Cache Systems

Exploiting Parallelism in Cache Coherency Protocol Engines

Andreas Nowatzyk, Gunes Aybay, Michael Browne,Edmund Kelly,
Michael Parkin, Bill Radke, Sanjay Vishin

Sun Microsystems Computer Corporation
Technology Development Group
Mountain View, CA 94043

Abstract: Shared memory multiprocessors are based on memory models, which are precise contracts between hard- and software that spell out the semantics of memory operations. Scalable systems implementing such memory models rely on cache coherency protocols that use dedicated hardware. This paper discusses the design space for high performance cache coherency controllers and describes the architecture of the programmable protocol engines that were developed for the S3.mp shared memory multiprocessor. S3.mp uses two independent protocol engines, each of which can maintain multiple, concurrent contexts so that maintaining memory consistency does not limit the system performance. Programmability of these engines allows support of multiple memory organizations, including CC-NUMA and S-COMA.

1 Introduction

Shared memory multiprocessors are gaining popularity due to their programming model, which tends to ease software development. In fact, most current multiprocessor systems use the shared memory paradigm, which leads to a growing body of software that is based on the assumption that all threads of a parallel application may access all memory. However, most of these machines use buses, which limits the number of supported processors to about 10. There are numerous proposals for scalable shared memory multiprocessors [1,2,3,4,5] that either have been built or are being implemented. Common to all of these machines is the use of a scalable switching fabric that passes messages between the processing nodes, which consist of one or more processors, I/O and memory. Generally, such switching fabrics do not support broadcasting, which means that the relatively simple memory consistency methods (snooping) from bus based MPs cannot be used. Instead, memory consistency is typically maintained by means of a directory and a cache coherency protocol that defines how memory transactions are translated into message exchange sequences between nodes that share data.

Cache coherency protocols for scalable, non-broadcast systems are more complex than those for bus-based multiprocessors. While it is possible to implement a scalable CC-protocol completely in random logic [3], it is preferable to use a more structured approach that uses a combination of hardware and software, where the hardware is accelerating the common operations and the software is handling infrequent, but complicated, corner cases [1,2].

This paper addresses the problem of designing efficient protocol engines to maintain cache coherency for scalable, shared memory multiprocessors. Following this introduction is a brief overview of several existing implementations. Subsequently, a discussion of the design space for high performance protocol engines addresses the specific needs to support memory consistency. In the remainder of this paper, the architecture of the S3.mp protocol engines is described and evaluated.

1.1 Background

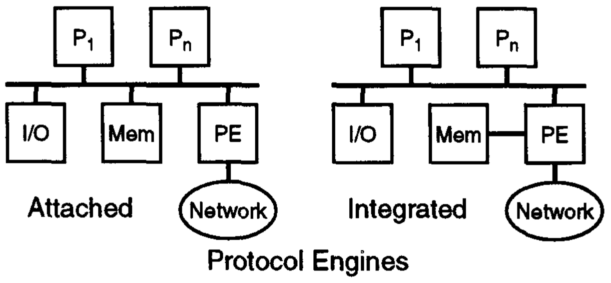
Remote memory references in the Stanford DASH [3] are processed by two units, the *reply controller* (RC) and the *directory controller* (DC), each of which are a collection of hardwired functional units (data paths controlled by finite state machines). Both the RC and DC use field programmable logic devices (FPGAs, Xilinx) and programmable read only memories (PROMs) that offer a very limited amount of programmability, which is mainly used to accommodate late design changes. The RC and DC use different, specific hardware structures and support only a limited degree of concurrency (local bus operations may proceed while a remote access is being processed).

The MIT Alewife [2] machine integrates the protocol processing onto one chip, the A-1000 CMMU [10]. This controller is tightly coupled to one processor chip and serves as its cache controller as well as the memory controller. This central position allows lower latencies. It also simplifies the logic because there is no need to support a local snoopy bus protocol in addition to the global CC-protocol, hence the hardwired coherency engine in the CMMU is smaller than the controllers in DASH. Alewife uses a hybrid approach to protocol processing where exceptional cases are handled by the local processor. This is made possible through the use of a special CPU chip that supports fast context switching. Because of the intimate relationship between the CMMU and a fast context switching CPU, multiple outstanding transactions are supported.

The PLUS [5] multiprocessor uses dedicated hardware (FPGAs) to tie one processor to local memory and an interconnection network. The memory consistency protocol is simpler and requires modifications to the application software.

The main contribution of the Data Diffusion Machine [4] is the concept of a cache only architecture where data is no longer bound to a physical memory location. Instead, all of memory is structured as one multilevel cache hierarchy were data migrates to where it is used. This memory architecture complicates the cache coherency protocol. The first implementations of cache only memory architectures (COMA) systems, such as the KSR-1, used hardwired logic.

Fig. 1. Basic Node Architectures



2 Protocol Engine Design Space

The architecture of a cc-protocol engine (PE) is strongly influenced by the way it is connected to the rest of the processing node. In general, it is only part of the multiprocessor hardware and is embedded in functional blocks that deal with the network, processor and memory interfaces. However for the purpose of providing a high level overview, the discussion of this support logic is deferred to the next section.

In the case of an attached protocol engine (Figure 1) the processing node was not specifically designed to be part of a shared memory multiprocessor, rather it is a conventional computing system (workstation, PC, etc.) that uses a bus to connect one ore more processors to memory and I/O devices. The advantage of the attached design is that the starting point is a fully functional computing environment and that the shared memory interface may be added to existing machines. However, the attached PE needs to deal with a design that did not anticipate its needs. This leads to extra complexity and overhead. For example, a snoopy bus generally assumes that all attached caches can perform the tag-lookup in a relatively short and fixed amount of time. However, the PE is representing a large, remote memory with interconnect latencies that tend to be much higher that local bus transactions. In order to be able to participate in a snoopy bus without slowing it down, the attached PE needs to have the ability to conservatively predict the need to interfere with a bus transaction in a timely manner, which may require significant amount of fast, static memory dedicated to a fast directory lookup table.

