynchronously prefetch the requested block of data from remote memory.

- NUMAchine supports efficient coherent memory-to-memory block copy operations, where the unit of transfer is a multiple, n, of the cache line size. The request to copy a region of memory is made to the target memory module, which for each n cache lines: kills any existing cached copies of the cache lines, and makes a request to the source memory module to transfer them. The source memory module collects any outstanding dirty copies of affected cache lines from secondary caches, and transfers the data to the target memory using a single large request of n cache lines. An efficient block transfer capability facilitates page migration and replication, which are used in NUMA systems to improve performance through improved locality [7, 14].

- System software can directly specify some of the fields of packets generated in the processor module. This can be used to implement special commands and to multicast packets to many targets by specifying the routing mask used for distributing the packet. For example, software can supply a routing mask to be used for write-back packets, causing subsequent write-backs of cache lines from the secondary cache to be multicast directly to the set of network caches specified by the routing mask (as well as to memory).

- Each processor module has two interrupt registers, one for cross-processor interrupts and one for device interrupts. Data written to an interrupt register is ORed with current contents of the register, and causes an interrupt. When the processor reads an interrupt register, the register is automatically cleared. For cross-processor interrupts, the requesting processor identifies the location of the target register(s) using a routing mask and a per-station processor bit mask. This allows an interrupt to be multicast to any number of processors, which may be used for an efficient implementation of TLB shoot-down [2] or other coordinated OS activities [19]. On a request to an I/O device, system software can specify the processor to be interrupted as well as the bit pattern to be written to the processor's interrupt register when the request has completed. This flexibility allows a processor to efficiently handle concurrent requests (such as I/O requests and memory-to-memory transfers) involving multiple devices distributed across the system.

- The performance of SPMD applications that barrier often depends on the efficiency of the barrier implementation [18]. To efficiently support barrier synchronization, each processor module has a barrier register that differs from the interrupt register only in that writes to the barrier register do not cause an interrupt. In a simple use of these registers, when a processor reaches a barrier it multicasts a request that sets a bit corresponding to its ID in the barrier register of each of the participating processors, and then spins on its local register until all participating processors have written their bit.

- At system boot time, the latency and bandwidth of all components of the system (e.g., network components, processors) can be constrained. This will alow practical experimentation to determine the effect of different relative performance between system components, such as processor speed, network latency, and network bandwidth.

While much of the functionality that results from the above control is obvious, sophisticated application and operating system software can make use of this control in a number of non-obvious ways. In the remainder of this section we give three non-trivial examples of how this control could be used.

Update of shared data

Consider the case where many processors are spinning on a data element (e.g., a eureka variable [18]) and some processor writes that data. With a write invalidate protocol, when the processor modifies the data all the shared copies of the data are invalidated. Hence, data accessed in this fashion involves both a large latency to make the modification and contention at the memory module when the spinning processors obtain a new copy. With the above control, software can instead temporarily bypass the hardware coherence, modifying shared data and multicasting it to the affected network caches without first invalidating the shared copies.

In particular, the system software interacts with the hardware to: 1) obtain the routing mask of network caches at stations caching the data, 2) lock the cache line to ensure that additional stations are not granted access to it, 3) modify the state of the cache line in the secondary cache to dirty, 4) modify the contents of the cache line in the secondary cache, and 5) multicast the cache lines using the routing mask obtained earlier. When the updates arrive at a network cache, the network cache invalidates any copies in local secondary caches. When the update arrives at memory, the cache line is unlocked.

Software managed caching

NUMAchine allows system software a fair bit of control over how data is cached and how coherence is maintained. At the simplest level, system software can specify on a per-page basis: (i) if caching should be disabled or enabled, (ii) if the coherence of cached data should be enforced by hardware, (iii) if hardware should allow multiple processors to have data in a shared state (or only allow exclusive access by a single cache), and (iv) if the processor supports it, if coherence should be maintained using an update or invalidate protocol.

We are currently evaluating supporting both sequential consistency, and a weaker model (that doesn't quite fit any of the established weak consistency definitions). The full overhead of this is not yet clear, and more importantly it is not clear what the performance advantages will be, since on our architecture the topology of the interconnect allows sequential consistency to be implemented at much lower overhead than other architectures.

For cache coherent pages, software can use some of the hardware control described above to improve performance. For example, multicasting data can be used by software to reduce latency, and data can be written back from any cache under software control to reduce the cost of coherence. Similarly, with a write update hardware protocol, processors that are no longer using the data can explicitly invalidate it from their secondary and network caches in order to reduce the overhead of updates.

Cacheable but non-coherent pages can be used to enable software controlled cache coherence. Such techniques can take advantage of application specific semantics to reduce the overhead of coherence for many applications [21]. To make the implementation of these techniques more efficient, NUMAchine maintains state about cache lines (such as which processors have the data cached) that can be directly accessed by the software. We also expect that the support for multicast interrupts provided by our hardware will be useful for some of these techniques.

