RC 21318 (94478) 20OCT98 **Computer Science/Mathematics**

Research Report

Hardware and Operating System Design for a Cluster-Based Scalable Shared Memory System

Kattamuri Ekanadham, Mark Giampapa, Joefon Jann,
Beng-Hong Lim, Pratap Pattnaik, and Marc Snir
IBM Research Division
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Alan Benner, Dean Liberty, David Sadler, and Gautam Shah
IBM RS/6000 Division, Poughkeepsie, NY 12601

Research Division
Almaden T.J. Watson Tokyo Zurich Austin

Hardware and Operating System Design for a Cluster-Based Scalable Shared Memory System[1]

Kattamuri Ekanadham, Mark Giampapa, Joefon Jann,
Beng-Hong Lim, Pratap Pattnaik, and Marc Snir
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

Alan Benner, Dean Liberty, David Sadler, and Gautam Shah
IBM RS/6000 Division, Poughkeepsie, NY12601

October 20, 1998

Abstract

This paper describes PRISM, a hardware cache coherent distributed shared memory (DSM) architecture based on a cluster of SMP nodes, where each node runs an independent operating system kernel. The architecture relies on tight coordination between the hardware and OS to solve the problem of scalability and fault containment in such systems. For scalability, the hardware and OS interactions are highly localized to a node, even for potentially global operations such as page faults and page migration. Inherently global operations, such as the mapping of global objects, amortize their cost over a large number of pages. For fault containment, the system avoids using physical addresses as global names to prevent wild writes on a malfunctioning node from affecting other nodes. For performance, the system allows efficient and flexible software control over memory and caching behavior (e.g., CC-NUMA vs Simple-COMA) of each page of memory. Moreover, the control is performed independently at each node.

All of these features are provided within a single unified system design. We are currently implementing the system and find its complexity to be equivalent to that of existing hardware DSM systems. We discuss the features of SMP nodes that are vital for building efficient DSM systems out of SMP clusters. We also discuss the features of the cluster interconnect that are necessary for performance and avoiding deadlocks. We present simulation results that demonstrate performance benefits of the architecture over previous designs.

Index Terms: distributed shared memory, cache coherence, memory management, symmetric multiprocessors, SMP clusters, parallel processing.

1.Contact Author: Joefon Jann (joefon@watson.ibm.com)

1 Introduction

Scalability is a prime consideration in the design of high performance computer systems. It allows a system to grow to a size and performance level that best fits users' needs and satisfy a wider spectrum of users. Scalability, together with the benefit of leveraging the cost-performance of commodity components, motivates a system design that is composed of a cluster of symmetric multiprocessor (SMP) nodes, with a minimal amount of specialized hardware to accelerate communication between the nodes.

The commercially successful IBM RS/6000 SP scalable system takes this approach by clustering commodity RS/6000 workstation-class nodes with a custom designed, high speed network. Each node runs a separate instance of the OS, but the system is unified by a single system management framework and parallel operating environment. The primary method of programming multi-node parallel applications on the SP is via explicit communication, using standard communication APIs such as sockets, PVM, or MPI. The advantages of such API's are their scalability as well as their portability.

However, the need for shared memory programming has not vanished. Shared memory provides an alternative paradigm that is more natural for programming certain classes of programs, and different parts of a parallel program do not need to explicitly send and receive data in order to share it. This can simplify the development of parallel codes, especially for applications with irregular data access and communication patterns. Compiler development can also be eased by shared memory.

The use of SMP nodes provides a further motivation. Unless shared memory is extended across SMP nodes, programmers will either have to restrict the use of shared memory to within an SMP node and use message passing between nodes, or use message passing exclusively, even within an SMP node. A more appealing alternative is to program using globally shared memory, and incrementally modify the program to incorporate message passing to improve performance whereever necessary.

These considerations motivate the need for shared memory support in a scalable, parallel computer system. This paper describes PRISM (*P*artitionable, *R*econfigurable, *I*ntegrated *S*hared *M*emory), a distributed shared memory (DSM) architecture that maps naturally onto a cluster of SMP nodes. PRISM's hardware support is a shared memory adapter that attaches to the memory bus of an SMP. The adapter provides mechanisms for flexible management and dynamic reconfiguration of shared memory pages. PRISM's operating system is structured as multiple independent OS images, one on each SMP node. The architecture relies on tight integration between the shared memory hardware and the OS on each node for scalability. By minimizing inter-node OS coordination and shifting some memory management to intra-node coordination between the OS and the hardware, we minimize the amount of global communication necessary for managing shared memory. Because each node's physical resources are managed by an independent OS image, and shared memory physical addresses are private to each node, the system is more easily partitioned and fault contained in the event of node failures.

PRISM's shared memory adapter architecture provides a facility to take different coherence actions on a per page basis. Each SMP node runs an instance of a full operating system like Unix or Windows-NT, with minor modifications for coordinating page translations with the local shared memory adapter. The operating system can dynamically and independently choose the behavior of each local page. As a concrete example, PRISM allows dynamic configuration of a shared page independently in either CC-NUMA or S-COMA styles, providing the advantages of both styles. In addition to flexible memory management, PRISM provides the following features that enhance performance and fault containment:

- Localized memory allocation and translation. Each node's kernel manages a completely node-private translation between virtual and physical addresses. This allows a standard virtual memory manager to run on each node and reduces the need for global TLB invalidations.
- Lazy home migration. The home node of a page contains the directory information for the cache lines within that page. Lazy home migration allows home nodes to be migrated without invalidating address translations. Non-home nodes update their knowledge of the home node's location only when they access the page. This provides efficient page migration.
- Optional mapping of remote shared memory pages at client nodes into local memory. S-COMA behavior is provided when a page maps to a page frame[1] in local memory. CC-NUMA behavior is provided when a page maps to a page frame in memory at a remote node.
- User controlled granularity of virtual to global address binding. This allows the cost of global coordination due to global address binding to be amortized over large, user defined regions instead of small, fixed-size pages. Once bound, address translations are maintained locally.

1.We use the term *page* to refer to a virtual or global page, and the term *page frame* or *frame* to refer to a physical page.

Results from simulating PRISM on a set of eight SPLASH applications show a significant performance difference with pages mapped in CC-NUMA or S-COMA styles. Most of the difference can be attributed to capacity effects (due to finite cache sizes or limited associativity) in both the processor cache and the S-COMA memory-based page cache. The results also show that simple adaptive run-time algorithms for choosing page types allows PRISM to achieve the best possible performance.

The rest of this paper is organized as follows. Section 2 describes distributed shared memory systems and contrasts the different cache coherence and memory management styles of those systems. Section 3 descrbies PRISM's hardware design and implementation. Section 4 describes the OS structure for managing shared memory and demonstrates how the system provides efficient page replication and migration. Section 5 summarizes features that we deem necessary for building a scalable shared memory system out of commodity components. Section 6 presents an experimental evaluation of PRISM's architecture that demonstrates the advantages of its flexible management and configuration of shared memory pages. Finally, Section 7 concludes the paper.

2 Scalable Shared Memory Architectures

This section compares existing approaches to providing hardware based, cache coherent distributed shared memory. It surveys serveral existing DSM systems, comparing their characteristics with PRISM's. We assume a system structure that is based on a shared memory adapter (labeled A) that interfaces to the memory bus of an SMP node, as illustrated in Figure 1. The adapter's responsibility is to snoop on memory bus for coherence traffic and extend the cache coherence across multiple nodes.

Figure 2 illustrates the generic high level architecture of a shared memory adapter. Physical addresses that appear on the memory bus either directly address local memory, or are handled by the adapter. Glo-

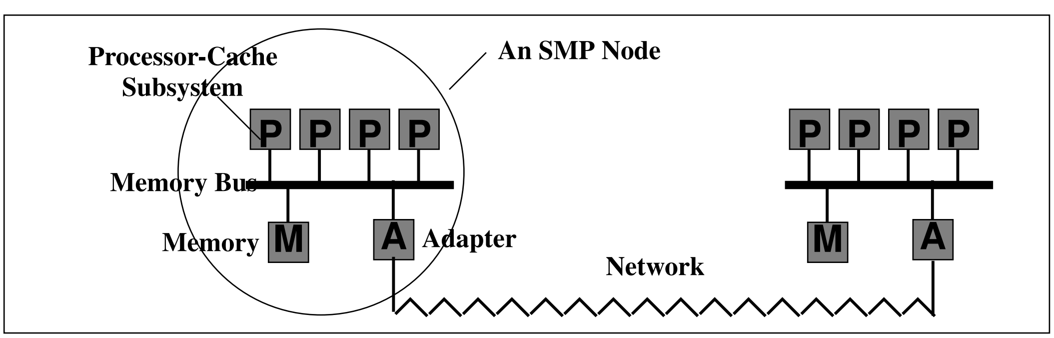


Figure 1 Illustration of a distributed shared memory system composed of SMP nodes interconnected by a network.

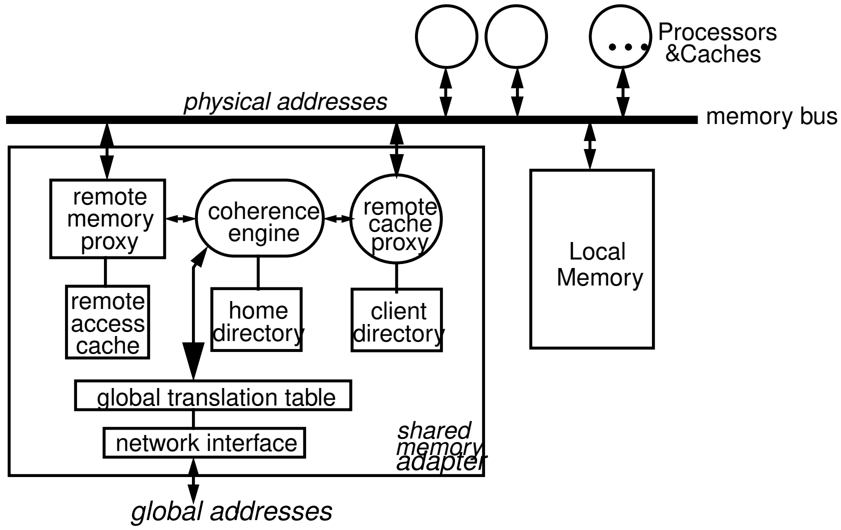


Figure 2 Generic architecture of a distributed shared memory adapter.

bal addresses flow between nodes and are used to name shared data, and a *global translation table* translates between physical and global addresses at each node. The translation may be the identity function, if physical and global addresses are the same. Each global address has a *home node* that maintains the coherence of the cache line at that address. All other nodes are referred to as *client nodes* of that line.

The *remote memory proxy* acts as a memory device or bus slave that handles cache misses to physical addresses that refer directly or indirectly to memory at remote home nodes. If the data is not available from the *remote access cache*, it redirects the cache miss to the adapter at the home node of the cache line. The home node's adapter maintains global coherence of its cache lines and keeps the coherence state and the list of sharers in the *home directory.* Data returned to the remote memory proxy will be cached by the requesting processor, and also by the remote access cache, if available.

The *remote cache proxy* acts as a processor device or bus master and filters accesses to physical addresses that refer to the local memory. It maintains the state of the lines in local memory in the *client directory.* The client directory state may be Invalid (a valid copy is at another node), Shared (local and zero or more remote nodes have valid copies), Exclusive (no remote node has a valid copy) or Transit (there is an ongoing coherence transaction for the line). If the copy in local memory is invalid, or if it is shared and the access is a write, then the remote cache proxy preempts the access to local memory and redirects it to the home node, where it is processed by the coherence engine. Returned data will be cached by the requesting processor and stored in local memory.

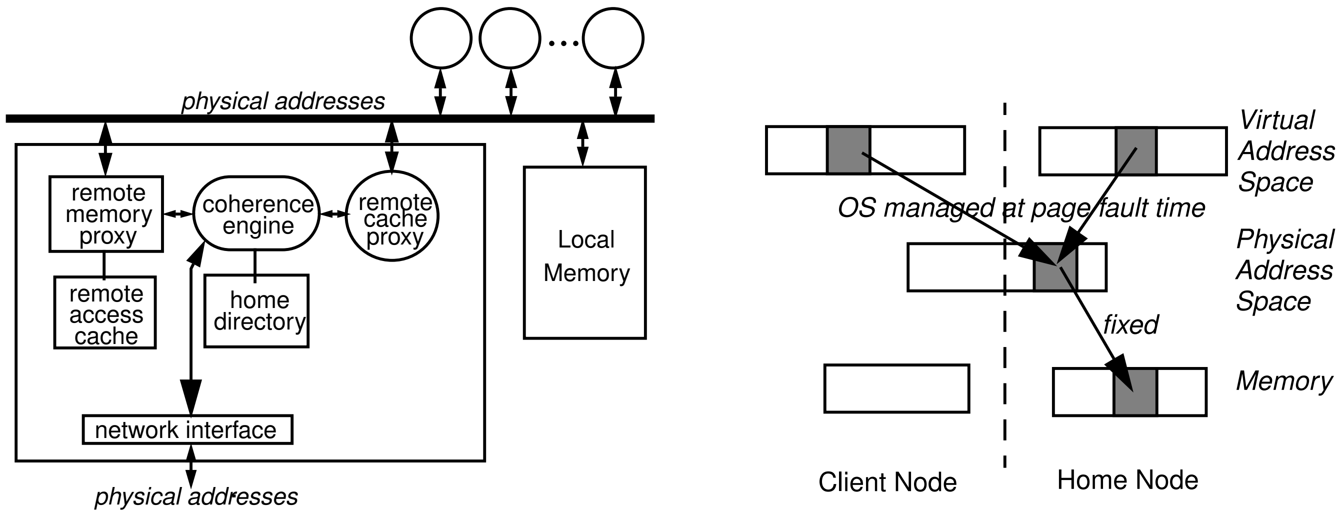


Figure 3 Shared memory adapter architecture and memory mappings in a CC-NUMA system.