In the case of integrated PE, the designer exerts more influence over the memory design. In particular, it becomes possible to increase the memory bandwidth such that remote memory requests can be served without using up bandwidth on the local system bus. Furthermore, since most bus structures allow for variable memory response time, the snooping timing constraints are easy to meet. Continuing this line of architecture evolution is the merging of the PE with the CPU chip, so that the first/second level caches become accessible to the PE without cumbersome and time consuming bus protocols.

PE designs can also be classified by their implementation technologies. Early PE designs were almost exclusively hardwired. However the trend is to more programmable designs because it becomes clear that cc-protocols have not matured yet. In fact this is an active area with considerable innovation potential. For example, automatic detection of migratory data objects has been shown to be effective while requiring only modest changes to a cc-protocol [11, 18]. At the far end of this implementation spectrum is the use of standard RISC processor cores, which are provided by many ASIC vendors as compact, fully tested megacells [12]. While shortening the design cycle, the use of a standard RISC core does however increase the number of cycles necessary to process the cc-protocol.

2.1 System Interfaces

The primary interfaces for the PE are the interconnect network and the connection to the processor. In the case of an integrated PE, it must also interface to the main memory.

For attached PEs, the processor interface is essentially the system bus, which also serves to access main memory. The challenge is to design the PE so that it can deal with a given system bus, which generally did not anticipate the needs of a PE. As mentioned above, participating as a third party in snoopy bus transactions faces stiff timing constraints, which generally requires the use of expensive, fast and power hungry static memory devices (SRAM) to hold tables of memory locations that have been cached by remote nodes. Dealing with a system bus also complicates matters because it generally supports a multitude of transaction types, word sizes and alignments. A particular troublesome issue is the support of bus-locking, which tends to cause deadlocks in a scalable system without broadcasting.

The interconnect network is generally tailored towards the need of the PE. Critical issues are bandwidth, message insertion rates and latency. It is desirable to have the ability to insert multiple messages concurrently. Furthermore, the use of either independent networks for requests and replies or the support for multiple priorities can eliminate the need for deadlock recovery methods.

While the memory interface requirements don't differ much from that of an ordinary processing node, it is advantageous to support multiple banks and to have the ability to process several concurrent requests. Advanced DRAM devices, such as RamBus and S-DRAM, offer improved interfaces that allow multi-banked systems with modest effort.

2.2 Sources of Parallelism

The most obvious source of parallelism within a PE stems from the fact that the PE plays two roles, that of a client which initiates transactions and that of a server that has to respond to remote requests. The demand on both units in terms of number of transactions and the complexity of each transaction is roughly equal.

Higher performance systems generally need ways to hide the latency of remote memory references, most of which result in a demand for more bandwidth and more concurrent transactions. Prefetching and lockup free caches will both supply the PE with multiple transactions before one has completed. Given that it is much easier to design interconnect networks with higher bandwidth than with lower latency, this parallelism can be readily used. However the PE design becomes more complicated due to the need to coordinate multiple transactions, allocate resources in a non-blocking fashion and keep transactions separate. Furthermore, maintaining the proper memory semantics becomes more difficult.

Finally, there is a certain amount of parallelism within each transaction itself, which is best matched with a pipelined design that schedules data movement and control transfers separately.

2.3 Protocol Families

The two main classes of cache coherency protocols are labeled nonuniform memory access (NUMA) and cache-only memory architecture (COMA). The non-uniformity in NUMA refers to the fact that the memory latency becomes a function of the location of the memory block. The PE support for NUMA requires a mechanism to keep track of the cached copies of a particular memory block. Typically, this is done by maintaining a directory. Directories can either be stored at the location of the main memory [1,2,3] or a the location of the (processor-) caches [7,8]. While the latter approach requires less dedicated storage for the directory, it does require a more intimate connection to the processor, which is typically not possible in the attached PE configuration due to the characteristics of the system bus.

In addition to the directory, NUMA systems benefit from a cache that stores the data from remote memory references. It turns out that in PE designs that use the client/server approach, the directory is only updated by the server while the cache for remote data is maintained exclusively by the client.

In addition to the facilities of a NUMA system, a COMA supporting PE needs to have a mechanism to locate the data for a given address. This involves the support for table with semi-associative lookup capability, much like the tag-store of conventional caches.

3 The S3.mp Protocol Engines

The S3.mp scalable, shared memory multiprocessor is an experimental research project that is being implemented by SMCC's Technology Development group. The S3.mp architecture is similar to systems mentioned in section 1.1, however unlike these conventional CC-NUMA MP's, S3.mp is optimized for a large collection of independent and cooperating parallel applications that share common computing resources which may be spatially distributed. Consequently, support for concurrent I/O operations, as in a video server, is important besides the traditional parallel applications. S3.mp nodes may be spatially separated by up to 200m, which means that a S3.mp system could be distributed over an entire building.

