Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study

Fong Pong, Andreas Nowatzyk*, Gunes Aybay* and Michel Dubois

Department of EE-Systems
University of Southern California
Los Angeles, CA 90089-2562

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

Abstract. This paper presents the results for the verification of the S3.mp cache coherence protocol. The S3.mp protocol uses a distributed directory with limited number of pointers and hardware supported overflow handling that keeps processing nodes sharing a data block in a singly linked list. The complexity of the protocol is high and its validation is challenging because of the distributed algorithm used to maintain the linked lists and the non-FIFO network. We found several design errors, including an error which only appears in verification models of more than three processing nodes, which is very unlikely to be detected by intensive simulations. We believe that methods described in this paper are applicable to the verification of other linked list based protocols such as the IEEE Scalable Coherent Interface.

1 Introduction

S3.mp (*Sun's Scalable Shared memory MultiProcessor*) is a research project which implements a distributed cache-coherent shared-memory system [12]. In S3.mp, cache coherence is supported by a distributed directory-based protocol with a small, fixed number of pointers and a hardware supported overflow mechanism which keeps processing nodes sharing a data block in singly linked lists. Cache coherence protocols that use linked lists have been proposed by Thapar [18] and are also used in the Scalable Coherent Interface (SCI) protocol [7].

To verify the S3.mp protocol is very difficult because the linked lists are maintained by a distributed algorithm. The addition and deletion of nodes from the linked list reorganize the list. In addition to the complexity of maintaining linked lists, the S3.mp protocol behavior is unpredictable because the non-FIFO (First-In-First-Out) interconnect network does not preserve the order in which messages are delivered between nodes.

We have applied the Stanford Murφ [4] tool and a specialized method based on a Symbolic State Model (SSM) [13] to establish the correctness of the S3.mp cache coherence protocol. During the verification, several design errors were discovered. We will describe two subtle errors which were found in the validation of S3.mp cache protocols. The first error violates *Store Atomicity* [3], which is a property of all SPARC memory models [17] prohibiting several processors from observing an inconsistent order of store operations. The loss of consistency occurs when the protocol allows more than one dirty cache line or the coexistence of shared and dirty copies. This condition, however, is too narrow to cover all possible protocol errors. The values of all data copies must also be tracked. The second error was detected in cases with more than three processors but was not detected in intensive simulation runs. The result indicates that there is a need for methods which can deal with protocol models with a large number of processors. Verification and simulation runs on small models are not always sufficient to establish the correctness of complex protocols.

2 Overview of the S3.mp System and Cache Coherence Protocol

The S3.mp implements a *CC-NUMA* (*Cache-Coherent Non-uniform Memory Access*) multiprocessor system (Fig. 1). A specialized interconnect controller is added to the memory subsystem of a standard workstation [11]. The S3.mp is then built by interconnecting clusters of

workstations to form multiprocessor workgroups which efficiently share memory, processors and I/O devices [12].

Each node may have several processors with private caches which are kept coherent via a snooping protocol supported on the *Mbus*. This is the system bus in the Sparcstation-10 and Sparcstation-20 series. Each processing node maintains a fraction of a globally shared address space. When a processor accesses a remote memory location, the memory controller translates the bus transaction into a message that is sent across the network to the remote memory controller.

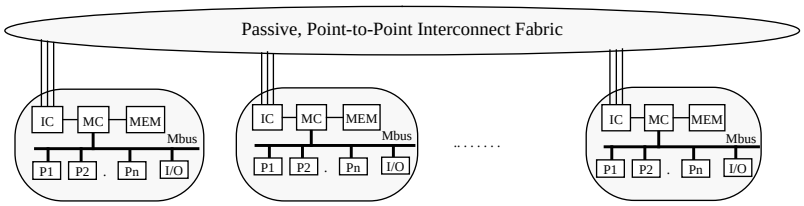

Fig. 1. Overview of the S3.mp System.

2.1 Distributed Directory-based Protocol

In the S3.mp, data consistency is maintained using distributed directories with a limited number of pointers. Pointer overflows are handled by a protocol with a directory organized as singly linked lists. Every memory block is associated with a *home* node which is the node where the physical memory page containing the memory block resides [9]. The home node serializes concurrent requests to the block and maintains a pointer to the head node of the linked list.

Part of the physical memory at each node is allocated as a large *InterNode Cache* (*INC*). A copy of every cache block retrieved from remote nodes is loaded into the INC. This copy is maintained in the INC for as long as there is a copy in one of the processor caches at that node. This *inclusion* property is exploited by the directory protocol. In addition to providing support for the protocol, the INC cuts the network traffic. The size of the INC is programmable up to half of the main memory at a node to provide support for data migration. As a result, capacity misses to shared remote data due to the small size of processor caches are served from the much larger INC (up to 32 Mbytes).