We may differentiate DSM systems by what they provide in the shared memory adapter, and by how they manage the shared memory address space. While previous research has focused on cache coherence protocols to optimize for different hardware and application requirements, we focus instead on memory management, *i.e.*, how the hardware and operating system cooperate in allocating, mapping and configuring shared memory in a scalable fashion. In particular, DSM system designs have the following choices:

- To use physical addresses as global names, or to provide a separate global address space.
- To manage local memory as a cache.
- To manage the system with a single or multiple OS images.

These choices lead to different adapter architectures and memory management structures, and have a significant effect on the performance, fault containment and scalability of the shared memory system. We review the differences between the major DSM organizations within this framework.

2.1 CC-NUMA

CC-NUMA systems provide a global physical address space for naming shared data. Figure 3 illustrates CC-NUMA adapter architecture and memory mappings. There is no client directory since local memory is not managed as a cache. However, there may be a remote access cache to serve as a large, albeit slow L3 cache. There is no global page table since physical addresses also serve as global addresses. Virtual addresses must map to the same physical address in order to refer to shared data.

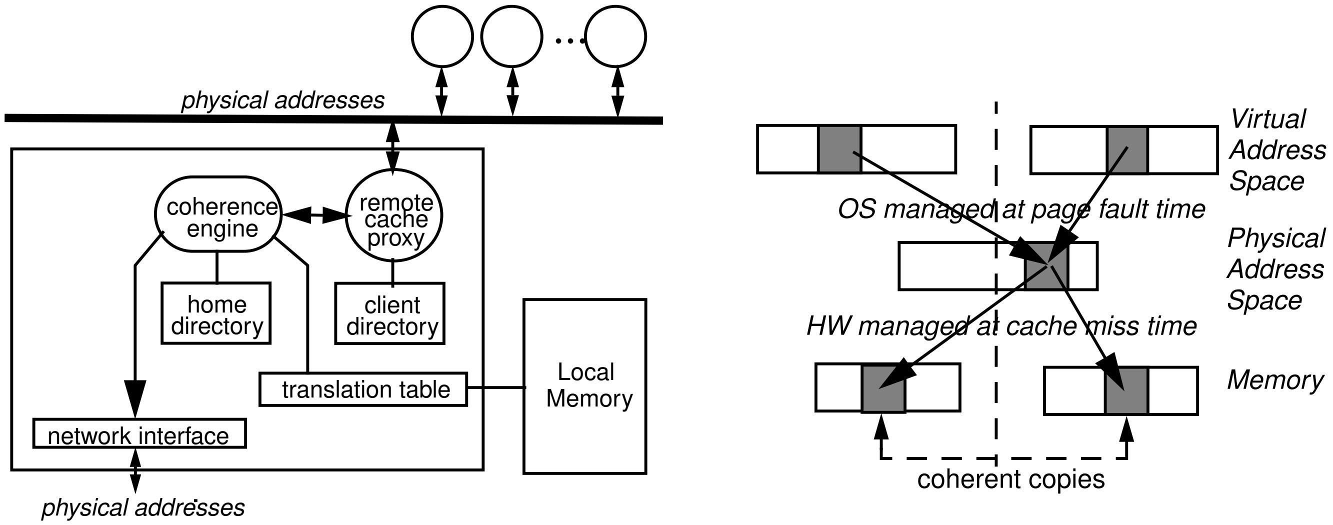


Figure 4 Shared memory adapter architecture and memory mappings in a COMA system.

In principle, a global physical address space provides a clean separation between the hardware and the OS. One can run a standard single-image SMP operating system on a CC-NUMA system. However, scalability and fault containment problems limit the system size. Further, maintaining a globally consistent mapping of virtual to physical addresses involves expensive global coordination whenever mappings change on any node. Also, allowing physical addresses to directly address memory in any node makes the system a monolithic failure unit. Thus, there have been efforts to restructure SMP operating systems to scale better on CC-NUMA machines [5][20].

2.2 COMA

Figure 4 illustrates the adapter architecture and memory mappings in a COMA system. COMA systems also provide a global physical address space for naming shared data and run a single-image operating system. However, they require specialized *attraction memory* hardware to manage local memory as a cache [8][7]. For each cache line accessed by a processor, the attraction memory hardware allocates space in the *local* memory for storing the cache line. The hardware also translates processor generated physical addresses to the local memory locations that hold the cache lines. Since all memory is managed as a cache, the shared memory adapter does not need to have a remote memory proxy. The management of memory as a cache is completely shielded from the OS.

This approach suffers from several drawbacks. The structure of the memory subsystem is quite different from a conventional memory subsystem and requires substantial complexity to handle dynamic mem-

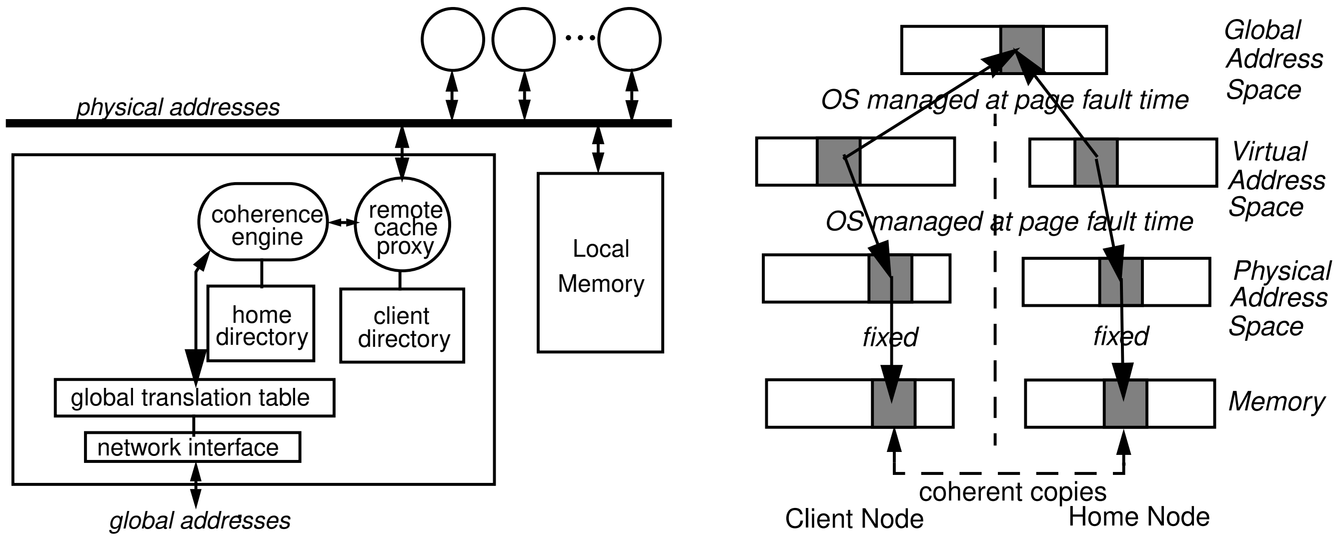


Figure 5 Shared memory adapter architecture and memory mappings in an S-COMA system.

ory allocation in hardware. Further, its cache coherence protocol is significantly more complex due to the need to ensure that the last copy of a cache line does not get discarded. All this adds to the system cost. There is also a performance penalty as all accesses to memory may be slowed by the need to involve the attraction memory, even for accesses to code or private variables. Finally, the problems of OS scalability and fault tolerance are still present.

2.3 Simple-COMA

Simple-COMA (S-COMA) systems provide multiple, node-private physical address spaces, and a separate global address space for naming shared data [18][17][16]. Figure 5 illustrates S-COMA adapter architecture and memory mappings. To share data, virtual addresses must map to the same global address, although they may map to different physical addresses at each node. Like COMA, S-COMA manages local memory as a cache for data referenced by local processors. The client directory maintains the state of these in-memory cache lines. Since memory is managed as a cache, there is no need for a remote memory proxy. Unlike COMA, however, local memory allocation and home node assignment is performed in software by the OS. This significantly simplifies the hardware design over COMA, and memory accesses to lines that are valid in local memory proceed without any additional latency.

The OS binds virtual pages to global pages as page faults occur [17]. A page frame in local memory *must* be allocated before shared data can be accessed. At that time, the OS also informs the shared memory adapter of the mapping between the global page and the newly allocated page frame. The adapter

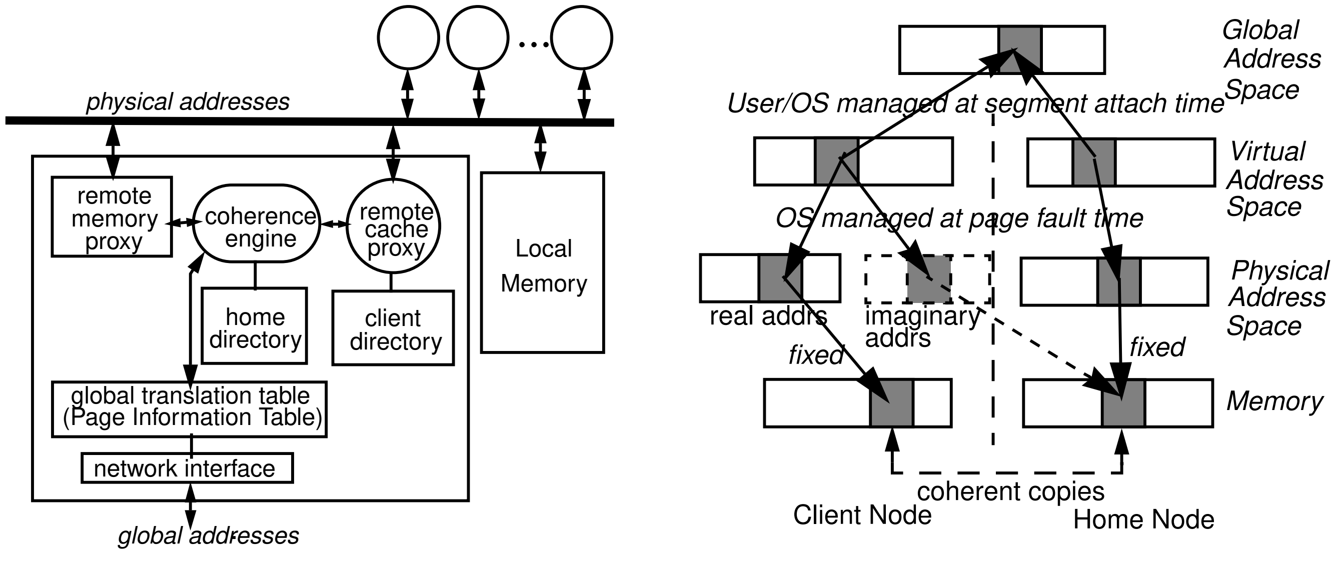


Figure 6 Shared memory adapter architecture and memory mappings in PRISM.

stores this mapping in the global page table. Since S-COMA provides node-private physical address spaces, it is natural to run an S-COMA system using independent OS images on each node. Virtual to physical address translations may be managed locally, although the OS images still need to coordinate for maintaining virtual to global address translations. One may view S-COMA as a DSVM system [11] where coherence is handled by hardware at cache line granularities instead of in software.

The main disadvantages of S-COMA over CC-NUMA are: i) the extra hardware for the client directory and for the global translation table, as well as the extra complexity to manage them, and ii) the need for local memory to be allocated for shared data, which leads to higher memory consumption.

2.4 PRISM

Figure 6 illustrates PRISM's adapter architecture and memory mappings. PRISM's system structure is most similar to that of S-COMA. It provides multiple, node-private physical address spaces, and a separate global address space for naming shared data, and can manage local memory as a cache. Each node runs a separate OS image.

Unlike S-COMA, however, PRISM has a choice of allocating local memory or not for shared data at a client node. It achieves this by providing the novel concept of *imaginary addresses*; node-private physical addresses that do not address any local memory. Accesses to imaginary addresses are handled by the remote memory proxy, which translates them to global addresses before forwarding them to the home node. At page fault time, the OS chooses to allocate a page frame in either real or imaginary physical address space. This allows PRISM to avoid the problem of high memory consumption in S-COMA systems. Imaginary frames provide CC-NUMA style memory access without the need to use a physical address as a global name. We call these imaginary frames *LA-NUMA* for Locally-Addressed NUMA.