Fig. 2. S3.mp System Overview

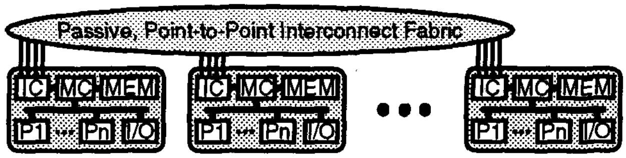


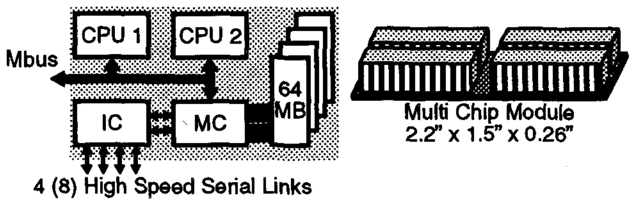
Figure 2 shows the generic S3.mp components: each node is essentially equivalent to a workstation with one or more processors, memory and I/O. The two S3.mp specific components are the interconnect controller (IC) and the memory controller (MC). The IC is a topology independent router that allows the construction of switching fabrics without centralized switches that have a bisection bandwidth that is comparable to that of a Cray T3D, but at much lower costs [13]. The interconnect fabric is composed of either fiber optic links or electrical connections that can carry bit serial data at rates exceeding 1.3 Gbits/sec. For the purpose of the subsequent discussion, the IC network should be regarded as a black box that can transport messages reliably between any two nodes in the system. This interconnect systems offers 4 levels of priorities. Because of the adaptive routing algorithm used by the IC, the message delivery order is not maintained.

The memory controller includes the protocol engines that are responsible for translating between local bus transactions and the messages that are transmitted over the interconnect system. There are two identical protocol engines, each with its own writable microcode memory.

3.1 The S3.mp processing element

Each S3.mp node consists of 2 gate arrays with about 200K gates combined that are part of one multi chip module (Figure 3). The MC interfaces directly to the memory chips, the local processor bus (Mbus) and the interconnect controller (IC). Normally, the MC serves memory operations from the local bus to the local memory, just like a conventional memory controller. However, it maintains a global 64 bit address space and allows the local processor to issue memory operations for remote data. The MC can also initiate transactions on the Mbus as a result of messages received from remote nodes. In the base configuration, two processors are connected to each MC through the Mbus.

Fig. 3. A S3.mp Node



The MC is designed to directly drive a 64 to 256 Mbyte memory array that uses 16 or 64 Mbit synchronous DRAM devices. Coherency is maintained on memory blocks of 32 bytes, which is the size of one cache line on Mbus based systems. The architecture is not tied to a particular cache line size because remote data is cached in the MC and the MC could maintain subblocks or multi-line objects.

The MC uses 18 bits of ECC overhead and 14 bits of directory overhead for every 32-bytes of memory. 14 bits provide sufficient storage to keep a 12-bit node pointer and 2-bit state for each cache line.

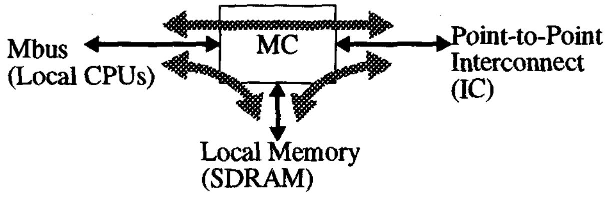
The Mbus of each S3.mp node can be connected to other devices, which is the primary facility to attach I/O devices to the system. In particular, it is possible to plug a S3.mp module into any Mbus slot (for example into a Sparc Station 10 or 20 Workstation).

3.2 The Memory Controller

The MC is responsible for handling accesses to local and remote memory and for implementing directory based cache coherence protocols. In addition to operating as a normal memory controller, the MC performs the following functions:

1. Maintaining the directory information for the local memory under its control
2. Constructing and sending messages to remote nodes on the network initiated by local MBUS transactions that require remote access or in response to messages received from other nodes on the network
3. Performing memory operations and MBUS cycles on the local node in response to messages received from remote nodes
4. Maintaining an Inter-Node Cache (INC) which is used to store a copy of every cache line retrieved from a remote node
5. Sending and receiving diagnostic messages to or from other nodes to program configuration parameters, handle errors, etc.

Fig. 4. Datapaths Through The Mc

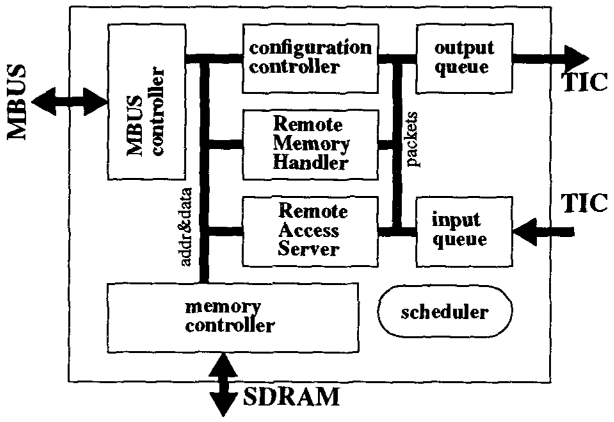


From the CPU's point of view, the MC operates as a virtual bus extender. CPUs connected to the MC on the local Mbus can assume that they are talking to a cache coherent bus. Except for the extra latency in accessing remote locations, details of getting data from remote nodes and distributed cache coherence protocols are completely hidden from these processors. Parallel application software targeted for a shared-bus cache-coherent multiprocessor system, written in accordance with the SPARC memory models [7], should execute correctly on a S3.mp system without any modification. However, performance tuning may be necessary to achieve good performance for some applications. The MC, acting as a virtual bus extender, allows single-threaded programs to utilize the collective physical memory in a cluster of workstations for memory intensive applications and allow uniform access to all I/O devices for high performance server applications.