In-cache zeroing/copying

The operating system must for security reasons zero-fill memory pages when passing them between applications. Similarly, operating systems often have to copy data between different buffers. For both of these operations, the cost of reading the data that is to be over-written can in many cases dominate performance.

NUMAchine minimizes the overhead for zeroing or copying data by allowing these operations to be done without loading the data that will be overwritten into the processor cache. To copy data between a source and target page, the operating system: (1) makes a single request to the affected memory module to invalidate any cached lines of the target page, mark the state as dirty, and set the routing mask (or processor mask) to the processor performing the copy, (2) creates the cache lines of the target page in the secondary cache by modifying the tag and state information of the secondary cache, and (3) copy data between the source and target page. Zero-filling pages is identical to copying pages, except for the final stage, where the created cache lines are instead zero-filled.

3.3 Performance Monitoring Hardware

NUMAchine includes considerable hardware dedicated to monitoring system performance in a non-intrusive fashion. Monitoring hardware is distributed amongst all major sub-systems of the multiprocessor, including the memory system, the processor modules, the ring-interfaces, and the I/O subsystem. For convenience, monitoring hardware is collectively referred to in this section as "the monitor".

The overall uses of the monitor are as follows: 1) Investigate and evaluate architectural features of NUMAchine, 2) provide real-time feedback concerning utilization of system resources, to allow tuning of application programs, 3) accumulate trace history information to aid hardware and software debugging, 4) validate our NUMAchine hardware simulator, and 5) characterize application workloads for high-level performance modelling or network simulations.

A key feature of the monitor is that it is implemented in high-capacity programmable logic devices (PLDs). Because the PLDs being used (Altera MAX7000 Complex PLDs and FLEX8000 FPGAs) are re-programmable, the same monitoring circuits can be re-configured to perform different functions. This offers tremendous flexibility because a wide variety of measurements can be made without incurring excessive cost.

In general, the monitor comprises a number of dedicated hardware counters, flexible SRAM-based counters, and trace memory. The dedicated hardware counters monitor critical resources such as FIFO buffer depths and network utilization. For example, bus and ring-link utilization are important overall performance metrics that can be monitored by dedicated counters. The SRAM-based counters are used to categorize and measure events in a table format. A good example of this is in the memory module, where transactions can be categorized based upon the type of transaction and its originator; a table counting each transaction from each originator would be monitored. This information can help identify resource "hogs", or even program bottlenecks. In addition to counters, trace memory (DRAM) is used to recall history information about bus traffic, for example. This allows non-intrusive probing into activity just before or after an important event such as a hardware error or software barrier.

A novel feature of the monitor is that information gathered can be correlated with execution of specific segments of code, by particular processors. This is implemented by a small register, called a *phase identifier*, at each processor. As executing code enters regions that should be distinguishable for monitoring purposes, the code writes into the phase identifier register; this information is appended to each transaction from the processor and is used by the monitor throughout the system.

In this paper, we discuss in more detail only those monitoring circuits associated with the memory subsystem. The reason for this focus is that memory system performance is a key aspect of shared-memory multiprocessor design, and offers many opportunities for improving performance. A memory module in NUMAchine, as mentioned earlier, consists of an incoming FIFO, DRAM for data, SRAM for state information, and an outgoing FIFO. The monitor measures the way in which the memory is being used by all processors in the system; to accomplish this, it monitors the incoming and outgoing FIFOs, and some of the state information for accessed memory locations. There are two main types of monitoring circuits in the memory module: multipurpose counters, and histogram tables. The purpose of each of these is discussed below.

3.3.1 Multipurpose Counters

Counters in the memory module count the occurrence of events that can provide software with an indication of how memory is being utilized. One usage of such information would be for software to recognize bottlenecks by measuring depths of the memory FIFOs. As another example, if the monitor shows a large number of invalidates at a memory location, then this might indicate that false sharing is occurring. The following are examples of events that can be counted:

- total number of transactions to memory module
- number of transactions of a specific type (e.g., read request, write request, write, etc.)
- number of invalidates sent out of memory module
- depth of incoming and outgoing FIFOs

3.3.2 Histogram Tables

The most interesting and useful monitoring circuits in the memory modules are histogram tables that allow accumulation of statistics concerning memory accesses. The hardware for generating these tables is of a general structure, and can be configured to collect different types of information. For each table, there are two halves: one that is being currently generated, and another that was already generated and has overflowed. The idea behind this is that once any entry in a table overflows, an interrupt is generated so that software can examine the information as desired, but in the meantime monitoring can still continue using the other half of the table. For brevity, we provide only a single example of such a table below, but there are several others that are available.

3.3.3 Cache Coherency Histogram Table

When designing a cache coherence scheme, or evaluating its effectiveness, it is important to know the typical access patterns that can be expected. To some extent, such information can be discovered through simulations, but, for practical reasons, simulated behaviour is always limited in scope. The cache coherency hit table provides a way for monitoring hardware to gather detailed information about the cache coherency state of accessed memory locations. More specifically, the following information is gathered in this table: for each type of memory transaction (e.g., read request, write permission request, etc.), the table accumulates a count of the number of times that each possible cache line state is encountered. In NUMAchine's cache coherence scheme, as outlined in Section 2.3, there are four possible cache line states: local valid, local invalid, global valid, and global invalid. In addition, a cache line can be locked or unlocked in each state. The histogram table would then contain eight rows for the cache line states, and enough columns for all transaction types of interest.