Also unlike S-COMA, the mapping between virtual and global addresses is performed when the user maps or attaches part of the virtual address space to a global segment. This mapping may be done at an arbitrary size, allowing the cost of coordinating global translations to be amortized over large user defined segments instead of fixed-size pages. Finally, home node locations are represented explicitly in a global translation table, termed a *Page Information Table* (PIT) instead of being encoded within an address. This feature is key to providing efficient page migration, as we will see in Section 4.5. Sections 3 and 4 describe PRISM's hardware and OS in more detail.

2.5 Examples of Existing Commercially Available DSM Systems

Some of the commercially available scalable shared memory systems include the HP/Convex Exemplar [2], the Sequent NUMA-Q [12] and the SGI Origin 2000 [10]. Each of these is a CC-NUMA system with a distributed memory, global physical address space and a single operating system image. In each of these systems any processor can access any memory location in the system.

The HP/Convex Exemplar uses SCI based protocols to maintain coherence across nodes connected via a toroidal network. Each node has up to 16 PA-8000 processors connected through a crossbar to memory and I/O. The OS allocates a configurable portion of local memory to serve as a remote access cache[15]. This cache is managed by the interconnect controller/shared memory adapter. A large cache could simulate COMA behavior. Although the architecture allows up to 32 nodes, the OS supports coherent shared memory only across 4 nodes (in 1996). Larger systems can be constructed using standard interconnects such as FDDI, HiPPI and Fibre Channel, though such systems lack shared memory between them. The Exemplar OS, SPP-UX, is composed of distributed microkernels. At boot time, the system can paritition some of the memory on a node to be local and prevent it from being accessed by other nodes. Although SPP-UX runs standard HP-UX codes , it provides extensions to allow applications to utilize the various memory classes (e.g., local, global, thread specific) available on the system.

The current implementation of the Sequent NUMA-Q [19] scales up to 64 nodes interconnected in a ring topology using SCI based point to point links. Each node is a Pentium P6 quad with its own memory and I/O. The NUMA-Q, like the Exemplar, maintains a cache of remote accesses. However, in NUMA-Q this cache is part of the shared memory adapter and is limited to 32MB. The OS manages some of the performance issues associated with NUMA systems by using page aware allocation policies including page replication for read-only pages and migration.

Each node in the SGI Origin 2000 consists of two MIPS R10000 processors connected to memory via a ‘‘hub’’. The hub can connect to I/O, memory, and the interconnection network. Up to 64 nodes can be connected using the Xpress Links in a hierarchical hypercube structure, although this is currently limited to 32 nodes because of IRIX scalability. The cellular IRIX OS distributes kernel text and data into cells, where each cell represents a control domain. Resources, such as free memory lists, are managed on a cell basis in order to reduce global contention. IRIX uses hardware assists in Origin to dynamically replicate read-only pages and to migrate pages. IRIX defines a memory locality domain (MLD), to associate certain characteristics (*e.g.*, affinity, page size, enabling page migration) to particular memory regions.

3 PRISM Hardware Architecture and Implementation

This section describes the design of the shared memory adapter and the implementation of the coherence protocol. Although we assume the adapter plugs into a memory bus, the design applies equally to a switch based interconnect between processors and memory within a node, except for a few critical changes. Because of space limitations, we do not describe these changes in this paper.

3.1 Architectural Overview

3.1.1 Physical Memory Address Space

The principal function of the adapter is to provide a variety of ways in which data is transferred between physical memory and memory bus. The memory in the entire system is divided into 4 KB *frames.* A frame at a home node acts as a repository for the contents of that page. Lines from that page may be accessed by any client node processors, as depicted in Figure 7. When a page is to be shared among nodes, each adapter implements a *Client Interface* that monitors the bus transactions which refer to the contents of that page. In addition, its home adapter also implements a *Home Directory Interface* for that page for providing directory based coherence.

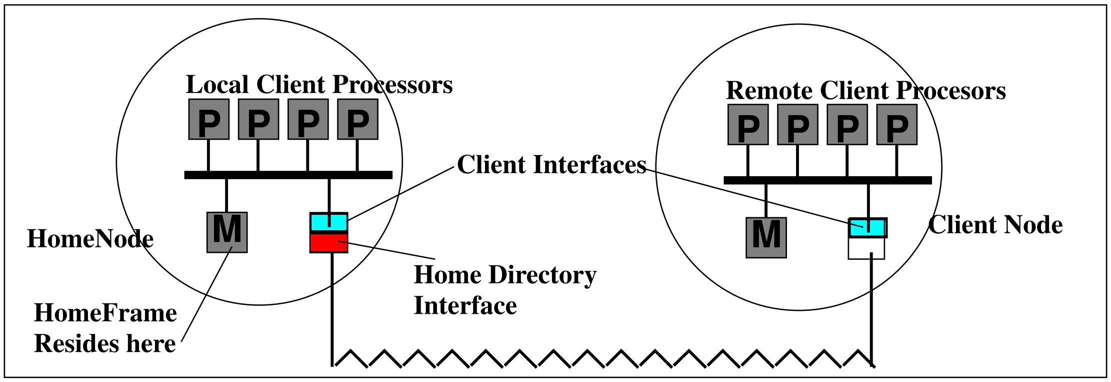


Figure 7 Client and Home Directory Interfaces in an Adapter

The OS at each node views the physical address space as a collection of *frame addresses*. Some of the addresses correspond to frames in that node's memory, some correspond to frames in a remote memory, and some are imaginary addresses that do not correspond to any frame in any memory. A *mode* is associated with each *frame address*, and determines how lines are transferred to and from the client caches. The mode is fixed by the 4 high order bits of the frame address. A more complex implementation could support dynamic assignment of modes to frames. Some of the modes and their meanings are listed in Table 1

Mode Name	How Data is stored/retrieved in this mode	Adapter action for addr transaction in this mode
Local	The page is mapped only in the home node and hence only Local Clients can access the frame.	Adapter ignores all transactions on these addresses letting the coherence mechanism within the node prevail
CC-NUMA	Home node for a global page is statically determined by some bits in its address and only the home node allocates a frame for it. For remote clients, data travels between their caches and home memory frame via the adapter.	The client interface in each adapter acts as a proxy memory, forwarding all access requests to the directory interface in the home adapter, which enforces coherence.
S-COMA	Home node for a global page is assigned by OS. Home node always allocates a frame for the page. In addition, every other node that has clients for the page allocates a frame in its memory. Lines are copied between home frames and other frames on demand. Remote clients access lines via transfers between local memory and caches.	OS sets up the home node information in the adapter when a frame is allocated to a global page. A client interface maintains state for each line in the frame. When an invalid line is accessed, client interface intercepts the transaction, communicates with home directory interface and copies the line from the home node into the Client frame.
LA-NUMA	Home node for a global page is assigned by OS. Home node always allocates a frame for the page. For remote clients, data travels between a cache and home memory via the adapter.	OS sets up the home node information in the adapter when a frame is allocated to a global page. The client interface in each adapter acts as a proxy memory, forwarding all access requests to the directory interface in the home adapter, which enforces coherence.
Command	Command mode addresses are mapped to the adapter and do not have any corresponding memory frame. Only write operations are permitted to these addresses.	When a processor writes to this address, the adapter receives the data and interprets it as a command. OS uses these commands to inform the mapping info (global to frame addresses) to the adapter. Commands are also used to unmap pages, prefetching lines and to read adapter status registers etc.

Table 1: Modes of Physical Frames and how their contents are managed in a system

3.1.2 Coherence Management

For global pages, the Home Directory Interface in the home adapter maintains global coherence. The Home Directory maintains the state for each line of the page and a list of nodes currently sharing the line. It implements a straightforward invalidation based protocol. The states and the transitions are depicted in Figure 8. The transitions take place as the Home Directory receives requests from Client Interfaces for read sharing (S) or exclusive access (X) to the line. Initially the line is not used by any client. Requests for sharing while in a shared state simply append the new node into the shared node list. When a request for shared/exclusive access is received while in the exclusive/shared state, respectively, the transit state is entered, and the home adapter communicates with the clients to downgrade/relinquish their access to the line. After receiving acknowledgements from the clients, the line is returned to the requester and the state is changed to shared/exclusive. To preserve the atomic nature of state changes, no new requests are handled in the transit state. The input queues backup and propagate the pressure to the clients.

In S-COMA mode, a client node has a frame allocated in its memory. The client interface at that node maintains the local state of each line in that frame in the client directory. The system works as follows: At the time of a page fault, the OS allocates from local memory a frame with addresses tagged with S-COMA mode. The OS informs the adapter of the new mapping by writing to a command mode frame. The adapter's client interface initializes the state of all lines in that frame to be invalid.

Figure 9 depicts the subsequent actions and state changes. The states represent the state of the line in this client. The transitions shown by the solid arcs take place when clients at this node make shared (S) or exclusive (X) requests for the line. In the Invalid state, the client interface of the adapter gives a *modified* response on the memory bus, thereby coercing the coherence mechanism in the node to believe that the adapter (acting as a processor) has an exclusive copy of the line. The client interface then communicates with the home adapter, passing the global address. The home adapter returns the data across the network,

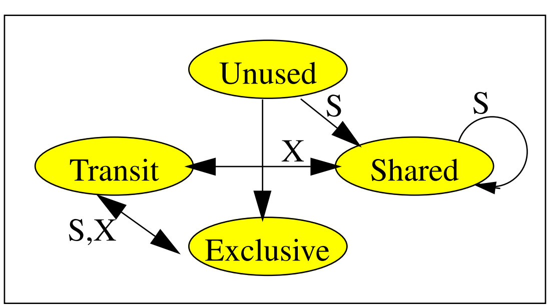


Figure 8 State Changes Maintained by Home Directory Interface

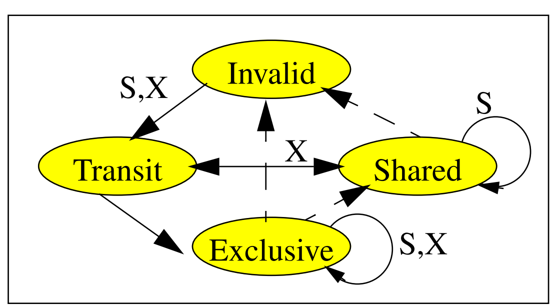


Figure 9 State Changes Maintained by Client Interface in an S-COMA Scheme

and the client adapter places the data on the memory bus. The requesting processor caches it and the memory stores a copy in the frame. During this time, the line is kept in the transit state, so that no new requests are handled for that line.

The client interface simply ignores shared requests in the shared state and all requests in the exclusive states, so that local coherence mechanism prevails. The dashed arcs in the figure correspond to messages sent by the Home Directory Interface for that line. The client either relinquishes the line and goes to Invalid state, or downgrades from exclusive to shared state. Note that client interfaces in the LA-NUMA and CC-NUMA modes do not keep any states for the lines, as a client node does not keep a copy of the page in its memory.

3.1.3 Memory Bus Characteristics

The memory bus shown in Figure 1, on page 4 is an SMP bus with the following features that we consider as fundamental for supporting distributed shared memory systems. It is a split-phase bus with a separate broadcast address bus. All units participate in an address transaction, providing coherence responses. An address transaction completes within a fixed window of time, determining the ensuing data transfer, if any. The data transfer takes place between the participants separately. There is no time constraint on the interval between the completion of an address transaction and the ensuing data transaction (except for a processor timeout after several tens of seconds). When a client requests data, the client interface in the adapter submits a response to indicate the data will be supplied. The adapter can then fetch the data from a remote home and supply the data after reasonable network delays.

The fixed response time for an address transaction, however, imposes severe restrictions on the design, as coherence information must be consulted before giving the address response. This forces the client interfaces to keep the coherence status in fast SRAM, as discussed below.

However, there are situations when a valid coherence response cannot be given within the specified time window for an address transaction. An example of this is a client's request to upgrade a line in its cache from shared to exclusive state. The response cannot be given until the Home is contacted and all other copies of the line are invalidated. To support this, the bus protocol provides two features: i) a *retry* capability, where a bus slave can request the master of an operation to retry the operation after a short period of time (usually 3-30 bus cycles), and ii) a *rerun* capability, where the bus slave can request the master to rerun the operation when the slave becomes ready to handle it. These capabilities allow the bus and other processors on the bus to make progress on operations during the relatively long period while data and control operations are being transported across the network. While *retry* and *reun* responses are sufficient to handle long delays in all situations, they incur additional bus transactions that may be eliminated by using the *modified* response whenever possible, as described above.

Another crucial characteristic of the bus in such a design is the ability for a bus unit to be able to act both as a master as well as a slave. As described earlier, different modes of frame addresses require the adapter to mimic the roles of a processor representing the remote clients and the role of a memory representing remote memory. This ability is critical for a unified adapter design.

3.1.4 Network Characteristics

Because of the distributed nature of the system, message ordering and reliable message delivery between nodes is important. One way of dealing with these situations is to assume nothing about the network and keep enough state in the nodes to resolve errors and race conditions. However, this leads to complex protocols with many states to be maintained in the nodes.