The MC is structurally divided into the following set of modules (Figure 5):

1. A bus controller to interface to the Mbus
2. A memory sequencer to interface to SDRAMs
3. the protocol engines to implement distributed cache coherency protocols (RAS and RMH)
4. A configuration controller unit to take care of configuration management, errors, diagnostics, etc.
5. Input and output queues for interfacing to IC (or to a general purpose point-to-point interconnect)

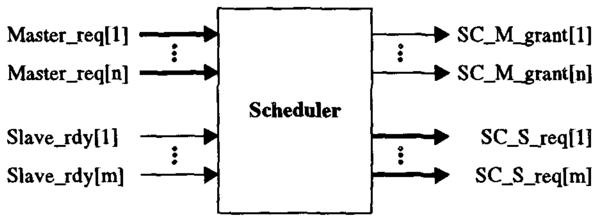
Fig. 5. The MC Structure



3.3 The MC Environment

The MC is designed like a system on a chip. Modules are designed to be as functionally self-sufficient as possible. Each module can generate and receive a small number of transactions. The number of signals going between modules are kept to an absolute minimum. Functional units communicate with each other through an address bus, a data bus and a packet bus. Access to these buses are managed by a central controller, the *scheduler*. The scheduler receives dedicated request lines from each master module and ready lines from each slave module (Figure 6) and sends dedicated grant lines to each master and dedicated request lines to each slave. Requests requiring data or acknowledgments to be returned are handled as split transactions. In addition, most slaves are required to be able to function as a master and initiate transactions to return data and/or acknowledgments.

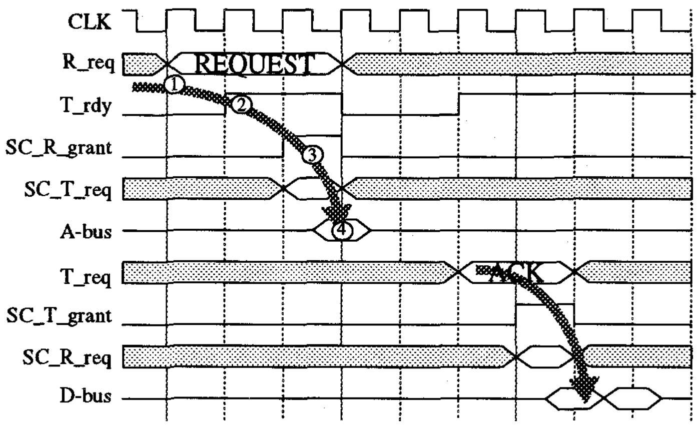
Fig. 6. Centralized Communication Control



A typical transaction proceeds as shown in Figure 7:

1. The initiator asserts its dedicated request line (R_req) to the scheduler. R_req is a multi-bit signal including information about the resources to be used (i.e. A-bus, D-bus, P-bus), the number of cycles required for the transaction, module ID of the target unit and the type of the transaction. Each unit has its own dedicated R_req line going to the scheduler.
2. Slave modules assert a dedicated ready line (T_rdy) to the scheduler whenever they are ready to accept transactions.
3. Every cycle, the scheduler examines request inputs from master modules and ready lines from slave units. When the scheduler determines that all resources required for a request are available, it schedules a transaction for that request. Necessary communication resources are reserved for the duration of the transaction. A grant signal (SC_R_grant) is sent to the requestor and the request type and requestor ID is relayed to the target module (with SC_T_req signal).
4. Requestors are responsible for driving data on the bus as soon as they receive the grant signal from the scheduler. Typically, requestors latch the data into their output registers in the cycle they assert their request. The grant signal from the scheduler is used as an enable to drive data from these registers to the buses in the same cycle the transaction is scheduled.

Fig. 7. Internal Bus Protocol of MC



The main transaction initiators within the MC, *the MBUS Controller, the Remote Memory Handler* and *the Remote Access Server,* compete for access to the memory controller. The memory controller is designed to be able to handle two simultaneous transactions to a 4-banked memory subsystem. The split transaction communication protocol used in the MC reduces the contention on internal buses by reserving the buses only when data transfer is taking place. The data bus has a bandwidth of 1 Gbyte/sec, that connects the Mbus (320 MB/sec peak), the memory (640 MB/sec peak) and the two protocol engines (sharing the I/O queues with 440 MB/sec). Using split transactions such that data is always pushed by the sending unit which initiates arbitration also introduces a level of parallelism *and* pipelining. It is possible to send the address for a transaction on the address bus while an independent data transfer is taking place on the data bus. Moreover, since acknowledges are generated by the target module as new transactions, communication is clock-delay independent. Changes in the cycle timings of transactions (i.e. time taken to respond to a certain request) does not affect the overall functionality.

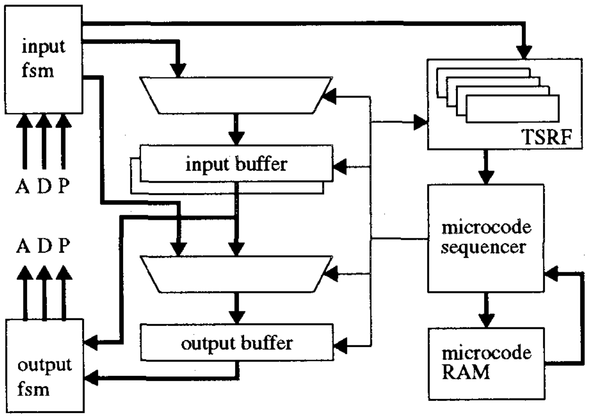
This design methodology does not necessarily result in the fastest possible implementation, however, it makes design management and verification simpler. The concept is similar to object oriented programming. Each module has a small set of externally visible data structures and a set of transactions (methods) that operate on these structures. Modules can be tested independently of the whole system by constructing simple test modules. Multiple implementations of modules can be maintained and cost-performance trade-offs can be postponed until late in the design process. Also, it is easy to reuse parts of the design targeted at different platforms and/or technologies. For example, the memory controller module has a very simple functional definition: it either reads or writes a 32-byte block of memory (a cache line) of memory. This requires that some of the logic to handle byte insertion, etc. is moved into other modules, but it makes the memory controller extremely modular. A Rambus or DRAM based version of the memory controller could be used in a future version of the MC design with minimal interface redesign effort.