To generate this table, the monitoring hardware needs inputs connected to the data outputs of the incoming FIFO (the bits that specify the type of memory access, and the memory address involved (so that monitoring can be restricted to an address range), as well as the bits in the SRAM that specify the state of the cache line being accessed. In addition, some of the other tables (not described here) that can be generated require other signals. Since the same hardware (PLDs) is reconfigured for each type of table, there are in general some inputs and outputs connected to the monitoring chips that may not be used when generating any particular table.

4 Performance Results

4.1 Simulation Goals

This section describes simulations results of the NUMAchine prototype. There are three main reasons for developing a simulator for NUMAchine: to estimate the performance of the prototype machine, to locate any weaknesses in the NUMAchine architecture and/or the particular parameters of the prototype, and to investigate a number of tradeoffs. In this document, the only simulations shown are those that provide indicators of NUMAchine's performance; however, the simulator is being used on a continuing basis to investigate architectural parameters, to improve the NUMAchine architecture. A complete behavioral simulation of a "virtual" machine has been implemented in software, using state-of-the-art simulation techniques. The simulation environment described in the next section is one step removed from a circuit-level simulation; all behaviour affecting timing and functionality was modelled in detail for each system component that could act independently.

For design verification our primary concerns were the efficiencies of: the rings, the Network Cache, and the cache coherence protocol. For the rings, the obvious question is whether network contention causes serious performance degradation. For the Network Cache, we are interested in its effectiveness at reducing network traffic. Finally, as mentioned in Section 2.3, the coherence protocol was designed optimistically, assuming that certain cases would not happen frequently; thus we wish to determine the actual frequency of these cases, in order to assess whether the protocol will perform as efficiently as hoped.

Beyond the above goals, there are many other questions on enhanced functionality that can easily be asked in a simulation. For example, the benefits of prefetching, broadcasting and weaker consistency models are all of interest. The answers to these questions (and others) are currently under investigation, but for brevity will not be reported here.

4.2 Simulation Environment

The performance of the prototype has been investigated by means of an execution-driven simulation using the SPLASH-2 [3] benchmark suite as input. The simulator itself uses Mint [22] as a front-end to interpret the native MIPS binaries produced for the suite. The back-end does behavioral modelling of the system at a cycle level. This includes all timing details (e.g. bus arbitration, DRAM and SRAM access times) as well as functional details, such as L1 and L2 data and instruction caches, and a packet-by-packet model of the rings. Figure 12 illustrates the NUMAchine simulation environment. A single binary running on either SGI or SUN workstations simulates both the multi-threaded application and the NUMAchine configuration, all of whose details are specified in the text parameter file. Run-time is quite good given the level of detail, with native versus simulated execution slowdown ratios of 100-300 when running on an SGI Challenge machine. Although aspects such as instruction fetching and serial code execution can be modelled in the simulator, they are time consuming and do not significantly affect results.[5] For this reason the results in the rest of this report will assume that only data caches and fetches are implemented, and only the parallel section of the code is modelled in detail. (The serial section of code still executes, but does not generate events.) Results from more detailed simulations will be contained in [4].

Table 1 gives the contention-free latencies for different types of accesses as a yardstick for comparison with results in later sections. For this data, we manually calculate the number of clock cycles required in the hardware to perform the various types of accesses (i.e., these numbers to not reflect such architectural features as caches). The two types of remote accesses represent: requests that traverse only a single lower-level ring, and requests that span the whole network. (Note that due to the single-path nature of a ring, the distance

[5]The SPLASH-2 applications take anywhere from 5 minutes to 24 hours to run on an SGI Challenge for each different system configuration. These times can double if both effects are turned on.