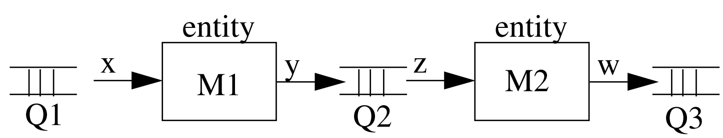
We took an alternative approach, where nodes keep minimal state and most of the state is passed in the messages. We use is the high speed network from the IBM SP. Although it has link level CRC checks and is very robust, the network does not guarantee reliability of messages. Source based routing tables are used which guarantee inorder message transmission on each route, barring omissions due to errors. We implement in the shared memory adapter a network interface that implements end-to-end acknowledgements. Sequence numbers are used for retransmissions and duplicate detection. This made the protocol state machines simpler, without having to worry about errors. In addition, the network implements multiple virtual channels with in-order delivery on each channel. The channels are non-blocking, in the sense that congestion on one channel does not affect messages on other channels (unless there is a logical dependence in the algorithms that send messages on these channels). This facility is crucial for guaranteeing that the distributed components do not get into a deadlock situation when attempting to send different types of messages. The design details are explained below.

3.2 Implementation Overview

This section describes the design and implementation of a prototype shared memory adapter for PRISM that will plug into the memory bus of an RS/6000 SMP. The objective of this prototype is the demonstration of the architectural concepts described earlier. The adapter is a single ASIC chip, with parts operating at 125 and 250 MHz. The memory bus operates at 125MHz. Most of the state information that is proportional to the size of the system is stored in off-chip SRAMs. The adapter can be scaled up by increasing the sizes of these memories. To monitor the adapter chip and to implement certain policies and exploratory features, the adapter card is equipped with an independent processor, referred to as the *CoProcessor*, with a private memory. The coprocessor has no direct access to the memory bus, other than through the adapter chip.

3.2.1 Design Principles

The design of the adapter is dominated by dataflow in order to expose maximal concurrency in the logic and minimize centralized control logic. The logic is broken into a few functional modules that are connected via hardware queues. A module dequeues a request from its input queue, processes it and communicates with other modules by sending messages to their input queues. A module is blocked when attempting to put a message into a full queue. Recursively, each module is organized internally in the same manner. The body of a module is a set of actors interconnected by input and output arcs. Each actor is modelled as a VHDL entity with a single process in it. Each entity keeps its data structures private. All interactions are through explicit messages, which carry most of the state information with them..



The construct {*connect* M1 y M2 z} automatically creates the buffers Q2 and the necessary handshaking signals between them such that

1. Values written by M1 on y will be delivered to M2 on z in that order
2. M2 gets blocked if it attempts to read z when Q2 is empty
3. M1 gets blocked if it attempts to write y when Q2 is full
4. If M1 writes y in a cycle, M2 should be able to read z in the next cycle, even if Q2 is empty

The construct if {rdy x,y} then ...code... end if; automatically generates logic to examine whether x is available and whether Q2 has space to write y. If so, x is consumed, the code is executed and y is output. Otherwise x is not consumed and y is not produced.

Figure 10 Dataflow Organization and High-Level Hardware Description Language Constructs for firing Actors

3.2.2 Basic Dataflow

The basic building block used in the design is shown in Figure 10. An entity is an actor that receives inputs and produces outputs. The interconnection of the actors is made by connecting the outputs of an actor to inputs of other actors. The first 3 properties shown are the usual pipeline properties. The last property ensures that the insertion of the intermediate buffers (Q2 in Figure 10) *does not introduce additional delay*. It works as in a conventional design where there is one latch between two connected entities.

To facilitate high level coding, certain useful constructs are designed as shown. A pre-processor automatically generates logic for the necessary handshaking signals, so that all the listed properties are satisfied. Given this apparatus, the designer can concentrate on breaking up the algorithm into as many actors as possible, write their behavior at high level and connect them up. Most of the difficulty in logic design is in the interconnections of components and these facilities simplify that work. Furthermore, since data structures are all local to an entity, one can easily partition the network of actors along the actor boundaries. Since every connection between two actors goes through a latch, the timing assertions need to be applied only within each actor. In our implementation, interconnecting arcs have 2 intermediate buffers, which is the minimum required to satisfy the last property listed. We use the same mechanism with larger number of buffers (which we call a Queue) between two large modules

3.2.3 Component Organization

The adapter chip is broken down into a few modules interconnected by queues as depicted in Figure 11. The queues mostly contain control information about a transaction. Some of these transactions may have to carry the 128 bytes of cache line data. Passing data through the modules results in wide paths. Instead, 3 data banks are implemented using on chip SRAMs. The data is deposited in them and the messages carry index into the data banks.

The Host Bus Interface is responsible for accessing the memory bus. It accepts requests from the bus master module and arbitrates for the bus and submits the master transactions. It also snoops on the bus and based on the mode of the frame address, it takes different actions as noted in Table 1. For S-COMA transactions, the client state is consulted (CS-SRAM) as detailed in the next section and the appropriate

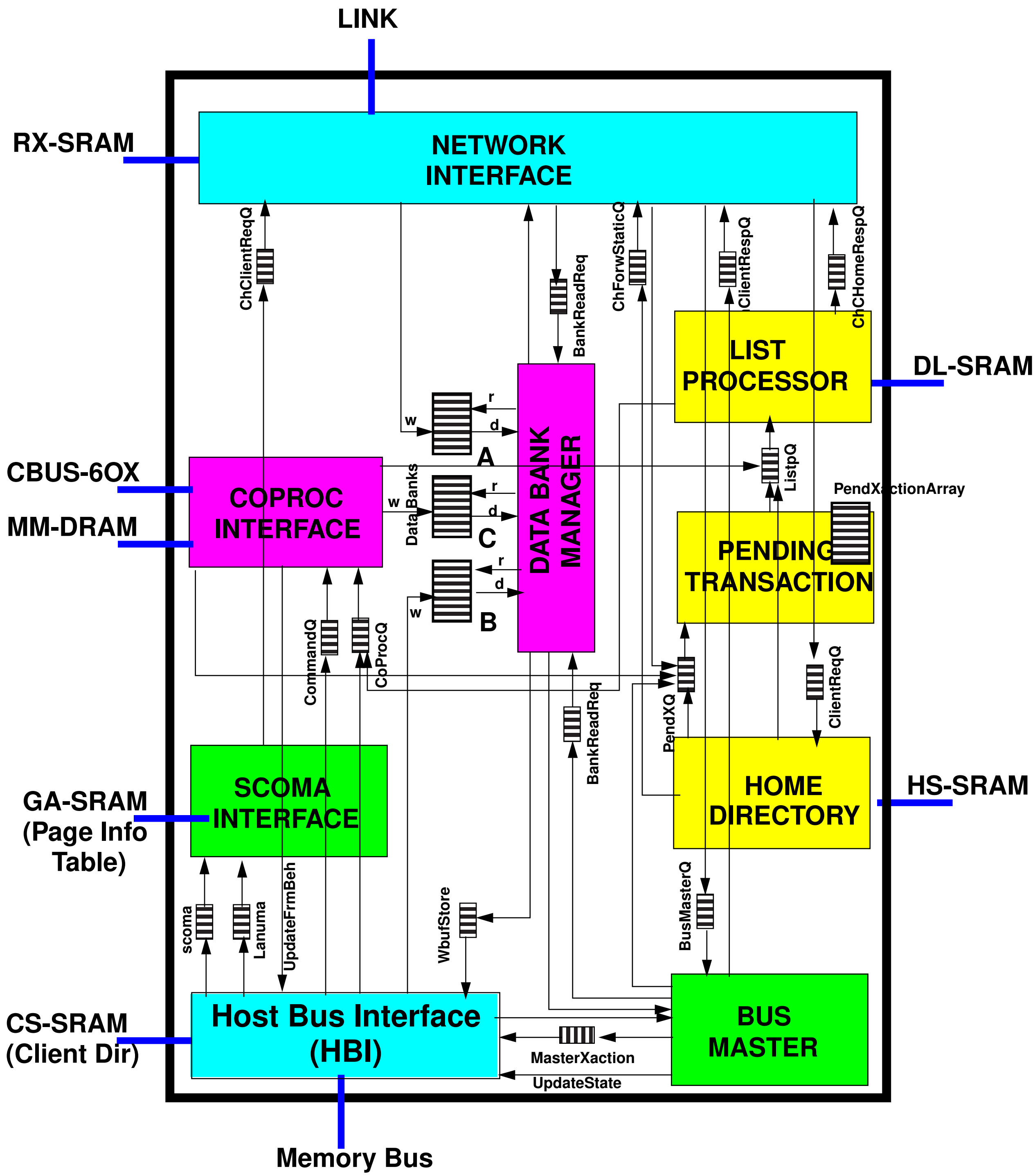


Figure 11 Modules and Interconnections within the Adapter ASIC.

bus responses are given. The transaction is then passed to the S-COMA interface. LA-NUMA transactions are also passed to the S-COMA interface.

Command mode transactions are passed to the Coprocessor interface. The S-COMA interface reads from an external memory (GA-SRAM) which implements the Page Information Table and gives the global address for the frame and the home node number. The information is used to package a message for the Home Directory Interface and sent to the Network Interface (as the home may be in a different node).

The Home Directory module receives (via the Network Interface) requests from clients. It consults the global state of the line (HS-SRAM) and determines the action. Two possible actions can take place: It can send a message to the Bus Master to read a local copy of the line and forward it to the client. Alternatively, it can send a message to the List processor instructing it to send invalidation messages to the clients holding a copy of the line. In this case, the state in HS-SRAM is set to transit, so that new requests on this line will not be processed. The details of the List processor are given in subsequent sections. It sends invalidation messages to the list of nodes which are read from DL-SRAM. It also makes an entry into a table maintained by the Pending transaction module. The pending transaction module is responsible for receiving the acknowledgment messages from the clients (via the Network Interface) and count them down. When the count reaches zero, a response is packaged and sent to the requesting client.

The Network Interface is responsible for sending and receiving messages from other adapters and the details are described in the following sections. The Coprocessor interface has two queues: All command

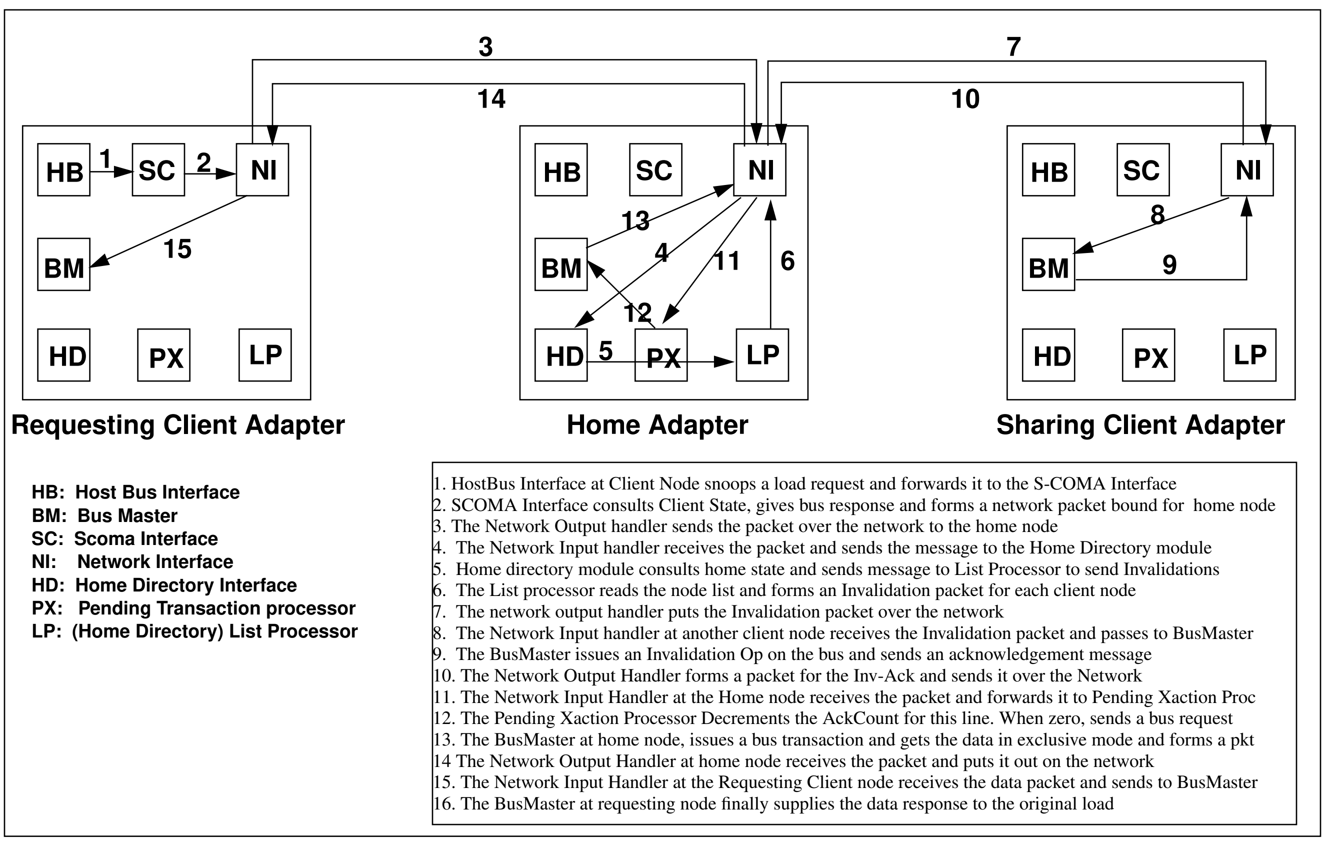


Figure 12 Sequence of Steps executed when a client requests exclusive access for a line held by another client

transactions are entered into the input queue which the Coprocessor can read. The Coprocessor can write commands into an output queue. The commands are of two types: they can be transactions the bus master is instructed to put onto the bus; they can be commands to set specific values to registers on the chip or to locations in one of the SRAMs.