This modular design methodology also decreases the complexity of the verification. Part of the development effort is to verify the correctness of the inter-module communication protocol using formal verification tools. Simple module interfaces and the small set of transactions supported by each module makes it possible to deal with the communication protocol at a high level of abstraction where formal verification tools like *Murφ* [14] and SMV [15] are applicable. At the module level, a simple interface and a small set of transactions enable designers to test their modules almost exhaustively before integration of the full-chip model. With the simple interface approach, integration of modules at different levels of abstraction is also possible.

3.4 Protocol Engines - RAS and RMH

The S3.mp cache coherency protocols are implemented by two protocol engines on the MC chip, the *Remote Memory Handler* (RMH) and the *Remote Access Server (RAS)*. The RMH is responsible for locally initiated memory operations that refer to remote memory. It retrieves data from remote nodes, services invalidation and data forwarding requests, and maintains the INC. The RAS serves requests by remote nodes to the local memory. It services data request messages from remote nodes, maintains the directory and generates invalidation and data forwarding messages.

Fig. 8. Protocol Engine Datapath



The RMH and RAS each use an instance of the microcode controlled datapath shown in Figure 8. This is a special purpose protocol controller with a simple instruction set that is tailored specifically towards the execution of cache coherency protocols. Each engine consists of 3 independently operating components: the input FSM, the actual micro controller and an output FSM. The input FSM receives requests from the system interface, which can be either a new packet from the input queue or a request from the bus interface unit. In either case, the input FSM receives the request and places it in the input buffer. It also interprets the request and decides if it belongs to one of the currently executing threads or if a new thread needs to be created. In the later case, the input FSM initializes all registers in the transaction state register file (TSRF) that collectively forms the state of a thread (addresses, program counter, auxiliary data, timer, retry counter, state variables, etc.). A TSRF entry has a total of 121 bits. The micro sequencer may be executing on a different, active thread while the input FSM operates.

The micro sequencer schedules a thread for execution once the input FSM has completed the initialization or when a thread is suspended or terminated while there is at least one other active thread in the TSRF. Context switches essentially require no extra time: the micro sequencer can be executing an instruction from TSRF entry #1 and #2 in adjacent clock cycles (15 nsec or 66 Mhz). The instruction set of the micro-sequencer includes a number of data move, test and set instructions that aid in the assembly and interpretation of messages. It also includes several more complicated instructions, such a performing a 3 way set-associative table lookup or a receive instruction that suspends the current thread until either a matching reply is received or a time-out occurs. Most instructions include a multi-way branch (2-16 ways). Instruction fetch, decode and branch take place within one clock cycle without branch delay slots.

The output FSM is invoked by the micro-sequencer to send replies or requests to other units (memory, output queue and bus interface). It off-loads the need to arbitrate for resources and control the data transfer from the micro-sequencer, which can proceed to work on another thread.

The key point of this design is that it implements a logical pipeline with three stages that receive transactions, process them and distribute the replies via dedicated agents that share resources.

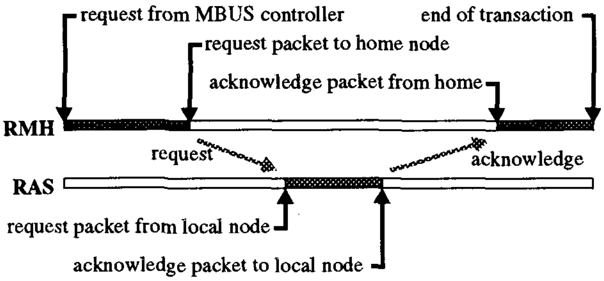
The cache coherency protocols are separated into two parts, the server (RAS microcode) and the client (RMH microcode). Microcode is stored in two on-chip RAMs, so that it is possible to program different protocols.

The typical operation of the protocol engines is shown in Figure 9. In the most common case, there are two kinds of transactions performed by the protocol engines:

1. Transactions that require an acknowledge. This kind of transaction typically has a short burst of local activity (i.e. local cycles, memory cycles) which is terminated by sending a request packet. From this point on, the microcode waits for a reply from a remote node. This wait period is in the order of microseconds in the current implementation of the S3.mp system. When the reply is received, the transaction is terminated with another short burst of local activity.
2. Transactions that do not require an acknowledge. These are typically initiated by receiving a request packet and can be served using only the local resources of the receiving node. They are terminated by sending an acknowledge packet to the requestor.

Due to the long latency associated with waiting for a reply, it would have been inefficient to run the protocol engines in a mode where the microcode sequencer was kept busy for the entire duration of the transaction. This is the rational that led to the design with multiple contexts. The TSRF has several sets of registers to keep track of multiple concurrent transactions. The lowest 7 bits of the cache block address (bits [11:5] of the local address for 32-byte cache lines) are used as a tag for transactions. These 7 bits are preserved during address translation, thus, they effectively divide the global address space into 128 sets. Any protocol engine can be working on *N* of these sets concurrently where *N* is the number of register windows in the TSRF of that protocol engine. The current implementation of the MC chip uses 4 TSRF windows. The optimal number of contexts is a function of the available concurrency, which is limited by the current set of CPUs that issue only one memory reference at a time. 4 contexts roughly match the remote traffic that can be expected from 2 CPUs, provided that there is some support for prefetching and block copy operations that can proceed in the background and may be used for I/O or page migration. Once processors with lockup free caches, speculative execution and the ability to issue multiple outstanding memory references become available, the number of TSRF entries will need to be increased.