We will not model the processor caches. This simplification is justified by two facts. First, the Mbus snooping protocol supporting the first-level coherent caches and the mechanisms for maintaining the inclusion property are well-understood -- whenever the INC receives a write-invalidation from the network, an invalidation is broadcast on the local Mbus to remove all potential copies in the first-level processor caches. Second, verifying the coherence protocol between the second-level INCs is our primary goal.

2.1.1 Directory and InterNode Cache States

We refer to the “state of the block in the INC” as the “cache state” and the “state of the block in the memory directory” as the “directory state”. For every memory block, the home directory can be in one of four stable states: *Resident* (RES; the block only resides in the home node. The first-level processor caches are looked up and/or invalidated by incoming requests from remote nodes), *Shared_Remote* (SR; the memory copy is up to date and consistent with remotely cached read-only copies. The directory contains a pointer to the head node of the linked list formed by nodes sharing the block), and *Exclusive_Remote* (ER; the memory copy is stale and the block is *owned* and modified by one remote node. The home directory contains a pointer to the remote owner of the block).

Every cache block in the INC can be in one of three stable states: *Invalid* (Inv; the cache does not have a valid copy), *Read-Only* (RO; the cache has a clean copy which is potentially

shared with other caches), and *Read-Write* (RW; the cache *owns* an exclusive copy of the block). INC tags also provide additional storage space for protocol information; they are used specifically to store '*pointers*' for the linked lists used by the protocol.

2.1.2 Cache Algorithm

The S3.mp protocol is an ownership-based, write-invalidate protocol. A read hit or a write hit to an owner copy do not cause coherence transactions. A cache miss is always serviced by the home node if there is no owner copy in the system. When an owner copy is present, the owner node is requested to provide the data. In the S3.mp, the requesting node always becomes the new head of the linked list.

When a write miss occurs, the linked list is cleared by an *ordinary* invalidation process. Invalidation is initiated by the home node which sends an invalidation to the head node of the linked list. The invalidation then propagates through the list and the last node acknowledges the home node to complete the invalidation process.

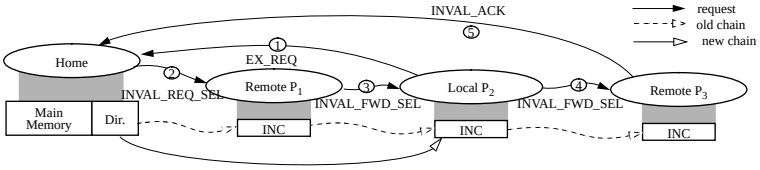

Fig. 2. Procure Ownership.

In the case of requesting for ownership, the home node initiates a *selective* invalidation cycle to remove all other cached copies but the requesting node copy. The transaction is shown in Fig. 2. The local node P_2 sends a request for ownership (EX_REQ) of the block to the home node. When the home node receives the request, it issues a selective invalidation request (INVAL_REQ_SEL) to the head node P_1 of the linked list. This invalidation message carries a null ID, which is not associated with any node. When P_1 receives the invalidation, it invalidates its copy and forwards the invalidation request (INVAL_FWD_SEL) to P_2. When P_2 receives the invalidation, it checks the node ID carried in the message. If it finds a null ID, it replaces the null ID with its own ID before propagating the message. In this case, P_2 will become the new exclusive owner of the block. If the message carries any other ID, P_2 will invalidate its copy and forward the invalidation to the next node in the list. Therefore, if more than one processor tries to write, the first one in the list will be granted ownership. This mechanism deals with the possibility that multiple processors in the list may issue EX_REQ messages concurrently, hence the number of outstanding EX_REQ messages are not known to the home node. After processing the first EX_REQ message, subsequent EX_REQ messages are obsolete and cause only harmless invalidations once they are processed by the home node. In the example, when P_3 receives the invalidation, it invalidates itself and acknowledges the home node.

Replacing a block in the state RW causes a write-back to the home node. If the victimized block is in state RO, a *conditional* invalidation message is used to remove the node containing this block from the sharing list. As shown in Fig. 3, the local node P_2 first sends an *uncache* request (UNC_REQ) to the home node, which subsequently sends a conditional invalidation (INVAL_REQ_COND) to the head node P_1. This invalidation message contains a *null* ID. When P_1 receives the invalidation, it updates its next pointer with the received ID. P_1 then forwards P_2 an invalidation message (INVAL_FWD_COND) which includes its own ID. When P_2 receives the invalidation, it invalidates itself and propagates to P_3 the same message received from P_1. When P_3 receives the invalidation message, it changes its pointer to the ID of P_1. P_3 acknowledges the home, which promotes P_3 to the new head node of the linked list. The conditional invalidation is similar to the selective invalidation, except that it removes only the cache blocks which are wait-

ing to be discarded in one pass through the sharing list and reverses the linked list. Linked list reversal solves the problem of unnecessarily invalidating part of the sharing list while invalidating a single node, which has been considered a disadvantage for protocols utilizing single linked lists [18]. The home node can tell from the acknowledgment when the last outstanding copy is discarded.