Figure 12 illustrates a typical path through these components when a client requests for exclusive access of a line held in shared state by another client. Step 1 starts when the requesting client issues a bus transaction for a line. After traversing all the 15 steps, finally the bus master responds with the data. The steps 6 through 11 may have several concurrent instantiations. In step 6, LP submits one invalidate transaction for each shared client node, and sets up the count down for PX. In step 11, PX counts down the corresponding acknowledgments and finally when the count reaches zero, it proceeds to step 12.

3.2.4 Client Directory Organization in S-COMA

In S-COMA mode, the client directory must be consulted in order to determine the bus response, as illustrated by the state diagram of Figure 9. There are 4 states and hence 2 bits are needed for each line to be monitored. As shown in Figure 11, we use an external SRAM to store the states. The host bus interface that snoops on the bus transactions must read this storage, respond to the bus and also may have to change the state to transit..

In addition, when the transaction is finally serviced by the home directory, the state must be set back to shared or exclusive as per the transaction. This might, in general, require a read, modify and write into the memory containing the states. The timing constraints to do all these operations are depicted in Figure 13. The figure shows the bus cycles and the events that take place as a result of a snooped transaction in S-COMA mode. The address response must be returned exactly in 5 bus cycles after seeing the address

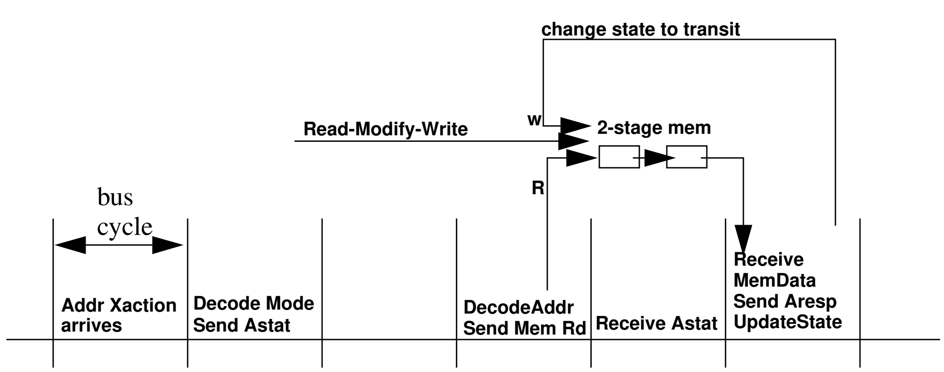


Figure 13 Timing Constraints for checking the Client state in Scoma Mode

transaction on the bus. Even if we use a fast SRAM that operates at double the speed of the bus and has a 2 stage pipeline, one can see the complexity of the logic involved. Furthermore, additional constraints could be imposed by technology: The memory word size could be a few bytes, forcing us to read and write a few bytes for each 2-bit change; The memory chip might use a single data bus for read and write, thereby requiring some dead cycles when reads and writes are interspersed. Our implementation uses a small on chip cache for the words of the SRAM, so that reads and writes can be done in the cache. We also incorporated checks to retry an S-COMA bus transaction when its state bits cannot be read in time

3.2.5 Message Types and Deadlock Avoidance

In the coherence protocol there are 4 main types of messages that travel across the network as illustrated in Figure 12. (A) Client Request messages such as in step 3, (B) Invalidation messages such as in step 7, (C) Acknowledgment messages such as in step 10 and (D) Home Response messages such as in step 14. The message types have a hierarchical relationship: A node servicing a Type A message may have to generate a Type B message. Denote this by A=>B. Similarly we have B=>C, C=>D and D=>A.

Thus, if we have a single network and several instances of these types of messages are in flight, we may get into a deadlock as resources at each node are limited. One way to avoid this is to assume that there are independent virtual channels for each type of message and send Type A messages on Channel A and so forth. Furthermore, assume that a node has the policy to receive and service a type of message as long as it has resources to send out any possible outgoing messages generated in that service. Then we can show that the 4 message types can be serviced without deadlock.

In the most congested situation, Type D messages can always be received and serviced as they do not require to send out any messages. After all Type D are serviced, Type C messages can be serviced and so on. By induction, we will always make progress if we have 4 independent channels. In our implementation, a message of Type A is accepted only when there is an empty slot in the tables of the Pending Transaction Module to keep track of its progress. This ensures that a Type C message can always be received and serviced (since its service is to count down). As long as we have the entry to record the count, we can always send the Type D message for the entry with zero count any time later when that channel is free. Using this property, we can send messages of Types B and D on one channel and get away with 3 independent channels.

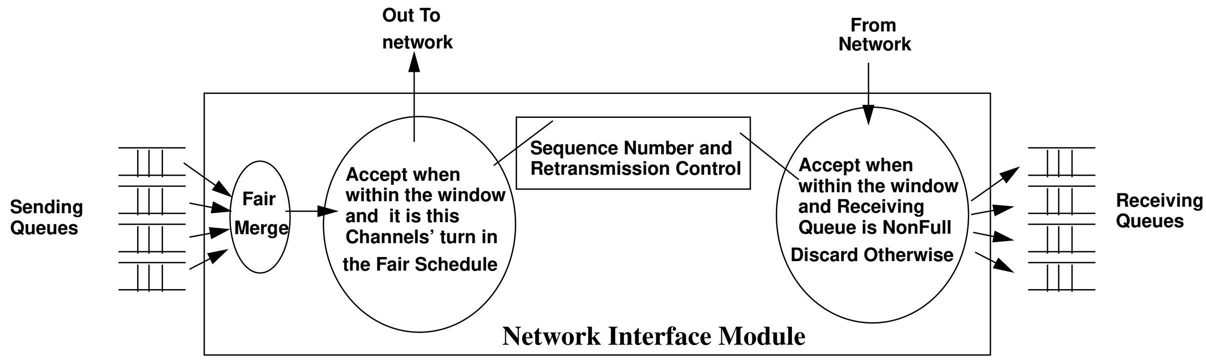


Figure 14 Schema Used by Network Interface Module to Implement Multiple Virtual Channels

This message typing strategy is modeled using the Murphi system and the coherence protocol was run to show that progress can always be made. The introduction of Home node migration, which is discussed later, introduces the need for 2 additional channels to avoid deadlocks.

The strategy used to implement multiple virtual channels is depicted in Figure 14. Each channel is associated with an engine that supports the capability to deliver packets in a reliable, sequenced, error free manner to every other node attached to the network. For this purpose, each channel engine maintains an independent stream of sequence numbers for packets to a corresponding channel engine at each partner node in the system. Each channel engine also supports recovery from soft errors on the network, by allowing retransmission capability for soft errors. While the bit error rate over the cables is extremely low, it is not precisely zero, since the network is distributed across a wide area (10's of meters), and is subject to electromagnetic interference, and interactions between distributed pieces of hardware. The hardware therefore buffers all transmitted data packets until corresponding ACK indications have been received from their target node channel engines, and recovers transmission through sender retransmission, if ACKs have not been received within programmed time span following data transmission. Ordering per destination is maintained on each channel, using end-to-end sequence numbers. Packets received with sequence numbers different than the expected sequence number will be discarded and the sender will recover by retransmission after a time-out. The sending side of the Network Interface ensures independence of the channels by polling the queues in a fair manner, so that no outgoing queue is starved forever. On the receiving side, a packet is accepted within the expected window only when the corresponding receiving queue is non-empty. The messages carry identification of the queue for this purpose. Packets destined for full queues are discarded forcing the sender to retransmit after time-out.

3.2.6 Directory Structure

The Home Directory is split into 2 parts: States and Lists of Sharing Nodes and these are stored in two

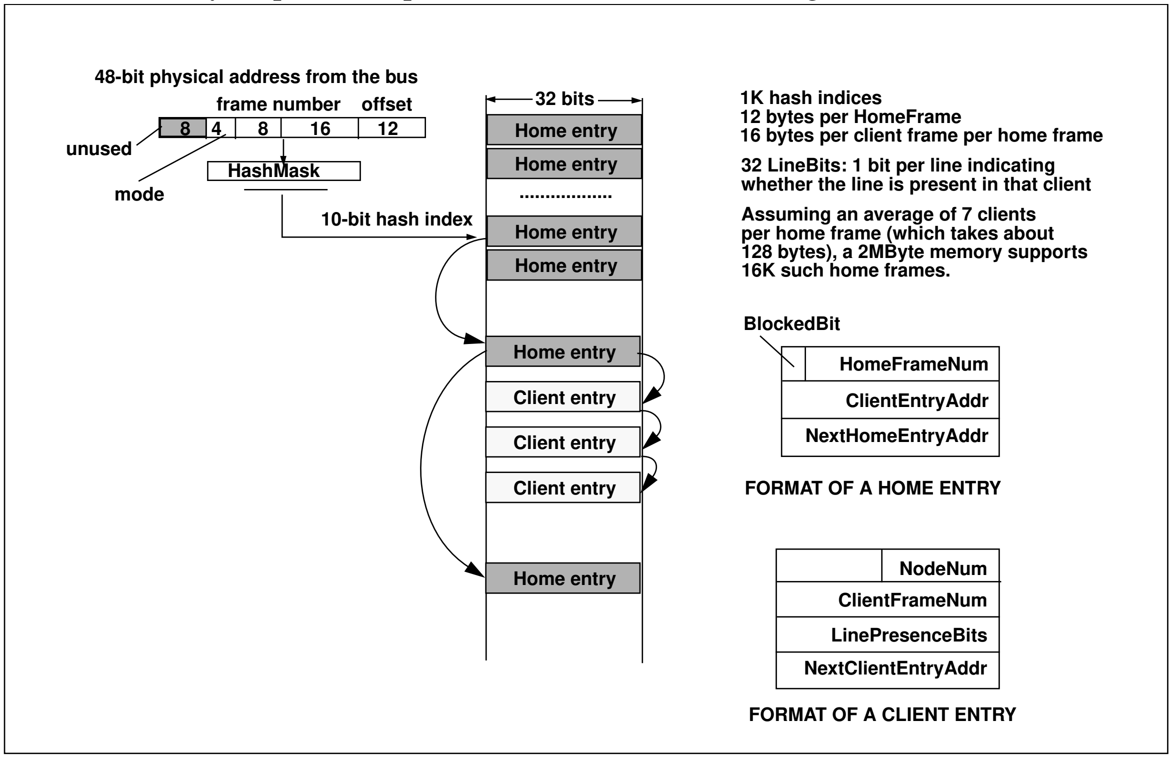


Figure 15 Organization of the Shared Lists of Nodes in the Home Directory

separate external memories. For each line in a home frame, the 2-bit state (as depicted in Figure 8) is stored in an external memory (HS-SRAM in Figure 11). Its organization is identical to the 2-bit client state stored in CS-SRAM. The Home Directory module reads the state for the line and determines what action must be taken.

The data structures storing the lists of sharing nodes reside in a separate memory (DL-SRAM in Figure 11) and are organized as depicted in Figure 15. The directory has two types of entries: *Home Entry* and *Client Entry.* The home frame entries are organized as a hash table consisting of 1K entries. Using a programmable hash mask, 10 bits are extracted from the frame number to form an index into the 1K entry hash table. Each home frame entry is 12 bytes long as shown and contains the home frame number, a pointer for the linked list of home frame entries in this bucket and a pointer to the head of the linked list of clients sharing this home frame. Each client frame entry has the client node number, the client's frame number, a 32-bit string indicating which lines in that frame are shared by that client and a link pointer to other clients on that list. Assuming an average of 6 to 7 clients per home frame, a 2MB memory can support the directory for 16K home frames.

Note that a home entry is allocated when a page is brought into the memory of the home node. Similarly a client entry is allocated when a new client node is mapping the page into its memory. Recall that the OS at the client node must communicate with the OS in the home node in order to map a frame. During this communication, the OS at home node causes a new client entry to be allocated and added to the linked list. The allocation and deallocation of these linked list entries is done by the microcode running on the Coprocessor, under commands from the OS. The Coprocessor can send messages to the List processor to read/write the list memory any time, to set or delete the entries. The hardware does not get involved in the memory management issues. The list processor module is an accelerator to traverse through the linked lists setup by the Coprocessor.