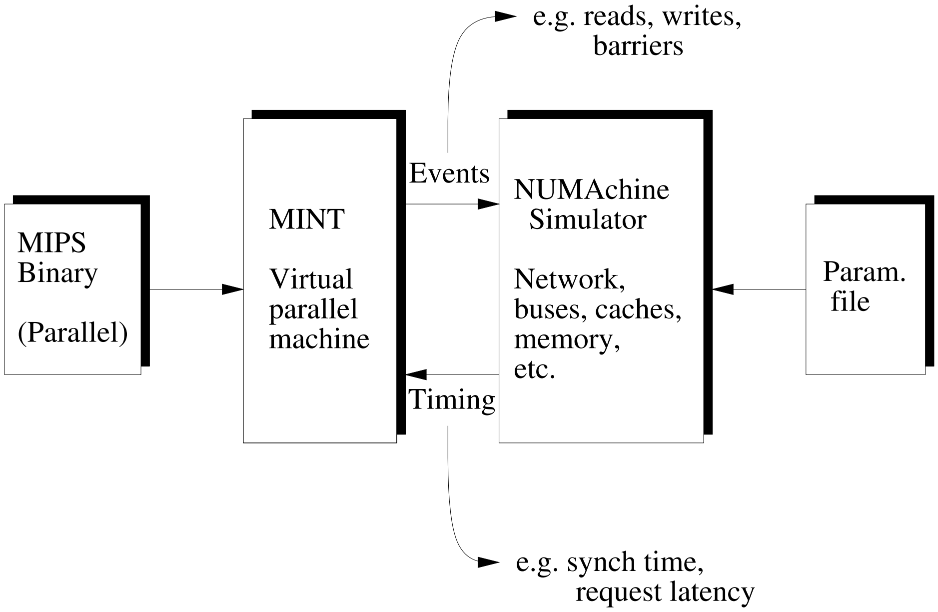


Figure 12: Simulation environment. The parameter file is a text file containing all information on timing and geometry.

Table 1: Contention-free request latencies in the simulated prototype. Reads and interventions involve 64-byte cache line fills. Upgrades contain no data, only permission to write.

Data Access Type	Latency (ns)	Latency (CPU cycles)
Local:		
Read	668	100
Upgrade	284	43
Intervention	717	108
Remote, same ring:		
Read	1652	248
Upgrade	1167	175
Intervention	1656	249
Remote, different ring:		
Read	1908	286
Upgrade	1508	226
Intervention	1932	290

between any two stations that are not on the same ring is equal to the span of the network, regardless of the position of the two stations.) Even without the effect of the Network Cache, these numbers indicate that the prototype behaves as a mildly NUMA architecture.

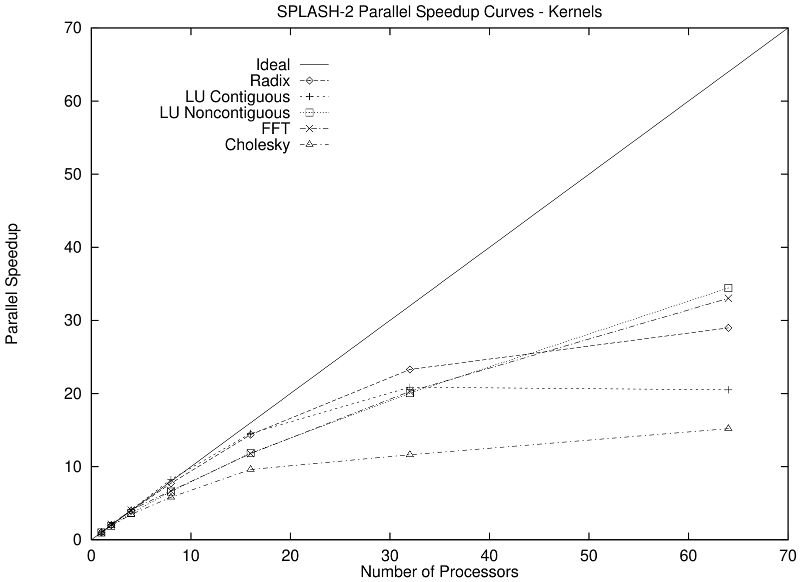


Figure 13: Parallel speedup for SPLASH-2 kernels

4.3 Overall Performance

In order to gauge the overall performance of the NUMAchine prototype, the Splash2 suite was run through the simulator to measure *parallel speedup*; for this data we consider only the *parallel section* of the code, and ignore the *sequential section*. In the Splash2 suite, the parallel section is defined as the time from the creation of the master thread, until the master thread has successfully completed a wait() call for all of its children. This is not a true speedup, but is in line with other performance measurements of this nature (e.g.,see [5] citesplash2.suite). In order to be conservative, all fixed hardware latencies are set to their actual values in the hardware if those values are known, and to pessimistic values otherwise. In addition, the results shown use a simple round-robin page placement scheme which is expected to perform more poorly than if intelligent placement were done. (For example, pages containing data used by only one processor, also called private pages, are not placed in the memory local to that processor, which would be simple to ensure in a real system.) For these reasons, We expect the actual prototype hardware to have equal or better performance than the results shown here indicate.

Figures 13 and 14 show the parallel speedups for the SPLASH-2 benchmarks. All benchmarks are unmodified, except for LU-Contig, which used a slightly modified block-allocation scheme to improve workload balance.[6]. Table 2 gives the problem sizes used for generating the speedup curves.

Highly parallelizable applications such as Barnes and Water show excellent speedups, as high as 57. Of more interest is NUMAchine's performance for code that has a higher degree of data sharing. For FFT and LU, examples of such code, the speedups are still good, especially given the small problem sizes. These results compare favorably with measurements of the SPLASH-2 suite in citeref:splash2.suite using a perfect memory system. This leads us to believe that with suitable tuning of both hardware and software, performance will be on par with the existing state-of-the-art.