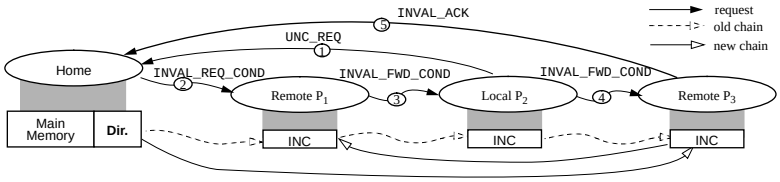

Fig. 3. Replace a Block in Shared State.

A *Recall* operation occurs when a processor at the home node has an access miss and there exists a remote owner. The current owner is asked to relinquish its copy and to return the new data to the home memory. The S3.mp supports a special operation called *Write-All* which allows a node to update an entire 32-byte memory block and forces *all* cached blocks to be invalidated using ordinary write-invalidation process. This operation accelerates bulk data moves or I/O operations.

3 Verification of the Protocol

3.1 Protocol Model

In the formal verification model several abstractions are made in order to construct a model with manageable complexity. Only one block is modeled and no information is kept on transactions to other memory locations [10,13]. We assume that replacements can take place at any time and we model them as processor accesses.

With respect to a single memory block, we abstract the architecture by the model of Fig. 4. The model consists of a home node and multiple processor-cache pairs. The first-level cache is explicitly modeled for the home processor in order to test the *Recall* operation. For remote processors, the first-level caches are not modeled as explained in section 2.1. Each processing node has only one processor which is associated with one *message sending channel* (*CH!*) and one *message receiving channel* (*CH?*) to model the message flow between caches and main memory. All message channels are characterized as non-FIFO buffers to simulate the network. Furthermore, we assume that messages are never lost.

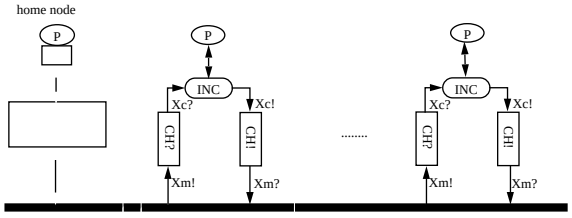

Fig. 4. Verification Model.

Another important abstraction made in the model is to allow only one *ghost* or *unresolved* message (per message type) in any message channel [2,14]. Ghost requests are caused by messages whose meaning is lost between their transmission and their reception because some other message has changed the state of the block. In the cases of uncache and ownership requests, there is a slight possibility in the S3.mp protocol to leave ghost requests in the network. For

example, consider the case where processors P_1 and P_2 initially share the block and issue requests for ownership of the block concurrently. Suppose that P_1's request wins the race so that P_2's copy is invalidated. P_2 is therefore forced to re-initiate a request for an '*exclusive copy*' of the block, leaving an out-of-date ghost request for ownership of the block floating in the system. The other source of ghost messages occurs when processors P_1 and P_2 send requests to discard their copies at the same time. If P_1's request arrives at the home node first, both P_1's and P_2's copies are discarded (a conditional invalidation removes *all* entries scheduled to be discarded), while P_2's request remains pending.

Ghost requests for ownership and for block replacement are designed to be harmless to the correctness of the S3.mp protocol. In a correct design, unresolved requests for ownership merely purge the linked list and unresolved uncache requests only reverse the linked lists. However, obsolete requests waste network bandwidth and slow down the system, as well as exacerbating the complexity of the verification. On the other hand, protocols that are free of ghost-messages require more message exchanges and limit the concurrency, which results in lower performance for the common case. Due to the non-FIFO interconnect network, the number of ghost messages is not bounded, which prevents convergence of the verification process. Limiting ghost messages in any given message channel to one per message type does not compromise the validity, but greatly simplify the verification. Because the network is non-FIFO and a ghost message can be indefinitely delayed in the model, processors may receive ghost messages in arbitrary states. An alternative approach to model ghost messages consists in using *pseudo processors* which only act only as *event generators* injecting ghost messages into the verification model unconditionally. In section 4.4, we will explain how the SSM method automatically supports this alternative.