The above organization works well when the number of clients for a home frame is relatively small. When this gets too large, a different organization (e.g., a bit vectors for client nodes for each line) may be preferable. While the list processor facilitates fast searching of the linked lists, it does not preclude alternative organizations that the Coprocessor might manage. To accommodate this, the home entry has a *BlockedBit*. When this is set, the list processor simply forwards the input request to the Coprocessor, which can implement an alternative organization for the list. Since the addition and deletion of home and client entries is done by the CoProcessor, the switch to software can be orchestrated dynamically at any time by the CoProcessor.

At the time this paper is written, the implementation is in the simulation stage. Most of the logic has been written and some critical parts have gone through the synthesis phase. Initial planning of the physical design phase is also in progress concurrently.

4 PRISM OS Design and Implementation

PRISM uses a distributed OS structure with multiple, independent kernels that each manages the resources within a single node. The kernels cooperate to provide globally shared memory segments, but do not themselves rely on the shared memory. This structure promotes scalability and fault isolation between the nodes. If a node fails, the rest of the nodes may continue running, although applications using resources on the failed node may be terminated.

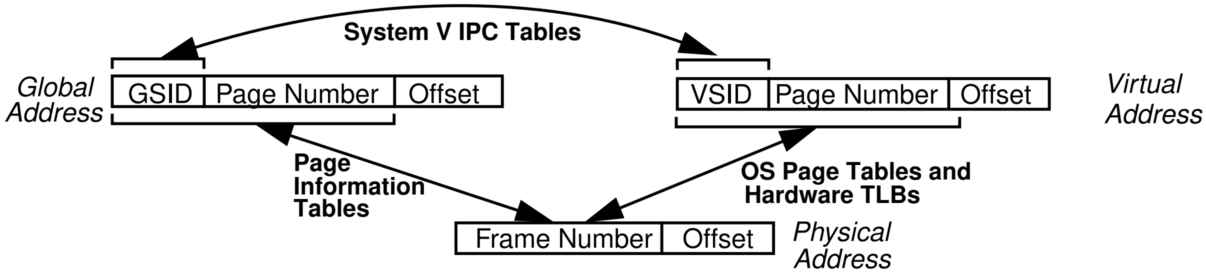


Figure 16 Mappings between virtual, physical, and global addresses. Labels on the arrows indicate where the mappings are maintained in the system.

The novel features provided by the OS are: i) user controlled binding of variable-sized virtual address regions to global segments, and ii) the association of modes to shared memory pages and the page frames backing them. These features are enabled by a set of OS services: Global Naming and Binding, Page Mode Binding, and External Paging.

4.1 Global Naming and Binding.

Like S-COMA systems, PRISM provides a separate system-wide global address space. Figure 16 illustrates the address formats for the system-supported address spaces. Virtual addresses are composed of a virtual segment identifier (VSID), a page number and an offset. Physical addresses are composed of a frame number and an offset. Global addresses are composed of a global segment identifier (GSID), a page number, and an offset.

Global binding occurs whenever virtual addresses are attached to global addresses, and involves global coordination. In CC-NUMA systems, global binding occurs at page fault time because physical addresses serve as global addresses. In S-COMA systems, global binding occurs whenever a virtual page is mapped to a global page, also at page fault time [17]. In contrast, global binding occurs in PRISM when the application explicitly attaches to a shared memory segment. This allows user-level software to dictate when global binding occurs, and to amortize the cost of global binding over large segments.

In PRISM, an application gains access to shared memory using globalized versions of System V shared memory [3]. In order to allow multiple processes of a parallel application to live within a single, globally shared address space, the application loader may create global segments and attach the virtual address space of each of its processes to the global segments at identical virtual addresses. Processes may also share memory by explicitly calling the globalized System V shared memory routines.

4.2 Page Mode Binding

Recall that the adapter takes different actions based on page frame modes. During a page fault, the OS allocates a page frame from one of the modes. For virtual pages attached to non-global segments, the OS allocates local-mode frames. For virtual pages attached to global segments, the OS at a home node allocates S-COMA mode frames, while the OS at a client node has a choice of between S-COMA and LA-NUMA mode frames.

The OS associates a page mode with each virtual page attached to a global segment. The best choice of page modes depends on application and system characteristics. For the choice between S-COMA and LA-NUMA modes, the results of Section 6 suggest that a good run-time strategy is to allocate pages in S-COMA mode if there is space in the page cache. The OS also provides a system call for the user to suggest the desired mode. The mode of a page may be changed dynamically and independently at each node.

4.3 External Paging

The OS needs to communicate with the adapter and with remote nodes during page faults and page-outs. When a page fault occurs at the home node, the kernel allocates and initializes a page frame and informs the local adapter about the new binding between the page frame and the global page associated with the faulting page. The adapter inserts the translation into the PIT and initializes the page's client directory state to *Exclusive*.

When a page fault occurs at a client node, the kernel sends a message to the home node to ensure that the global page is paged-in at home. The home node maps in the page if necessary, and adds the client node to a list of clients for the page. It then returns the home frame number to the client. Upon receiving a response from the home node, the client node's kernel informs its adapter about the new binding between the global page and the page frame, as well as the home node and home frame number for that page. The adapter inserts the translation into the PIT and initializes the page's fine grain tags to *Invalid*.

During a home node page-out, the kernel sends a message to all clients for the page requesting them to page out their corresponding client page frames and write back any modified data to the home. Once all clients have acknowledged the page-out request, the home node writes the data out to disk and removes the translation from the PIT. During a client node page-out, the client node's kernel flushes back any modified data to the home node, informs the home node's kernel of the page out, and removes the translation from the PIT.

We ensure that a page is paged-in at home when handling a client node page fault in order to prevent a client node cache miss from causing a page fault at a remote home node. Otherwise, a cache miss may take an arbitrary amount of time to complete, causing the memory bus to time out and generate an error. Deadlock is also possible, *e.g.*, processors may be stalled in an uninterruptible state at cache misses waiting for each other to be interrupted and service a request to page-in a page.

To reduce the number of page-in request messages from client to home, after the initial page fault, client nodes can set a *home-page-status* flag that denote whether a page is mapped in at the home. As long as the flag is set, subsequent page faults on the same page do not need to contact the home node.

4.4 Allocating, addressing and accessing shared memory

We now fit all the pieces together and summarize how shared memory allocated, mapped and accessed in PRISM. The three major steps are:

1. ***Allocating a segment***. The user allocates a global segment by calling `shmget()` with a globally unique key. The local kernel sends a message to a global IPC server that checks if the segment has been previously allocated. If not, the IPC server requests all home nodes for the segment to create the segment locally. Once the segment has been allocated at all the home nodes, the IPC server returns a global segment id (`gsid`) that identifies the segment.
2. ***Addressing a segment***. The user attaches and gains addressability to the segment by calling `shmat(gsid,...)`. The kernel sends a message to the global IPC server to increment the attach count for the segment, and sets up a local mapping between the calling process' virtual address space and the global segment.
3. ***Paging-in a page***. The first access to a newly attached page encounters a page fault. The page fault handler determines the global page and the home node for the faulting virtual page using information set up during `shmat`, and contacts the home node to map in the page. The handler allocates a local page frame and informs the local adapter to insert the page frame and global page information into its PIT and set the frame's fine grain tags to invalid. At this point, all the translations between the global, virtual, and physical addresses are set up and memory accesses to the page can now occur at hardware

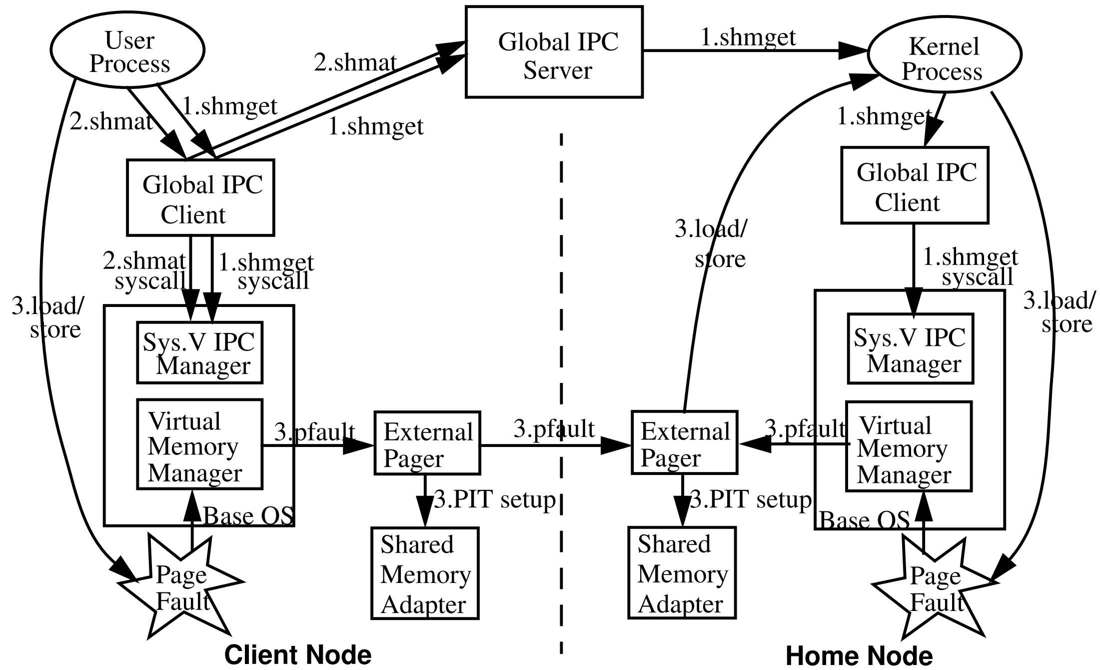


Figure 17 Communication patterns between client and home nodes during allocation, addressing, and page faults of global shared memory segments.

speeds. The processor re-executes the memory access, a cache miss occurs and the shared memory adapter processes it completely in hardware as outlined in Figure 12.

Figure 17 illustrates how the various OS and hardware modules interact during these steps to provide access to globally shared memory. We minimize any necessary modifications to the base AIX operating system by relying on its external pager and kernel extension interfaces. The *Global IPC client* intercepts System V shared memory functions to ensure that they are performed globally. The *Global IPC server* is a logically centralized entity that coordinates the creation and manipulation of global shared memory segments. It communicates with kernel processes on each remote node. The *kernel process* acts as a proxy for remote user processes to create and manipulate global segments. The *external pager* is responsible for communicating page translation information to the shared memory adapter.

4.5 Page Replication and Migration

A concrete example of the benefits of PRISM's system structure of multiple OS images managing node-private address spaces is efficient page replication and migration. Page replication and migration are important techniques for improving data locality, especially for systems with long remote access latencies [22]. With S-COMA and LA-NUMA frames, PRISM provides its OS the option of replicating a page or not by selecting between S-COMA and LA-NUMA frames independently at each node at page fault time.

Contrast this with existing page replication algorithms that are constrained to replicate read-only pages and that need to coordinate the replication globally.

Page migration reduces remote communication by moving the directory and coherence information for cache lines within a page to the node that accesses the page most frequently. It may also reduce the need to write out a page to disk. Since access patterns are difficult to predict and may change dynamically, it is important to be able to migrate the page efficiently. In existing systems, page migration is achieved by unmapping a page and remapping it to a frame that belongs to a different node. This is a time consuming operation that requires TLBs to be invalidated and page tables to be updated globally. In contrast, because PRISM does not encode home node locations in any of its addresses, it is able to migrate a page without changing any address translations and without requiring global TLB invalidations and page table updates. It does so using a technique termed *lazy home migration* that migrates the directory and coherence information of a page to a different node without having to notify all clients of the migration.

4.5.1 Lazy Home Migration Algorithm

The algorithm is implemented in the OS. We define three different roles for nodes:

1. Each global page has a *dynamic home* that maintains the directory for the cache lines in the page and enforces the coherence of each line. This dynamic home may migrate using the scheme described below.
2. Each global page also has a *static home* that tracks the location of the dynamic home of the page, and coordinates the migration of the dynamic home.
3. A node that maps a global page is a *client* for that page.

When a global segment is created, the global memory subsystem assigns a static home to each page of the segment. Initially, the static home and the dynamic home are the same. A client learns about the location of the a page's static home when it attaches to the segment containing that page. When a client faults on a page, the page fault handler communicates with the static home, which returns the location of the dynamic home. The client then communicates with the adapter, telling it the page's frame number, its global page number, and its static and dynamic home. The adapter stores this information into the PIT.

Figure 18 illustrates the interactions between the nodes when processing cache misses and home migrations. When a client node misses on a cache line, it sends a *Request* message to the dynamic home node listed in the PIT entry for that page. The dynamic home processes the request and sends a *Response*

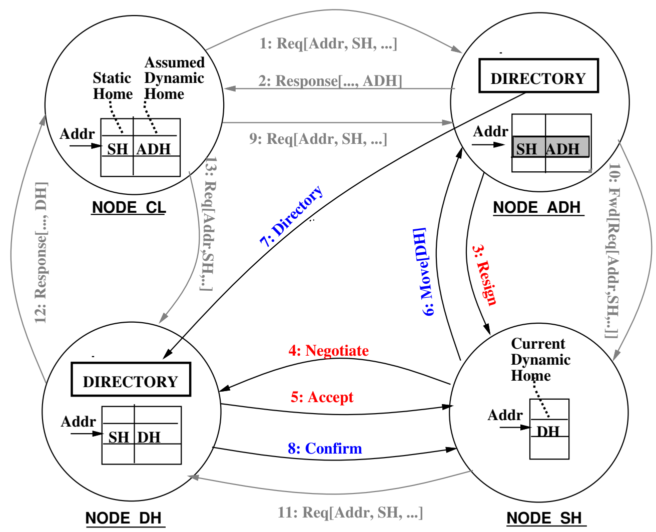


Figure 18 A sample scenario of the sequence of messages sent in a home node migration.

message to the client. The access completes when the *Response* message is received by the client. Note that a client node always sends the static home node location with each *Request* message.