[6] For the sake of other SPLASH-2 experimenters, the BlockOwner routine was changed to $(I + J) \bmod P$

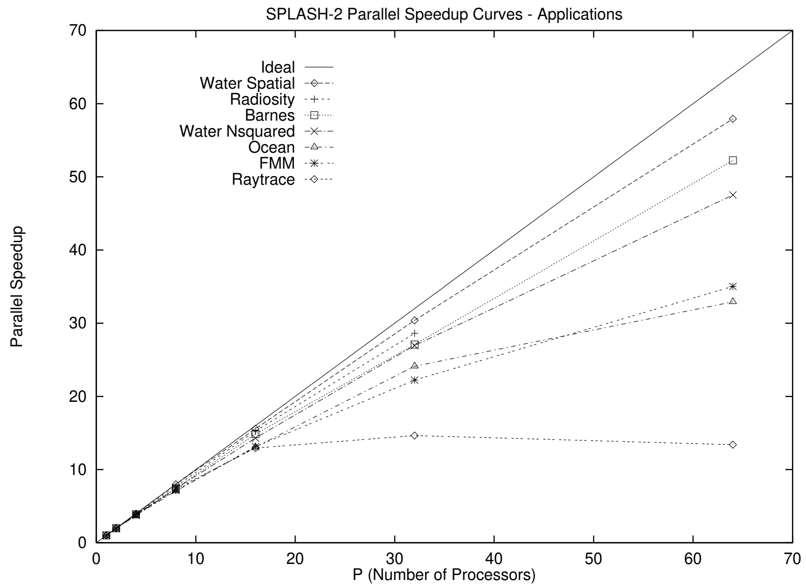


Figure 14: Parallel speedup for SPLASH-2 applications.

Table 2: Problem sizes used for the SPLASH-2 benchmarks.

Benchmark	Problem Size/Input
LU Contiguous & Non-contiguous	512x512 Matrix, 16x16 Blocks
FFT	65536 Complex Doubles (M=16)
Radix Sort	262144 Keys, Radix 1024
Cholesky	tk18.O Input File
Water Spatial & Nsquared	512 Molecules, 3 Steps
Ocean	258x258 Grid
Barnes	16384 Particles
FMM	16384 Particles
Raytrace	Teapot Geometry
Radiosity	Room in batch mode

4.4 Performance of Communication Paths

The efficiency of NUMAchine's interconnection network can be shown using a number of performance metrics. Figure 17 depicts the utilization of the station buses, local rings and central ring. It indicates that none of these components is likely to become a performance bottleneck. Figure 8 shows the delays in ring interfaces. Each vertical bar shows two components of the delay. The lower portion of each bar corresponds to the minimum delay (in the absence of network traffic) and the upper portion indicates additional delay due to traffic contention. The average packet delays in the upward and downward paths in the local ring interfaces are shown in Figure 4.5. The upward path delay is small for all applications. The larger delays for the downward paths are due to the way in which we have implemented the packet handler and the queues, which we are currently redesigning to reduce these delays. Packet delays from the central ring to a local ring

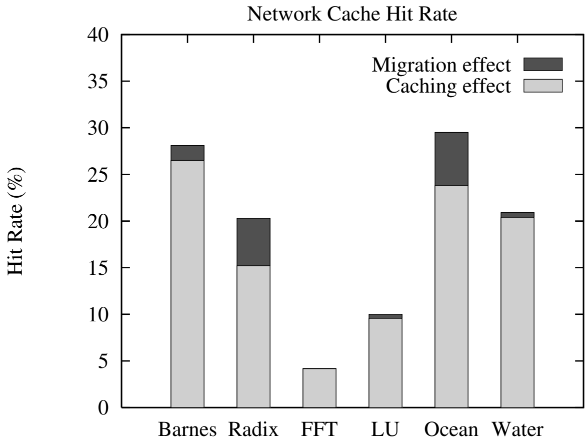


Figure 15: Network cache total hit rate.

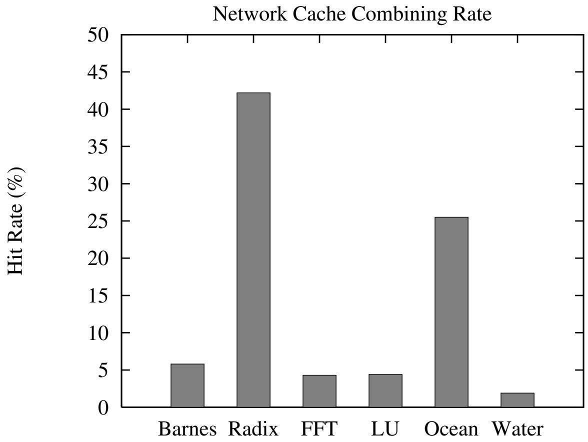


Figure 16: Network cache combining rate

have the same small delays as for the upward path in Figure 4.5. The average packet delays from a local ring to the central ring are only slightly larger, as shown in Figure 4.5. This indicates that for our targeted system size the rings are expected to perform well.