3.2 Modeling of Data Consistency

The S3.mp supports three memory consistency models: *Total Store Order*, *Partial Store Order*, and *Relaxed Memory Order* [15]. The strict *Sequential Consistency* model [8] is not supported. Rigorously speaking, the S3.mp cache protocol only guarantees the property of consistency: the value of a load access *La* to memory location *a* is the value written by the most recent store *Sa* that was globally performed or by the most recently initiated store by the same processor. The "globally performed order" among stores is defined by the program order and the execution order of synchronization accesses [1]. The programmer can specify the order constraints of memory accesses either implicitly by the choice of memory model or explicitly by using *memory-barrier* (`MEMBAR`) instructions [17].

To verify the property of consistency, we need to keep track of values of all data copies. Extending the abstraction in [13], the values of any cached copy can be in one of five states: `NoData` (the cache has no valid copy), `GlobalFresh` (the cache has an up-to-date copy; value is defined by the latest globally performed write), `GlobalFresh_Hold` (the cache writes a new value which is not yet visible to other processors), `LocalFresh` (a valid copy for read; a new value defined by another processor is not yet visible to the cache), and `Obsolete` (the cache has an out-of-date copy). The memory copy can also be one of the above states.

Consider the case of a write miss handled by the S3.mp protocol in Fig. 5. Initially, processors P_1, P_2, and P_3 share the block. Their copies are globally fresh as well as the memory copy. When processor P_0 experiences a write miss, it sends a request to the home node which provides a data copy to P_0 and initiates an invalidation cycle to purge the linked list. As soon as P_0 receives the data, it performs the pending write and immediately defines a new value, whereas P_2 and P_3 have not yet received the invalidation. As a result, inconsistency may exist among data copies cached by P_0, P_2, and P_3, nevertheless, the system still operates correctly in conformance with the SPARC consistency models by "hiding" the value stored by P_0 from other nodes until

the invalidation cycle is complete. In our abstraction, when P_0 performs the write, it defines a new value (`GlobalFresh_Hold`), but this value is not yet visible to other processors. Processors P_2 and P_3 are still allowed to read their locally fresh (`LocalFresh`) copies. Thus, the read accesses by P_2 and P_3 appear ahead of the write access by P_0 in a legal order of execution. As shown in Fig. 5, the `MEMBAR` instruction orders the store access and memory accesses after the `MEMBAR` instruction. The `MEMBAR` instruction stalls processor P_0 until the completions of preceding accesses. When finally receiving an acknowledgment from home, P_0 holds the most recent value, and all other copies in the system (including the home memory copy) become obsolete.

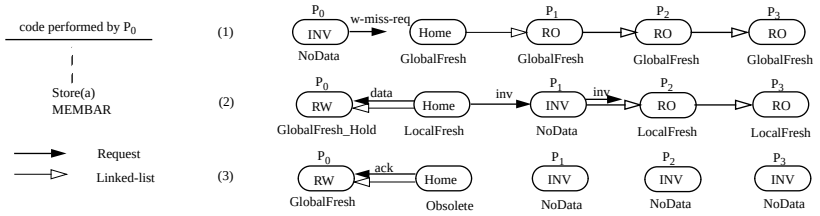

Fig. 5. Trace Values of Data Copies.

From this abstraction, we verify that the S3.mp protocol never allows a processor to read a memory location with an obsolete value.

3.3 Verification Methodologies

3.3.1 State Enumeration

First, we use the Stanford Murφ system [4] to verify the protocol. The Murφ implements a state enumeration method which explores all possible system states. We start the expansion process with an initial state in which all processors are in the Invalid (`INV`) state, and the home node is in the Reside (`RES`) state. All possible transitions are exercised, leading to a number of new states. The same process is applied repeatedly for every new state until no new states are generated. (Some transitions may lead back to states which have already been generated.)

To deal with the large state space, the Murφ exploits the symmetry of the system by grouping together states whose representations are permutations of each other [6]. According to the symmetry as shown in Fig. 4, the contexts of processors (represented as base machines) can be swapped without affecting the behavior of the system. Given a protocol model with *n* processors, the maximum reduction is *n*!. The Murφ also incorporates state encoding to reduce memory usage and hash tables to speed up the search and comparison operations.

We have successfully applied the Murφ to verify completely a system model including one home node and two remote nodes. Many design errors were found quickly in this small model. The trace generation facility provided in the Murφ has proven to be very useful. However, this model is fairly small and a moderately larger model is needed to obtain more reliable verification results.