If the request is sent to a node which is neither the static nor dynamic home, the request is forwarded to the static home, using the static home node identifier included in the message. If a static home receives a request for which it is not the dynamic home, the request is forwarded to the current dynamic home for processing. A dynamic home always includes its identity within each response message along with the cache line data, and the client node updates its PIT with this latest dynamic home location upon receipt.

When a dynamic home no longer wants to maintain the home directories, it sends a *Resign* message to the static home. It continues to act as the dynamic home until it receives a *Move[DH]* message from the static home, at which time, it sends the directory to the new dynamic home specified in the *Move[DH*] message and ceases to be the dynamic home.

When a static home receives a *Resign* message, it negotiates with other nodes to select a node (possibly itself) to be the next dynamic home. Once selected, it sends a *Directory message* to the home which is to be the new dynamic home. Upon receipt of a *Confirm* message from that node, the static home then sends a *Move[DH]* message to the current dynamic home, indicating the location of the new dynamic home. It also holds off forwarding any request to the new dynamic home until it gets a confirmation from this new dynamic home

The migration is coordinated only between the static and dynamic homes without involving any of the client nodes. Clients update their view of where the dynamic home is only when they encounter a cache miss. No page translations need to be modified. These features lead to a more efficient migration operation over previously known page migration algorithms.

Migration policies are needed to decide when and where to migrate a home. The adapter's PIT entry contains the static and dynamic home node fields as well as a consultation count field that keeps track of the amount of cache miss traffic for a page that is directed to the dynamic home from remote nodes and from itself. By examining these consultation counts, the operating system can enforce a migration policy. For example, it can migrate the dynamic home to a node that has accessed this page most frequently. Baylor *et al.* [4] evaluates several home migration policies, and demonstrates the importance of efficient home migration.

5 Architectural Requirements on Commodity Components

Our implementation of PRISM relied on certain features and hooks present in the commodity IBM RS/6000 SMP and AIX components that we used. This section abstracts these essential features so that they can be considered in future designs.

5.1 Hardware Components for Inter-Node Cache Coherence

The shared-memory adapter manages the inter-node coherence of shared data. It implements several major functions that involve interacting with the SMP hardware:

1. *Address Filtering:* filtering of addresses based on how corresponding memory is supported.
2. *Transactional Interface to Memory*: independent request-response protocol for memory access
3. *Fault Recovery:* ability to recover when memory transactions fail
4. *Coherence Engine:* an engine with facilities to save state and communicate with other engines

5.1.1 Address Filtering

The filtering agent recognizes memory references and notifies the coherence logic whenever coherence actions need to be performed. In a bus based system, the system must provide a means for attaching a snooping device to the address bus. In a non-bus based system, a memory directory on a node might track the state of a particular line. In these cases, the filtering agent would ideally be integrated with the directory controller. In any case, the filtering agent must be able to examine memory references.

5.1.2 *Transactional Interface To Memory*

Memory can exist in a system in multiple flavors, where each flavor requires processing that spans from a few cycles to longer latencies. To deal uniformly with such diverse memory flavors, a processor must treat memory as a transactional system to which requests are submitted and responses may arrive, possibly in different orders. Both the processor and memory subsystems and the interconnects must not be blocked while a request is being serviced. If necessary, it may "stall" the memory operation, and proceed with another thread of execution. This requires interruptable and restartable memory operations and a split transaction bus, as exists in many modern, high performance systems [9]. Such buses allow long memory latencies without stalling other processors in a node; the bus is freed between the two halves of the split transaction. This is crucial for ensuring good overall node performance under high latencies. Furthermore, the processor-memory interconnect should not distinguish the nature of a port on the interconnect. The behavior must be uniquely identifiable by the responses. For instance, a bus must allow a single bus unit to act as both a bus master and a bus slave.

5.1.3 *Fault Recovery*

As a consequence of the transactional nature of memory access, processors must be able to respond to faults. In a scalable system, it is important that a single memory error not cause widespread failure. Note that global memory errors may result from inter-node communication failures as well as in a node's memory subsystem. Furthermore, it is critical that a failure of one distributed application not cause an error in an unrelated application, or an application which is not using the global memory facility at all.

The hardware must provide a means for trapping data errors on loads and stores, and presenting those errors to software for resolution. The errors must be recoverable; it is unacceptable to take down an entire node because one process on that node referenced global data that was not available. The software which is invoked by the data error handler must limit the scope of the failure. Although it may not be possible to keep the referencing process from terminating, the software must be able to identify the referencing process, and to request that the processor continue to function after the failed instruction is handled.

5.1.4 *Coherence Engine*

Finally, as elaborated in the implementation, the coherence engines must have efficient communication between them. The communications fall into two categories: Infrequent communications for setting up global information (*e.g.,* client node asks home node to bring in the page) and frequent communication for keeping cache coherence. In our implementation, we employed OS-to-OS communication for the former and adapter-to-adapter communication for the latter. A uniform interface for both would be preferable. Finally, coherence engines must be equipped with fast local storage to keep their state.

5.2 Operating System Components

In order to use a commodity operating system, it is important not to require modifications to the base operating system. Fortunately, most modern operating systems provide carefully controlled interfaces for extending the operating system. In terms of managing shared memory, we require extensions to the virtual memory manager through these interfaces.

5.2.1 Extensible Kernel Library

The global memory subsystem requires a set of kernel-level library calls in order to globalize the System V shared memory subsystem. For example, when calls are made to manage a global segment, the calls can be passed to the global memory subsystem and the GSID manager that are implemented as kernel extensions. AIX allows kernel extensions to be added to and removed from a running system.

5.2.2 Involvement In Paging Activity

The operating system must provide the subsystem with a way of managing global shared memory pages and coordinating their translations with the adapter. These actions may be implemented without modifying the OS if it provides an external paging interface like Mach's external pager [24]. Alternatively, one may rely on a mapped file interface and map global segments to a special file system that will receive requests to page data to/from a file, such as those available in Unix or Windows NT [13]. In either case, the interface must notify the external pager each time a page is mapped or unmapped from a frame, regardless of whether data needs to be written out or not. The external pager is invoked when designated memory address are involved in paging. The external pager must also be able to respond to a request from another node, referring to data for which this node is the home. It must be able to initiate a page in, so that the other node can access the data.

6 Experimental Evaluation

This section presents experimental measurements from simulating a 32-processor PRISM system composed of eight SMP nodes with four PowerPC processors each. It demonstrates the tradeoffs between different page modes and evaluates simple run-time policies to select page modes.

6.1 Experimental setup

We use a PowerPC version of Augmint [14], an execution driven simulator, to construct architectural models for the L1 and L2 caches, the system memory, the PowerPC SMP memory bus and its coherence protocol, the adapter and the inter-node coherence protocol, and the network interface. The models also simulate the OS's memory management algorithms by directly incorporating code from the AIX virtual memory management subsystem. The page size is 4 Kbytes. Latency and contention is accounted for at all system resources except the processor internals and network switches.

All cycle times are in terms of processor cycles. Model parameters are chosen to be representative of systems with 5-10 *ns* cycle times. The memory bus is a 16 byte-wide, fully pipelined, split transaction bus with separate address and data paths that operates at half the processor speed. The one way, end-to-end network latency is 120 cycles.

The home directory is modeled as DRAM with an 8K-entry cache to speed up directory access. We assume a full-map directory. A directory cache hit takes 2 cycles, while a directory cache miss takes 22 cycles. The PIT is modeled as SRAM with a 2-cycle lookup time. PIT entries include home frame numbers, but the directory entries do not include client frame numbers. This optimizes reverse translation of global to physical addresses at home nodes, but not reverse translation for invalidation messages from the home to the client nodes. Reverse translation at client nodes use a hash algorithm to search the PIT. Table

2 presents the uncontended cache miss latencies and paging overheads of the simulated system, as measured by a memory latency microbenchmark.

Memory Access Type	Latency (cycles)
L1 miss, L2 hit	12
Uncached, line in local memory	36
Uncached, line in remote memory	573
2-party read/write to modified line	608
3-party read/write to modified line	866
2-party write to shared line	608
(3+n)-party write to shared line	1142+80n
TLB miss	30
In-core page fault, local home	2300
In-core page fault, remote home	4400

Table 2: Memory access latencies and page fault overheads of the simulated system as measured by a memory performance microbenchmark

6.2 Applications and machine configurations

To determine the benefits of the flexibility of choosing page modes, we ran eight applications from the SPLASH-I and -II benchmark suites [23] under various page mode policies. The applications, listed in Table 3, are compiled using IBM's XLC compiler at optimization level `-O2`. The measurements are taken only during the parallel phase of these applications. Memory behavior is simulated for data accesses to both private and shared memory, and homes for shared memory pages are assigned round robin across the nodes.

The SPLASH benchmarks have small working set sizes. We find that with a 16-KB L1 cache and a 1-MB L2 cache, the working set fits within the L2 cache and the choice of page modes does not affect performance significantly. Communication related coherence traffic (as opposed to capacity or conflict related traffic) incurs the same cost in either S-COMA or LA-NUMA page modes and dominates under these conditions. To expose the effect of capacity related misses that occur under more realistic problem sizes, we simulate an 8-KB L1 cache and a 32-KB L2 cache, as in [6].

Application	Problem Description and Size
Barnes	Hierarchical N-body, 8K particles, 4 iters
FFT	Fast-Fourier Transform, 64K complex doubles
LU	Blocked LU decomposition, 512x512 matrix, 16x16 blocks
MP3D	Rarefied air flow simulation, 20,000 particles, 5 iters
Ocean	Simulation of ocean currents, 258x258 grid
Radix	Radix sort, 1M integer keys, radix 1K
Water-Nsq	O(n^2) water molecule simulation, 512 molecules, 3 iters
Water-Spa	O(n) water molecule simulation, 512 molecules, 3 iters

Table 3: Application benchmark types and data sets.

We compare the application performance under the following page mode policies:

- SCOMA All shared pages are allocated in S-COMA mode and the page cache in memory is large enough to fit all the pages. This simulates an infinite page cache and represents an optimal configuration because there are no capacity misses to remote nodes.
- LANUMA All shared client pages are allocated in LA-NUMA mode. This results in the same performance characteristics as a CC-NUMA machine, except for PIT lookup to translate between local and global addresses.
- SCOMA-70 The page cache size is limited to 70% of the maximum number of client S-COMA page frames allocated on each node in the SCOMA configuration. Note that this is a static measure, and the number of client pages in active use may actually be smaller. Page-outs select the Least-Recently-Used (LRU) S-COMA client page frame for replacement. The LRU considers only accesses from local processors.

The following adaptive policies use a page cache size identical to SCOMA-70's, and allocate S-COMA client page frames until the page cache is full. Subsequent client page faults make a choice between allocating S-COMA and LA-NUMA frames.

- Dyn-FCFS The OS maps new pages using LA-NUMA mode frames once the page-cache is full.
- Dyn-Util The OS queries the local adapter for the S-COMA client page frame with the largest number of fine grain tags in *Invalid* state. Frames with fine grain tags in *Transit* state are skipped. The page currently mapped to that frame is unmapped and set to LA-NUMA mode so that future page faults on this page *at this node* use LA-NUMA frames. The newly freed S-COMA frame is then reallocated to the

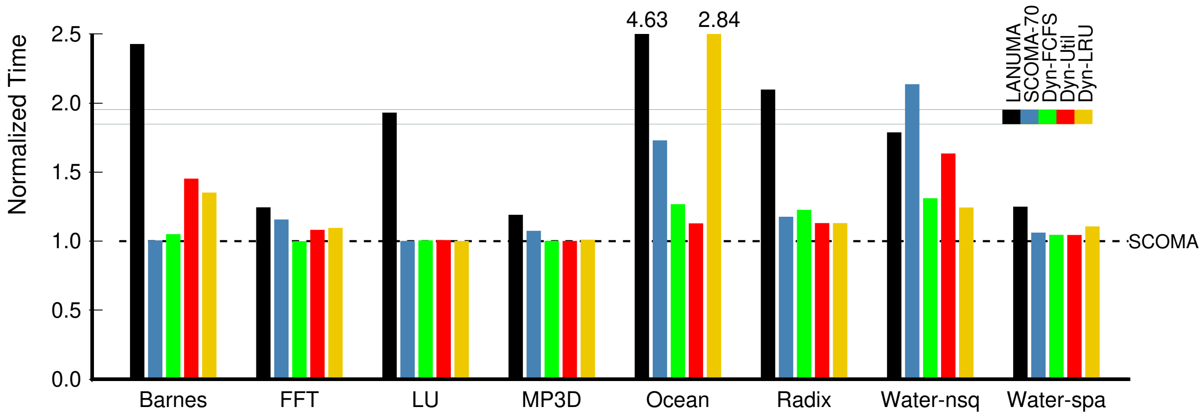


Figure 19 Execution time under different page modes, normalized to SCOMA execution time.

faulting page. The policy tries to target pages that are either lightly utilized or used for communication data and changes them from S-COMA to LA-NUMA mode.