Fig. 9. S3.mp Protocol Engine Activity


request from MBUS controller
end of transaction
request packet to home node
acknowledge packet from home
RMH
request
acknowledge
RAS
request packet from local node
acknowledge packet to local node

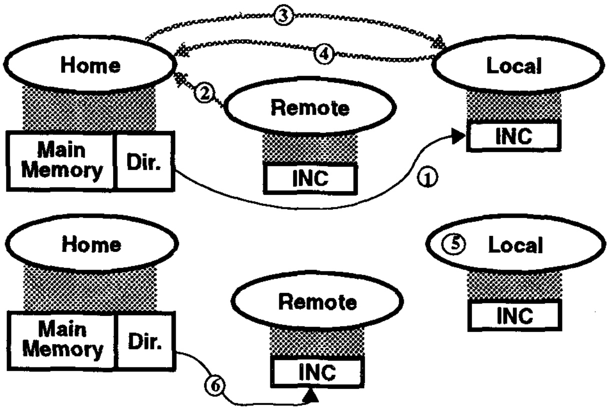
Transactions received by the protocol engines can be of two types: request or acknowledge. Request transactions require a new microcode thread to be generated. For transactions received from the local bus controller, the transaction type relayed by the scheduler is used as an entry point into the microcode. For packets received from the input queue, type information from the packet is used as the entry point. The next available TSRF window is assigned to this transaction and a new thread is generated and marked as ready to execute. When the microcode sequencer detects that this thread is ready to run, it starts executing this thread. The TSRF entry corresponding to this thread becomes part of the state space of the microcode sequencer for the duration of this execution. Until this thread is completed, all other subsequent transactions having the same ID will be suspended by the input FSM. Once the microcode sequencer completes the initial set of local operations and sends a request packet to a remote node, the current thread goes to sleep and any other thread which is marked as ready to execute can start running on the microcode sequencer.

When an acknowledge packet is received by the RAS or the RMH, ID field of this packet is compared to the ID fields of threads that are currently sleeping in that module. A sleeping thread waiting for the particular acknowledge packet is found. This thread is woken up, i.e., marked as ready to run. The microcode sequencer picks up the thread whenever it is available and continues executing after the point where the thread had gone to sleep previously. Basically, a thread goes to sleep whenever it needs to communicate with a remote node and is woken up whenever the acknowledge message is received. Between these two events, microcode sequencer is free to service other transactions.

A set of watchdog timers, one for each TSRF thread, are used to deal with the case where an acknowledge is expected but never received from a remote node. Whenever a thread waits for a reply, its timer is programmed to generate an error acknowledge if an acknowledge is not received within a preset amount of time.

In addition to providing a way of matching waiting threads and acknowledge messages from remote nodes, the ID field provides a convenient way to lock the directory and/or the INC while these data structures are in a transient state. The RAS can potentially lock N out of 128 slices of the directory independently, where N is the number of register windows in the TSRF. The RMH can also lock accesses to the INC with the same mechanism. By convention, the RAS does not touch the INC while the RMH does not access the directory. ID-locking is utilized exclusively by S3.mp directory protocols to implement delay independent operation. The example in Figure 10 illustrates the ID-locking feature.

Fig. 10. INC Locking Mechanism



1. Initially, the directory is in the *Shared_Remote* state and the local node has a valid copy of the cache line.
2. A remote node issues a write to the same line and sends an exclusive ownership request to the home node.
3. The home node sends an invalidation request to the local node.
4. Before receiving the invalidation request from the home node, the local node decides to discard the line and sends an uncache request to the home node. This request and the invalidation request from the home pass each other in the interconnect network.
5. The local node receives an invalidation request while it was expecting an acknowledgment for its uncache request. Since the ID's match[1], the waiting RMH thread resumes execution. The invalidation request is treated as an acknowledgment and the thread is terminated.
6. The home node receives an uncache request message while it was expecting an invalidation acknowledge. The home node can safely treat this request as an acknowledgment to its invalidation request since the processors on the local node cannot access the INC for the same address until the invalidation request is received by the local node. The transaction is completed and ownership of the cache line is transferred to the remote node by changing the directory to *Exclusive_Remote* state.

1. Once the input FSM has matched the ID of the received message against all active TSRF entries, a subsequent address match is performed if there was an ID hit. Because of the uniqueness of the ID, only one such match is possible, hence only one address comparator is needed.

3.5 Flowcontrol and Deadlocks

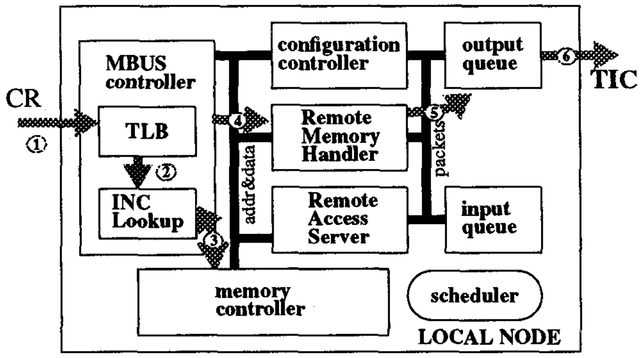
Given that there is only a finite number of TRSF entries, flow control is needed to avoid acceptance of a request without the required resources. The input and output queues provide flow control and the scheduler will not grant a request unless the destination unit is ready. However, simply refusing to accept new messages can result in deadlocks because transactions may require acknowledge messages to complete. To avoid such deadlocks, the message exchange subsystem (I/O queues and IC network), support multiple levels of priority. The protocol engines have two virtual ports with independent flow control such that secondary messages can be sent at an elevated priority while blocking new requests that have lower priority.