3.3.2 Symbolic State Model (SSM)

Based on the *symmetry* and *homogeneity* of cache protocols (e.g., all bases machines in Fig. 4 are symmetrically and functionally identical), the SSM method differs from other state enumeration methods in the ways of representing and pruning global states. Since, in all existing protocols, data consistency is enforced by either broadcasting writes to all copies or by invalidating the copies in all other caches, the exact number of data copies in a shared state is irrelevant to protocol correctness. What is critical is whether there exists 0, 1, or multiple copies in a particular state (such as more than one `RW` copy). As a result, the SSM maps system states to

more abstract states which do not keep track of the exact number of copies. The following repetition constructors are used to represent global states (for a detailed justification, see [13-14]).

Definition 1. (Repetition Constructors)

1. **Null** *(0) indicates zero instance.*
2. **Singleton** (1) *indicates one and only one instance.*
3. **Plus** (+) *indicates one or multiple instances.*
4. **Star** (*) *indicates zero, one or multiple instances.*

In a system with an unspecified number of caches, we group base machines (Fig. 4) in the same state into a *state classes* and specify the number of caches in the class by one of the repetition constructors. For example, we can represent all the global states such that "one or multiple caches are in the Invalid state, and zero, one or multiple caches are in the Read-Only state" by ($\mathtt{Inv}^{+}$, $\mathtt{RO}^{*}$). This representation includes a large set of states, which would otherwise be listed explicitly in a state enumeration method.

Unfortunately, this abstract representation is not sufficient to represent linked lists. Therefore, we use a *hybrid* model in which processors in the linked lists are explicitly maintained and other processors are represented by the abstraction. The formal definition of a global state (called composite state) is defined as follows.

Definition 2. (Composite State) *With respect to a given block, a composite state represents the state of the protocol machine for a system with an arbitrary number of caches. It is constructed over an explicit linked list which contains the states of processors sharing the block. Processors that are not kept in the linked list are grouped and represented by the abstraction of the symbolic state model. Specifically, a composite state has the form*

$$\left(q_{\mathcal{MM}} \rightarrow {}_{R_1}C1_{S_1} \rightarrow {}_{R_2}C2_{S_2} \rightarrow \ldots \rightarrow {}_{R_k}C_{k_{S_k}}, \left({}_{R_{k+1}}C_{k+1}{}_{S_{k+1}}^{r_{k+1}}, \ldots, {}_{R_n}C_{n_{S_n}}^{r_n} \right) \right),$$

where C_i is the cache state, R_i and S_i represent the states of the message receiving and *sending channel respectively, $r_{i:k+1..n} \in [0, 1, +, *]$ and $q_{\mathcal{MM}}$ is the state of the home memory. The arrows represent the pointers in the linked list.*

Repetition constructors are ordered by the possible states they specify. The resulting order is $1 < + < *$; the null instance is ordered with respect to *, i.e., $0 < *$. These two orders lead to the definition of *state containment*.

Definition 3. (Containment) *Composite state S_2 contains composite state S_1, or $S_1 \subseteq S_2$, if*

1. S_1 and S_2 have exactly identical linked list,
2. $\forall {}_{R}C_{S}^{r_1} \in S_1 \quad \exists {}_{R}C_{S}^{r_2} \in S_2$ such that ${}_{R}C_{S}^{r_1} \leq {}_{R}C_{S}^{r_2}$ i.e. $r_1 \leq r_2$ and
3. $q_{\mathcal{MM}1} = q_{\mathcal{MM}2}$

where $r_1, r_2 \in [0, 1, +, *]$.

This definition is an extension of our previous work which deals with snooping and central directory-based protocols [13,14]. We have added the requirement that two global states must have exactly identical linked lists in order to model the distributed directory structure of the S3.mp. As shown before, the SSM abstraction leads to a *monotonous containment relation* such that, if $S_1 \subseteq S_2$, then the family of states represented by S_2 is a superset of the family of states represented by S_1. Augmenting the abstraction with the explicit linked list does not affect this property of monotonicity. Because S_1 and S_2 must have the same linked lists, states evolving from activities of processors in the linked lists of S_1 and S_2 are identical.

Theorem 1. (Monotonicity) *If $S_1 \subseteq S_2$, then for every $\overline{S}_1$ reachable from S_1 there exists $\overline{S}_2$ reachable from S_2 such that $\overline{S}_1 \subseteq \overline{S}_2$.*

As the expansion process progresses, the SSM only keeps composite states which are not contained by any other state. At the end of the expansion process, the state space is partitioned into several families of states (which may be overlapping) represented by *essential* states.