4.5 Performance of the Network Cache

The simplest measure of Network Cache performance is the hit rate, defined as,

$$\text{hit rate} = \frac{\text{number of requests satisfied locally}}{\text{total number of requests, not counting retries}},$$

and shown in Figure 15. (Note that local interventions are counted in the numerator.) Retries are generated locally when a cache line is locked in the NC due to a pending remote request. This locking could be due to a request to the same or a different cache line (cache conflict) from another processor. However, given

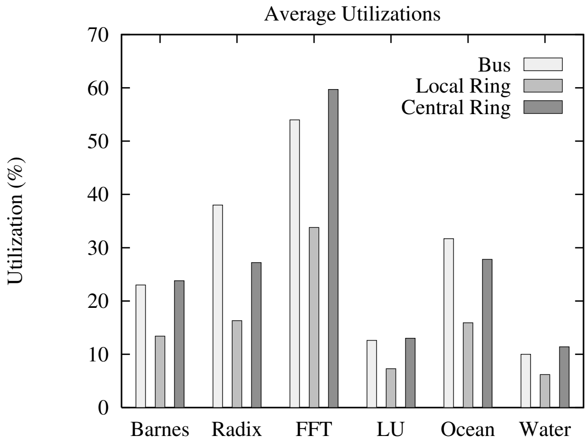
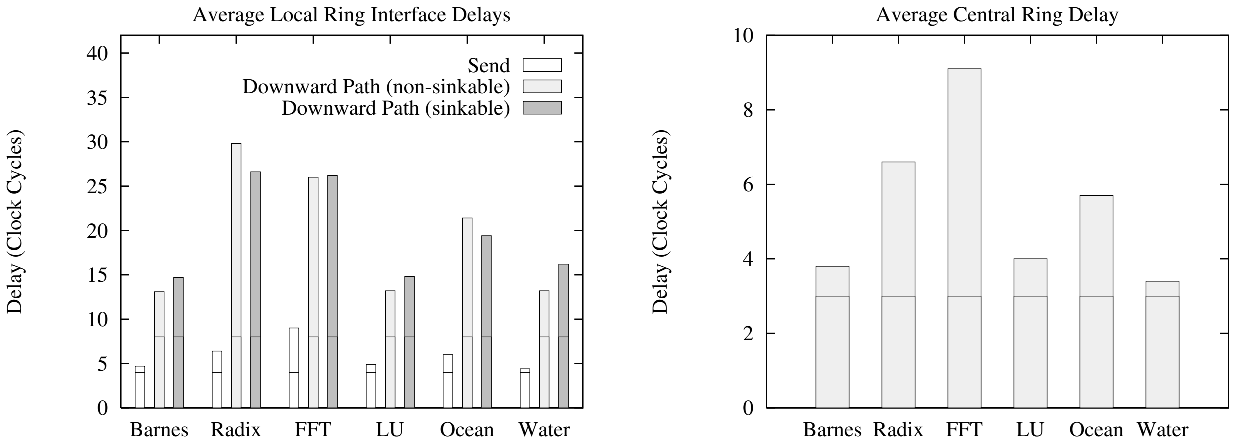


Figure 17: Average utilization of communication paths.



(a) Delays in the local ring interface.

(b) Delay in the upward path of the central ring interface.

Figure 18: Local and central ring interface delays.

the large size of the NC and the fact that there can only be 4 outstanding requests at one time (because each R4400 processor can generate only one request at a time), the chances of such a conflict are slim. Most retries are due to concurrent requests to the same cache line. When the pending request returns through the network and unlocks the line, the next retry will succeed. (Assuming the line is not ejected in the interim, which is unlikely.) This masking-out of simultaneous requests is termed the 'combining' effect, since multiple requests result in only a single network access. This effect is displayed in Figure 16.

Another reduction in network traffic is gained from what is termed the 'migration' effect. In essence, when data brought onto a station is then accessed by another processor, a remote access is potentially saved. This is true both for data that is dirty, as well as data that is shared. It is worthy of note that a system utilizing bus snooping would also see this benefit, but only for dirty data.

Table 3: Percentage of local requests to NC that result in a false remote request being sent to memory.

Application	Percentage of false remotes
Cholesky	< 0.5
FMM	< 1.0
Ocean	< 0.3
Radiosity	< 0.2
Radix	< 0.5
All others	<< 0.01

4.6 Performance of the Coherence Protocol

As mentioned at the beginning of this section, the coherence protocol was designed under the assumption that certain cases would occur only infrequently. To assess the validity of that assumption, we measure the frequency of those cases here.

The first case involves the inexactness of the filter mask. It is possible that an "old" write permission request that has been travelling through the network for some time can reach memory *after* previous requests have invalidated the requester's shared copy. Since the filter mask is not precise, it is possible for the memory module to erroneously believe that the requester still has a shared copy, in which case it will respond with only an acknowledgement, granting ownership to the requester. The requester will see the ownership grant, but will not have valid data. In this case, the requester must send a *special write request* to memory, indicating that data must be returned. The above scheme is an *optimistic* design, in that a memory module always assumes that a requester has correct data, in spite of the ambiguous directory information. The alternative would be for the memory module to assume that the requester does not have valid data when such ambiguity arises; this implies that data would always be sent, and this would be wasteful unless that data is almost always needed. The simulation results shown below indicate that the optimistic choice is the right one. Across all the applications and for all system geometries (representing hundreds of millions of requests to memory) only 4 special read requests were ever sent. This result is a manifestation of the well-known property of multiprocessor systems that a given cache line is almost always shared by 1, 2 or all processors, and very rarely by some number in between; the chances that three stations share a line in just the right way for the optimistic assumption to fail are small.