3.6 MC Directory Operation

As mentioned above, the directory is actually part of main memory, hence it is read on every normal memory read operation. The local bus controller checks the directory on every memory access without increasing latency. On local references that need to recall or invalidate remotely cached data, the RAS is called to perform these operations. Only the RAS may change the directory state.

The MC reserves a programmable fraction of the main memory and uses this storage for the INC, which is required to include all locally cached blocks of remote memory and which is used for an address compression scheme that minimizes bandwidth demand for control messages. Since remote references have at least 3x higher latency than local memory references, even a relatively slow INC is beneficial. The INC is programmable in size and may occupy up to 50% of the total memory. It is implemented as a 3-way set associative cache with LRU replacement policy. Although S3.mp in this configuration is still fundamentally a CC-NUMA architecture, it does have many of the properties of a cache only memory architecture (COMA). Hence S3.mp offers a variable degree of COMA behavior that can be used to fine tune the system for specific applications.

Fig. 11. Reading Data from a Remote Node



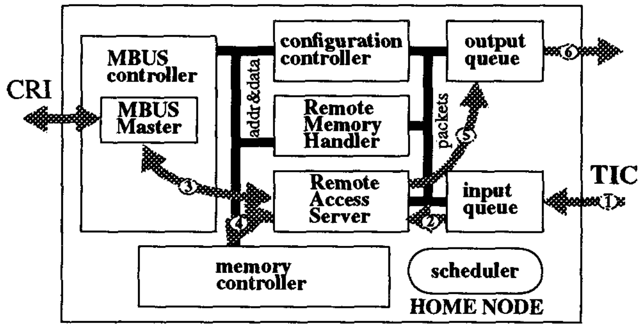
3.7 Putting it all together

This section contains a brief description on how one remote memory transaction progresses through the MC. The first agent is the local bus controller, which receives the transaction (1) in Figure 11 and identifies it as a reference to a remote part of the address space (2). The bus controller has read-only access to both the directory state and the INC state. It is capable of performing all operations that do not require assistance from a remote node, for example processing an INC hit (3).

In the case of an INC miss, the bus controller aborts the bus transaction, freeing it for other transactions[2], and forwards a copy of the INC state to the RMH (4). In this transaction, all state pertaining to a transaction is handed over to the RMH. This saves time because the RMH does not need to read the INC state. However, it also introduces a problem: the INC state may be become stale before the RMH processes the request. A mechanism similar to the Load-Linked/Store-Conditional instruction pair for synchronizing multiprocessors is used to track the validity of the INC state.

Once the new thread begins executing in the RMH, it will queue a request message (5) according to the CC-protocol and wait for a reply.

Fig. 12. Servicing Remote Read Request



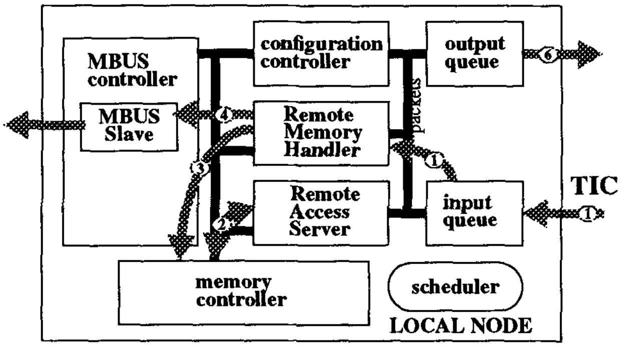
When the request packet is received by the home node (Figure 12), the following sequence of events happen:

1. The request packet is removed from the interconnect by the input queue
2. The input queue decodes the type of the packet to determine the destination unit and sends the packet to the RAS
3. The RAS requests a snoop cycle on the home Mbus. If one of the home caches have an exclusive copy of this cache line, data is read from this cache. This snoop action is necessary since the current S3.mp protocols do not distinguish between the resident directory state and states where a cache line is shared or owned by one of the CPUs at the home node.
4. If the snoop cycle has failed, the RAS reads the cache line and the associated directory information from the main memory.

2. The Mbus is a tenured bus, but allows a relinquish&retry reply, which is an approximation to a split transaction protocol, which would have superior performance.

5. If the directory is valid (i.e. the directory is not in *Exclusive_Remote* state), an acknowledge packet including the data is constructed and sent to the output queue. The RAS updates the directory information in the main memory if necessary. Assuming that no other node had a copy of the cache line involved in this transaction, directory state will be changed from *Resident* to *Shared_Remote* and the directory pointer will be updated to point to the local node.
6. The acknowledge packet including the data is sent back to the local node through the interconnect.

Fig. 13. Completion of a Remote Read



After the acknowledge packet is received by the remote node (Figure 13):

1. The acknowledge packet is removed from the interconnect by the input queue and sent to the RMH. This packet wakes up the sleeping RMH process which had initiated the request for this transaction.
2. The RMH reads the INC tags corresponding to the address of the cache line from the memory and allocates an INC entry. If there are no free INC entries available, RMH victimizes an existing INC entry in LRU fashion.
3. Data from the packet is written to the INC and INC tags are updated.
4. The MBUS controller is acknowledged to complete the transaction.