Definition 4. (Essential State) *Composite state S is essential if and only if there does not exist a composite state $\overline{S}$ such that $S \subseteq \overline{S}$.*

Based on the monotonicity property of the SSM abstraction, we run the verification incrementally. The verification is started by limiting the maximum number of processors allowed to actively share the block to two (the length of the linked list). This number is then increased by one in each consecutive run. The set of essential states reported at the end of the current run is used as the set of initial states for the next run in order to quickly accumulate the state information. As the model size grows, the state space expands. We therefore run the verification until we run out of memory to store the state information.

4 Verification Results

Because of the high complexity of the S3.mp protocol, we have used a technique called *case-splitting* to avoid dealing with the entire protocol at one time. We have identified and isolated events that can be verified separately. For instance, the reason for modeling the first-level cache of the home processor is to test the *Recall* operation. The *Recall* operation occurs when an access misses in the cache of the home node while an exclusive copy exists in a remote node. To test *Recall* operation, we first run a small model of three processors with the full set of operations; then the home processor cache and the accesses from the home processor are removed in larger models. This simplification is justified because the small model covers the case in which a remote owner exists; regardless of the model size, only one owner is allowed in the system, and concurrent transactions are serialized by the home directory. When the home processor initiates a *Recall* transaction, there must exist only one owner copy and this transaction is guaranteed to complete before the next coherence transaction is executed by the home directory. This makes it possible for coherence transactions initiated by home processors to be checked in small models and eliminated from larger model so that the doubling of the size of the state space is avoided.

The *Write-All* access can be dropped in larger models for the same reason, i.e. a *Write-All* access essentially results in an ordinary write-invalidation which clears the sharing list. Since the home directory is locked during the propagation of invalidations, the invalidation caused by a *Write-All* request is equivalent to a normal write-invalidation operation. Therefore, in larger models, we do not need to repetitively model the *Write-All* operation.

4.1 Efficiency of the SSM Method

All the verifications were ran on a Sun4/690 system with 500Mb of physical memory. The performance of the verifications for all model sizes from 2 to 5 processing nodes is listed in Table 1. As shown in the table, the verification process is memory-intensive, partly because the SSM state information is not encoded in the SSM and, since the state classes are dynamically created and discarded, a large amount of memory is needed to maintain the complete representation of a global state in the SSM. The verification time increases rapidly with the number of nodes in the sharing list. Because we use the results generated in a current run as the set of starting states for the next run, the verification time accumulates. Most of the time is wasted for generating, searching and comparing states which have been produced. In fact, the efficiency of the SSM method depends on how fast the final set of essential states are generated. Unfortunately we do not have a very good heuristic to this end. The results listed in Table 1 are obtained by

using a a *depth-first* expansion scheme. Namely, the next expansion path is explored only when the current expansion path is exhausted.

TABLE 1 : The Verification Results by Using the SSM Method.

Sharing Degree	Verification Time	Memory (Mb)
2	11.33	0.03
3	7,419.57	28.32
4	326,413.80	352.46
5	402,638.63[a]	500

a. This run is not fully completed.

As expected, the hybrid SSM method does not generally avoid the state explosion problem because processors sharing a memory block are explicitly tracked.

4.2 Data Inconsistency Caused by Stale Write-Backs

Although requests in the S3.mp protocol are always acknowledged, messages are *context sensitive*. Two nodes may concurrently send two independent request messages for the same memory location, such that each request is regarded as the acknowledgment by the recipient. For example, the home node may issue a *Recall* request while at the same time the remote owner sends a *write-back* request to the home node. No additional messages are generated to complete this exchange. The advantage of context sensitive semantics is lower latency and less traffic in some cases, but the design and verification of the protocol are more complex.

An error of data inconsistency caused by a stale *write-back* is shown in Fig. 6. Initially cache P_0 has a dirty copy of the block, and performs a write-back (`WB_REQ`) to the home node. In order to guarantee that the memory receives the block safely, cache P_0 keeps a valid copy of the block until it receives an acknowledgment. Meanwhile, cache P_1 sends a request (`CRE_REQ`) for an exclusive copy to the home node. Cache P_0 then processes the *data-forward-request* (`D_FWD_E_REQ`) from home, sends the data to P_1, and continues, ignoring the acknowledgment from the home node. P_1 then performs its write and victimizes the block after some unrelated misses. As shown in Fig. 6, a race condition exists between the two write-back requests. If the write-back from P_1 wins the race, the stale write-back from P_0 overwrites the values updated by P_1. Note that, in this example, all state transitions are permissible.

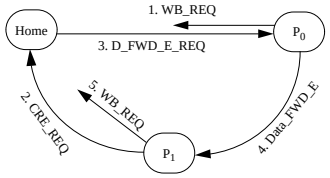

Fig. 6. Data Inconsistency Caused by Stale Write-Back.