- Dyn-LRU As in SCOMA-70, the OS pages out the Least Recently Used S-COMA client page frame. In addition, the LRU page is set to LA-NUMA mode so that future page faults on this page *at this node* use LA-NUMA frames. The newly freed frame is then reallocated to the faulting page.

6.3 Results and analysis

Figure 19 presents the execution time of each of the applications under different configurations, normalized to the execution time on SCOMA. SCOMA has the best performance since no page outs are necessary, and capacity related misses are serviced by local memory. However, it also consumes the most memory. Table 4 shows the total number of page frames allocated and the average page frame utilization for both private and shared memory in the SCOMA and LANUMA configurations. Page frame utilization measures the fraction of cache lines within an allocated frame that is actually accessed. This is a static measure that does not take into account the frequency of accesses to each line. SCOMA uses significantly more pages and results in lower page utilization than LANUMA. If there is significant memory pressure, SCOMA suffers from the software analog of cache capacity misses and client pages have to be paged out.

Application	Page Frames Allocated		Average Utilization	
	SCOMA	LANUMA	SCOMA	LANUMA
Barnes	3376	616	0.478	0.576
FFT	4888	976	0.276	0.829
LU	2888	592	0.576	0.873
MP3D	1520	304	0.198	0.677
Ocean	8808	4056	0.732	0.956
Radix	13352	2288	0.330	0.940
Water-Nsq	1232	536	0.753	0.894
Water-Spa	672	160	0.315	0.652

Table 4: Page Consumption and Utilization Statistics.

6.3.1 Static configurations

SCOMA-70 models the effect of paging activity due to SCOMA's higher memory consumption by restricting the page cache size to be 70% of the number of client page frames allocated at each node in SCOMA. The difference between the number of page frames allocated by SCOMA and LANUMA in Table 4 yields the number of client frames.

Application	Remote Misses			Page-Outs
	SCOMA	LANUMA	SCOMA-70	SCOMA-70
Barnes	267651	3348808	295817	8457
FFT	122338	186026	128850	11432
LU	115433	991951	115441	510
MP3D	279970	373081	289065	856
Ocean	629986	8002014	1779388	22457
Radix	254201	1394601	363404	15883
Water-Nsq	111074	970560	521016	68290
Water-Spa	40611	178713	69767	2949

Table 5: The number of remote misses to shared memory that fetch data from a remote node in the static configurations, and the number of client page outs in SCOMA-70.

SCOMA-70 significantly outperforms LANUMA in Barnes, LU, Ocean and Radix. The primary reason is the number of capacity related cache misses. Table 5 shows the number of remote cache misses and client page outs. Capacity related misses account for the extra number of remote misses in LANUMA and SCOMA-70 over SCOMA. In these applications, LANUMA encounters a much larger number of capacity related misses than SCOMA-70. SCOMA-70's page cache reduces the number of remote misses over LANUMA, but at the price of increased paging activity. In Water-nsq, the number of remote misses in LANUMA and SCOMA-70 are comparable but paging overhead tips the balance in favor of LANUMA.

Neither SCOMA-70 nor LANUMA is ideal when cache and memory resources are constrained. In LANUMA, evicted cache lines may need to be written back then refetched from a remote home. In SCOMA-70, the same happens for cache lines in client pages that are paged out. The trade-off depends on application access patterns, the L1/L2 cache size, and the S-COMA page cache size. Previous research [21][25][6] have also observed similar tradeoffs between CC-NUMA, COMA and S-COMA architectures.

6.3.2 Adaptive configurations.

In a number of applications, the static configurations of LANUMA and SCOMA-70 perform significantly worse than the ideal performance indicated by SCOMA. Recall that SCOMA does not encounter capacity related misses. We have an opportunity to improve performance by blending page modes, either through explicit user/compiler selection or run-time adaptation. Here, we investigate three adaptive run-time policies as described above: Dyn-FCFS, Dyn-Util, and Dyn-LRU. All three policies initially service page faults by allocating S-COMA page frames, and differ only in the action they take once the page cache is full. Dyn-FCFS is implemented purely in the OS and does not require any hardware support. Dyn-LRU and Dyn-Util require hardware support in the shared memory adapter to monitor reference traffic to the local page cache. To avoid hardware support, the OS can also approximate Dyn-LRU by using standard pseudo-LRU algorithms developed for normal paging. Note that converting a page from S-COMA to LA-NUMA mode is a purely node-local decision, and run-time policy is invoked only at page faults and do not incur any overhead during normal operation.

Figure 19 shows that the adaptive configurations outperform the static configurations in most cases and perform quite well overall. Table 6 shows that the adaptive configurations significantly reduce both the number of remote misses over LANUMA and the number of client page outs over SCOMA-70. Except for a few cases, they achieve within 10% of ideal performance. However, Barnes on Dyn-Util and Dyn-LRU and Ocean on Dyn-Util perform significantly worse than on a static configuration of SCOMA-70

Table 6 reveals the cause of the poor performance. In these cases, the adaptive configurations experience significantly more remote misses than SCOMA-70. This occurs because *reuse* pages were converted

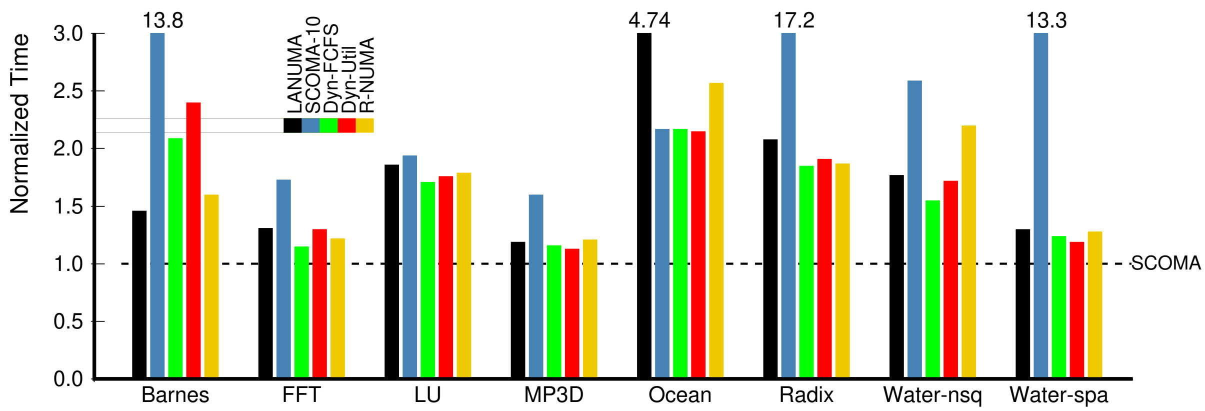


Figure 20 Execution time with a small page cache, normalized to SCOMA.

to LA-NUMA mode, and cache capacity evictions caused the data on those pages to be repeatedly refetched from remote home nodes.

Application	Remote Misses			Page-Outs	
	Dyn-FCFS	Dyn-Util	Dyn-LRU	Dyn-Util	Dyn-LRU
Barnes	709684	1354715	807393	930	895
FFT	122338	122364	124944	5558	5651
LU	119378	116931	115441	509	509
MP3D	280679	280413	283559	404	413
Ocean	1253209	830618	3709983	1449	1464
Radix	492143	495263	368294	3878	3883
Water-Nsq	530448	814619	284861	855	873
Water-Spa	81326	75038	102713	251	258

Table 6: The number of remote misses to shared memory that fetch data from a remote node, and the number of client page-outs in the adaptive configurations. Page-outs do not occur in Dyn-FCFS.

In our experiments, SCOMA-70 generally performs better than LANUMA. These results are contrary to those in Falsafi and Wood [6] where CC-NUMA generally performs better than S-COMA. The reason for this difference lies in the size of the S-COMA page cache. We set the page cache size at 70% of the maximum number of client pages allocated by SCOMA, while Falsafi and Wood fix the page cache size at 320 KB. A 320-KB page cache provides only 5%-25% of the necessary number of client pages for Barnes, FFT, LU, Ocean and Radix, and causes enough paging activity to favor LANUMA.

Figure 20 presents the results when we reduce the size of the page cache to allow only 10% of the maximum required page frames, instead of 70%. It also includes R-NUMA, an adaptive policy based on the one presented in [6] for converting LA-NUMA pages that incur significant reuse traffic to S-COMA mode. The results confirm that under high memory pressure, LANUMA performs better than SCOMA-10, which suffers from severe page thrashing in Barnes, Radix, and Water-spa. They also show that the simple adaptive algorithm Dyn-FCFS is able to match the performance of R-NUMA.

The R-NUMA algorithm for determining when to switch from CC-NUMA to S-COMA may by combined with our adaptive algorithms to implement a policy that switches modes in both directions. Designing and evaluating more sophisticated algorithms is a subject for further research

7 Summary and Conclusions

This paper presents the design and implementation of PRISM, a scalable shared-memory system that integrates the design of its hardware and operating system to provide system scalability and fault containment from both cache coherence and memory management perspectives. It's system design maps naturally onto a cluster of SMPs to provide shared memory across SMP nodes.

The system allows memory pages to be configured with different behaviors so that performance may be tailored to application needs. The OS provides system calls, as well as run-time algorithms to select the behavior of a page independently and dynamically at each node. We introduce the novel concept of Locally-Addressed NUMA (LA-NUMA) memory that allows remote memory addressing via node-local instead of global physical addresses. Avoiding global physical addresses enables localized memory management, lazy home migration, and enhances fault containment.

Simulations of PRISM on a set of SPLASH benchmarks demonstrate the benefits of the flexible choice of S-COMA and LA-NUMA behavior. By allowing pages to be allocated in either S-COMA or LA-NUMA mode independently at each node, PRISM can match and even outperform pure S-COMA and CC-NUMA systems. We examined run-time policies that significantly outperform static LA-NUMA and S-COMA configurations, and usually achieve within 10% of optimal performance.

In conclusion, PRISM's integrated hardware and OS architecture minimizes the amount of global coordination necessary for managing globally shared memory in order to enhance system scalability. It incorporates the positive aspects of existing DSM systems.

8 References

[1] ANSI/IEEE *Standard Scalable Coherent Interface* (#1596-1992).

[2] G. Astfalk and T. Brewer. An Overview of the HP/Convex Exemplar Hardware. `http://www.hp.com/wsg/tech/technical.html`.

[3] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[4] S. Baylor, K. Ekanadham, J. Jann, B.-H. Lim and P. Pattnaik. Lazy Home Migration for Distributed Shared Memory Systems, *Proc. Fourth International Conference on High Performance Computing (HiPC)*, Bangalore, India, December 1997.

[5] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*. December 1995.

[6] B. Falsafi and D. Wood. Reactive-NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, CO, June 1997.

[7] S. Frank, H. Burkhardt, and J. Rothnie. The KSR1: Bridging the Gap between Shared Memory and MPPs. In *Proceedings of the IEEE COMPCON '93*. February 1993.

[8] E. Hagersten, A. Landin, and S. Haridi. DDM -- A Cache-Only Memory Architecture. IEEE *Computer*, September 1992.

[9] J. Hennessy, D. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[10] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *24th Annual International Symposium on Computer Architecture*. May 1997.

[11] K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems. *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*. April 1989.

[12] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*. May 1996.

[13] R. Nagar. Windows NT File System Internals: A Developers Guide. O' Reilly and Associates. September 1997.

[14] .A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures. In *Proceedings of the 1996 IEEE International Conference on Computer Design (ICCD)*, October 1996.

[15] OS Development Group, Hewlett-Packard Convex Technology Center. SPP-UX: An HP-UX Binary Compatible, Microkernel-Based Operating System. White paper from *http://www.hp.com/wsg/tech/technical.html*. March 1996.

[16] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*. April 1994.

[17] A. Saulsbury and A. Nowatzyk. Simple COMA on S3.mp. `http://www.playground.sun.com/pub/S3.mp/simple-coma/isca-95/present.html`, 1995.

[18] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*. January 1995.

[19] Sequent Corporation. Sequent's NUMA-Q Architecture. `http://www.sequent.com/news/papers/qnumaarch/`. 1997.

[20] SGI. Cellular IRIX™ 6.4 Technical Report, *http://www.sgi.com/Technology/whitepapers.html#irix*, 1996.

[21] P. Stenstrom, T. Joe, and A. Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 80-92, May 1992.

[22] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. October, 1996.

[23] S. C. Woo *et al*. The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proceedings of the 22nd *International Symposium on Computer Architecture,* Italy, June 1995.

[24] M. Young et al. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th Symposium on Operating System Principles*, pages 63-76. Nov. 1987.

[25] Z. Zhang and J. Torrellas. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 272-281, San Antonio, TX, February 1996.