The second case of interest arises due to the direct-mapped nature of the network caches. It is possible for the network cache to lose directory information due to replacements by other requests. The most costly effect of this choice is when data has been made dirty locally on a station, but this information is subsequently thrown out of the NC. A request for this line now misses, and is sent to memory, which sends the request back indicating that its filter masks indicates that the local station already has that data, in LV state. At this point the NC does the intervention that it could have done immediately if the directory information had not been lost. We call these types of misses *false remote requests*. Again the simulations show that this case happens very infrequently. Table 3 indicates the percentage of all local requests that end up generating false remote requests. Only for one application, FMM, does the percentage approach 1 %.

Both of the above cases arise due to a loss of information in the coherence protocol. (In one case it is imprecision in the directory bits, in the other it is the wholesale loss of all local directory information.) The conclusion is that full state/directory information is not necessary for the efficiency of the cache coherence protocol. The cases for which the protocol chose simplicity over efficiency are those that happen rarely enough that overall performance is not affected.

5 Related Work

Over the past few years, a number of scalable multiprocessors that support a single coherent view of memory have been designed and/or built. In this section, some of the features of recent machines are considered, in order to show how NUMAchine compares to other approaches.

The Stanford DASH multiprocessor [15] uses clusters of processors that share a single bus, with clusters interconnected by a mesh. It uses a directory-based hardware cache coherence protocol that, on a write to a shared cache line, requires separate invalidates be sent for each of the copies, and requires acknowledgments for each invalidate. In the NUMAchine protocol, only a single invalidate message is used, and no acknowledgements are required. DASH employs a small cache in each cluster called a Remote Access Cache (RAC). NUMAchine's network cache includes the functionality of the RAC; however, the key to the effectiveness of NUMAchine's network cache is its large capacity, being at least as large as the combined capacities of the secondary caches on a station.

The FLASH multiprocessor [13], under development at Stanford University, will provide a single address space with integrated message passing support. A programmable co-processor, called MAGIC, serves as a memory and I/O controller, a network interface, and as a communication and coherence protocol processor. Through this programmable co-processor, FLASH provides a high degree of flexibility. NUMAchine uses a different approach to providing flexibility. The basic protocols, such as coherence, are implemented in hardware to ensure good performance, but software has the ability to override the hardware when different protocols are desirable.

The Alewife machine from MIT [1] shares the FLASH approach of integrating a single address space with message passing. Its approach for achieving flexibility is to implement common case critical path operations in hardware, letting software handle exceptional or unusual conditions. For example, it uses limited directories [8] to implement cache coherence, where hardware supports directly a small number of pointers, and software must handle the case when cache lines are shared by a larger number of processors. An important difference between Alewife and NUMAchine is that Alewife relies on a great deal of custom hardware. As a result, it is harder for Alewife to track the rapid improvements in workstation technology.

The KSR multiprocessors [12] from Kendall Square Research use a ring-based hierarchical network to interconnect up to 1088 processing cells. These systems implement a Cache Only Memory Architecture (COMA), which automatically replicates data to requesting cells. Although NUMAchine uses a similar interconnection topology, there are a number of fundamental differences between the two networks. In the KSR systems, each processing cell must snoop on ring traffic to maintain cache coherence. This effectively involves a directory lookup and slows the speed of operation. Furthermore, a combined cache directory is needed at each level in the interconnect hierarchy, containing all the directory information in the levels below, which severely limits the scalability of the architecture. The replication of data in the COMA memory is effective in reducing memory and network contention [6]. NUMAchine captures most of these benefits with its network caches, but without affecting scalability and at a considerably reduced cost.

Other interesting multiprocessor projects include the ASURA [17] multiprocessor being developed at Kyoto University in Japan, Typhoon [20] from the University of Wisconsin, the Cray T3D system [18] from Cray Research, and the Exemplar from Convex [9]. ASURA has many similarities with NUMAchine, but its equivalent of the network cache uses very long cache line sizes (1 Kbyte), which may lead to considerable false sharing. Typhoon has similar flexibility goals to FLASH, and also depends on a programmable co-processor to implement its coherence policy. The T3D does not support cache coherence in hardware. The Exemplar uses a crossbar to interconnect processors in a cluster and uses SCI rings to interconnect clusters and maintain inter-cluster coherence. The distributed directory-based protocol implemented by SCI, using linked lists, can introduce considerable cache coherence latency overhead.