If we do not keep track of the values of the data copies, this error goes undetected because the system correctly allows only one exclusive owner copy at any time. In our abstraction, when P_1 performs its write, it defines a globally fresh copy as compared to the obsolete copy carried by P_0's write-back message. This error is easily detected.

4.3 Data Inconsistency Found in a Larger Model

A fairly complex error which only appears when the verification model includes more than three processors was found and the sequence of events leading to this error shown in Fig. 7 is as follows:

1. Initially processor P_0 writes back its exclusive copy (`WB_REQ`) to the home node. Also, processor P_1 and P_2 request shared copies (`CR_REQ`). P_0 keeps a valid copy of the block until it receives an acknowledgment from the home node.
2. The home node receives the request for a shared copy from P_1. The home node establishes a new link pointing to P_1 and issues a data-forward-request (`D_FWD_REQ`) to the current owner P_0. Processor P_1 is considered to be the new head of the list.
3. The data-forward-request from the home node, is interpreted by P_0 as an acknowledgment. P_0 forwards its copy to P_1 (by `DATA_FWD_E`) and invalidates its copy. When the home node receives the write-back from P_0, it updates the memory copy and releases the directory entry locked by the transaction from P_1. The directory changes to the stable state `SR`.
4. Home receives the request for shared copy from P_2. The home memory copy is supplied to P_2, which becomes the new head of the list. P_1 is now the second node in the list. P_1, meanwhile, is still waiting for the copy forwarded by P_0.
5. Home receives a harmless ghost request for replacement of a read-only copy (`UNC_REQ`, section 3.1). This request merely reverses the linked list in a correct protocol. When P_1 receives the conditional invalidation passed by P_2, P_1 needs to keep its pointer to P_2 to set up the list correctly.
6. When P_1 receives the forwarded exclusive data from P_0, P_1 nullifies its pointer since it thinks that it has the only copy. This causes the chain to be broken and P_2 is left off the list. When other processors write to the block, P_2's copy is not properly invalidated.

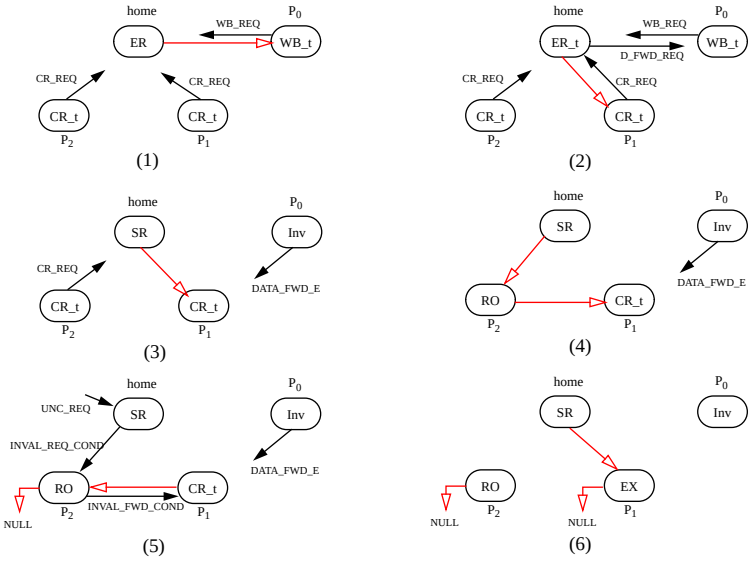

Fig. 7. Data Inconsistency due to a Broken Chain.

This error only occurs when at least four processors are involved. It was not detected in the model with two or three processors.

4.4 Generation of Events in the SSM

It should be noted that the state enumeration method and the hybrid SSM abstraction do not deal with the extreme case that the sharing list is infinitely long. To some extent, the SSM method loses some of its advantages over other approaches of verifying cache coherence proto-

cols for arbitrary number of processors [13]. Nevertheless, we still exploit the other advantages provided by the symbolic state model in spite of this constraint.

Given a limited amount of memory and computation time, the advantage of using the SSM model versus a state enumeration method is that more test sequences are generated than in the state enumeration method which has a fixed number of processors. The state enumeration method only explores the interactions between the set of processors included in the model. The hybrid SSM model generates additional sequences of memory accesses initiated by processors out of the linked lists. When a processor is retired from the linked list, it moves to the part of the state representation abstracted by repetition constructors and interacts with the rest of the system for the rest of the expansion process.