3.8 Cache Coherency Protocols

S3.mp was initially designed exclusively to support a CC-NUMA protocol and a large internode cache that resides in main memory and that could be changed in size when the system is initially turned on (static configurability). Advances in the ASIC technology during the design of the MC made it possible to replace the mask-programmed ROMS in the RAS and RMH with writable microcode SRAMS.

Discussions with Ashley Saulsbury from the Swedish Institute of Computer Science (SICS) showed that his ideas on how to implement a Simple-COMA [16] system are applicable to the S3.mp nodes with only minor extensions to the protocol engines. The most significant modification concerns the INC access logic, which was modified to support a data-less mode that only stores the tag and state information. This turns the INC into a reverse translation table.

3.9 Evaluation and Comparison

Alewife and S3.mp use roughly comparable technology (LSI Logic ASICs) for the PE, both of which can be expected to perform better than the implementation with standard components that was used in DASH. Unlike Alewife and DASH, S3.mp uses microcoded engines that offer programmability without slowing the design. A dual RISC core implementation was considered for S3.mp, which would have reduced the design time and increased versatility. This option requires roughly the same amount of chip area (RISC core + SRAM), but would need about 10x more cycles to process the cc-protocol. For a comparable design, the RISC core needs to be augmented by a larger register files to support rapid context switching and by a packet handler to assemble and decode messages efficiently and that could utilize the co-processor interface of the RISC core.

TABLE 1 : Protocol Engine Performance

	S3.mp	Alewife	DASH
Client #of gates	28 K	13 K	45 K
Server #of gates	28 K	16 K	23.5 K
Clock frequency	66 Mhz	30 Mhz	33 Mhz
Client #of concurrent operations	4	>= 3 (SW assisted)	1
Server #of concurrent operations	4	>= 3 (SW assisted)	1
Client #cycles for simple read	6+3	~8	10+11
Server #of cycles for simple read	8	~8	11

The S3.mp protocol engines achieve very good performance in terms of low flow through latency and high throughput with a relatively modest amount of logic. This opens the possibility that these PEs can be integrated onto the processor, which is one of the project goals. At the same time, programmability allows to adopt advances in cache coherency protocol designs.

4 Summary

Hardware accelerators to process cache coherency protocols are critical component of scalable shared memory multiprocessors. While there is a wide range of implementation choices for such protocol engines, it was shown that adding programmability by either microcoding or the use of embedded RISC cores are viable design options. It is advantageous to provide multiple context so that latency hiding techniques may use sever concurrent memory transaction to deal with high network latencies.

The S3.mp project opted to use microcoding to build a relatively small PE that is balanced with the other components of the memory controller chip.

5 References

[1] Nowatzyk, A., Aybay, G., Browne, M., Kelly, E., Parkin, M., Radke, W., Vishin, S., *"S3.mp: Current Status and Future Directions"*, Workshop on Shared Memory Multiprocessors, ISCA'94, Chicago (to be released as a Technical Report of the University of Southern California, Los Angeles)

[2] Agarwal, A.,Kubiatowicz J., Kranz, D., Lim, B., Yeung, D., D'Souza, G., Parkin, M. *"Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors"*. IEEE Micro, June 1993, pages 48-61.

[3] Lenoski, D. *"The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor"*. PhD Dissertation, Stanford University, December 1991.

[4] Hagersten, E., Landin, A., and Haridi, S. *"DDM - A Cache-Only Memory Architecture"*. IEEE Computer 25,9 (September 1992), 44-54.

[5] Bisiani, R., and Ravishankar, M. K. *"Design and Implementation of The Plus Prototype"*. Technical Report, Carnegie Mellon University, August 1990.

[6] Li, K., *"Shared Virtual Memory on Loosely Coupled Multiprocessors"*. PhD Dissertation, Yale University, September 1986.

[7] IEEE Std 1596-1992, *"Scalable Coherent Interface"*. Institute of Electrical and Electronics Engineers, Inc., Service Center, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331

[8] Thapar, M., Delagi, B., and Flynn, M., *"Linked List Cache Coherence for Scalable Shared Memory Multiprocessors"*, Proceedings of the 1993 International Conference on Parallel Processing, pages 34-43.

[9] Nowatzyk, A., *"Communications Architecture for Multiprocessor Networks"*. PhD Dissertation, Carnegie Mellon University, December 1989.

[10]Kubiatowicz, J., *"The Alewife-1000 CMMU: Addressing the Multiprocessor Communications Gap"*, HotChip Symposium 1994, Stanford California

[11]Dubois, M., Skeppstedt, J., Ricciulli, L., Ramamurthy, K., Stenstrom, P., *"The Detection and Elimination of Useless Misses in Multiprocessors"*, ISCA'93, San Diego, California

[12]LSI Logic Product Information on R4000 embedded controller RISC core, LSI Logic, 1551 McCarthy Blvd., Milpitas California 95035.

[13]Nowatzyk, A., Parkin, M., *"The S3.mp Interconnect System and TIC Chip"*, HotInterconnects'93, Stanford CA, Aug. 1993

[14]Dill, D., Drexler, D., Hu, A., Yang, C. *"Protocol Verification as a Hardware Design Aid"*. 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors, (October 1992), 522-52

[15]McMillan, K., *"Symbolic Model Checking"*, 1992, Carnegie Mellon University PhD Thesis, Pittsburgh, PA 15213

[16]Hagersten, E., Saulsbury, A., Landin, A., *"Simple COMA Node Implementations"*, 27th Hawaii International Conference on System Sciences, 1994

[17]Kendall Square Research, *Technical Summary*. 1992. 170 Tracer Lane, Waltham, MA 02154-1379

[18]Cox, A., Fowler, R., *"Adaptive Cache Coherency for Detecting Migratory Shared Data"*, ISCA'93, San Diego