6 Concluding remarks

In order to be successful, future multiprocessor systems must be cost effective, modular, and easy to program for efficient parallel execution. The NUMAchine project seeks to address these issues by developing a cost-effective high-performance hardware platform supported by software to ease the task of developing parallel applications and maximizing parallel performance. In this report we have provided an overview of the NUMAchine hardware architecture and presented simulation results to demonstrate some of the implications of the architecture on performance.

The NUMAchine ring hierarchy gives the desired simplicity of implementation. Since there are only three connections to each node, it is possible to use wide datapaths. We have developed a simple routing mechanism that allows the rings to be clocked at high speed. An shown in the evaluation section, the bisectional bandwidth of our network is sufficient for typical applications running on the target system size. In addition, the high-speed operation results in low latency for remote accesses.

The hierarchical nature of the NUMAchine rings allows for a natural implementation of multicasts. This feature is exploited by the coherence mechanism to invalidate multiple cache lines using a single packet. It is also exploited to implement an efficient multicast interrupt mechanism and to implement, in hardware, support for efficient barrier synchronization.

The cache coherence support in NUMAchine is highly optimized for applications where most sharing is localized within a single station, in which case coherence is controlled by the local memory or network cache and no remote interactions are required. A two-level directory structure is used, where the number of bits per cache-line grows only logarithmically with the number of processors in the system.

In addition to localizing coherence traffic, the network cache serves as a larger shared tertiary cache for the processors on the station. It is implemented in DRAM, which will allow us to experiment with very large cache sizes in order to avoid remote accesses. Also, the network cache serves as a target for such operations as multicast writes; system software can cause cache lines to be multicast to a set of stations where it is expected that the data will soon be required.

The NUMAchine architecture is one component of the larger NUMAchine project, which involves development of a new operating system, parallelizing compilers, a number of tools for aiding in correctness and parallel performance debugging, and a large set of applications. For this reason, our prototype will include extensive monitoring support. Also, it will allow system software to take control of the low-level features of the hardware, facilitating experimentation into hardware-software interaction.

References

[1] A. Agarwal, D. Chaiken, G. D'Souza, et al. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS Memo TM-454, Laboratory for Computer Science, Massachusetts Institute of Technology, 1991.

[2] R. Balan and K. Gollhardt. A scalable implementation of virtual memory HAT layer for shared memory multiprocessor machines. In *Summer '92 USENIX*, pages 107–115, San Antonio, TX, June 1992.

[3] T. be filled in later. To be filled in later. *To be Added Later*, 1995.

[4] T. be filled in later. To be filled in later. *To be filled in later*, 1995.

[5] T. be filled in later. To be filled in later. *To be filled in later*, 1995.

[6] R. Bianchini, M. E. Crovella, L. Kontoothanassis, and T. J. LeBlanc. Memory contention in scalable cache-coherent multiprocessors. Technical Report 448, Computer Science Department, University of Rochester, 1993.

[7] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple but effective techniques for NUMA memory management. In *Proc. of the 12th ACM Symp. on Operating System Principles*, pages 19–31, 1989.

[8] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proc. of the Fourth Int'l Conf. on ASPLOS*, pages 224–234, New York, April 1991.

[9] Convex Computer Corporation. *Convex Exemplar Systems Overview*, 1994.

[10] K. Farkas, Z. Vranesic, and M. Stumm. Cache consistency in hierarchical-ring-based multiprocessors. Tech. Rep. CSRI-273, Computer Systems Research Institute, Univ. of Toronto, Ontario, Canada, January 1993.

[11] K. Farkas, Z. Vranesic, and M. Stumm. Scalable cache consistency for hierarchically-structured multiprocessors. *Journal of Supercomputing*, 1995. in press.

[12] Kendall Square Research. *KSR1 Technical Summary*, 1992.

[13] J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH multiprocessor. In *Proc. of the 21st Annual ISCA*, pages 302–313, Chicago, Illinois, April 1994.

[14] R. P. LaRowe Jr. and C. S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, Nov. 1991.

[15] D. Lenoski, J. Laudon, K. Gharachorloo, et al. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, March 1992.

[16] D. E. Lenoski. The design and analysis of DASH: A scalable directory-based multiprocessor. Technical Report CSL-TR-92-507, Stanford University, January 1992.

[17] S. Mori, H. Saito, M. Goshima, et al. A distributed shared memory multiprocessor: ASURA — memory and cache architectures —. In *Supercomputing '93*, pages 740–749, Portland, Oregon, November 1993.

[18] W. Oed. The Cray Research massively parallel processor system CRAY T3D. Technical report, Cray Research GmbH, München, Germany, Nov. 15 1993.

[19] S. K. Reinhardt, B. Falsafi, and D. A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Usenix Symposium on Microkernels and Other Kernel Architectures*, pages 73–89, September 1993.

[20] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21st Annual ISCA*, pages 325–336, Chicago, Illinois, April 1994.

[21] H. S. Sandhu, B. Gamsa, and S. Zhou. The shared regions approach to software cache coherence on multiprocessors. In *Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, May 1993.

[22] J. E. Veenstra. Mint Tutorial and User Manual. Technical Report 452, Computer Science Department, University of Rochester, May 1993.