Consider the scenario shown in Fig. 8. In S_0, processors P_0, P_1 and P_2 initially share the block and issue requests (`EX_REQ`) for ownership of the block at the same time. Other processors have no copies and are grouped in the abstracted part of the representation. In the example, some processors are in the `INV` state and some processors are in the `CRE_t` transient state with issued requests for the owner copy of the block in their sending channels. In accordance to the protocol, when P_2's request wins the race, it obtains an exclusive copy. P_0 and P_1 are retired from the sharing list and are forced to regenerate requests (`CRE_REQ`) for owner copies. The states of P_0 and P_1 are combined into a new class in the abstraction, this new class evolves with the state expansion and never vanishes as shown in the transition from S_2 to S_3. As a result, processors grouped in the abstracted part of the representation act as *event generators* which constantly inject new memory transactions into the system. The original requests (`EX_REQ`) issued by P_0 and P_1 are ghost messages in S_2 and S_3.

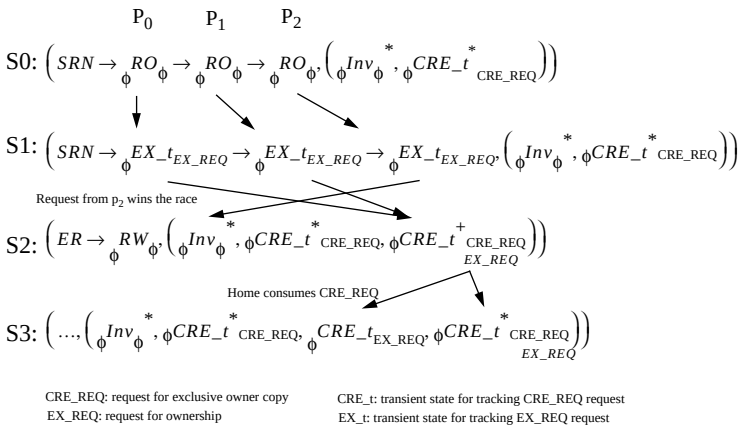

Fig. 8. State Expansion and Event Generation in the Hybrid SSM Method.

The SSM method can be simplified even more by dropping the plus (+) constructor. The plus constructor was introduced to track the existence of a data copy. For the S3.mp protocol, we do not have to be concerned about the exact number of processors in a particular state class since deterministic information such as the processors sharing the same memory block is maintained in the linked lists. The plus constructor can therefore be dropped without affecting the validity of the SSM method.

5 Conclusion

We have presented the results of verifying the S3.mp cache coherence protocol. The difficulty in verifying a distributed directory protocol is to abstract the linked lists efficiently, while

correctly preserving properties that need to be checked. Since correct maintenance of the linked lists is orthogonal to maintaining data consistency, a possible solution is to isolate the problem of verifying the integrities of the linked lists from the problem of verifying data coherence. We think that the techniques applied to the S3.mp cache coherence protocols are applicable to other linked-list based cache coherence protocols, for example the SCI [7].

We have also demonstrated how to formulate the condition of data consistency in the context of relaxed memory consistency models. The approach in this paper only verifies the property of consistency, for which the state of a single memory block must be tracked. A more difficult problem is the verification of correct memory ordering of all memory accesses according to the memory consistency model. No formal method has been conceived as of today to tackle this problem. Even if a formal framework is found, the states and the values of multiple memory locations would have to be tracked, and this requirement would no doubt exacerbate the state space explosion problem.

References

[1] Adve, S.V. and Hill, M.D., "Weak Ordering--A New Definition", *Proc. of the 17th Int'l Symposium on Computer Architecture*, May 1990, pp.2-14.

[2] Archibald, J., "The Cache Coherence Problem in Shared-Memory Multiprocessors", Ph.D Dissertation, University of Washington, Feb. 1987.

[3] Collier, W.W., Reasoning About Parallel Architectures, Prentice Hall, Englewood Cliffs, New Jersey.

[4] Dill, D.L., Drexler, A.J., Hu, A.J. and Yang, C.H., "Protocol Verification as a Hardware Design Aid", *Int'l Conf. on Computer Design: VLSI in Computers and Processors*, pp. 522-525, Oct. 1992.

[5] Holzmann, G.J., "Algorithms for Automated Protocol Verification", *AT\&T Technical Journal*, Jan./ Feb. 1990.

[6] Ip, C.N. and Dill, D.L., "Better Verification Through Symmetry", *Proc. 11th Int'l Symp. on Computer Hardwae Description Languages and Their Applications*, pp. 87-100, Apr. 1993.

[7] James et al., "Scalable Coherent Interface", *IEEE Computer*, June 90, Vol 23, No. 6, pp 71-82.

[8] Lamport, L., "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Trans. on Computers*, Vol. C-28, No.9, Sept. 1979, pp.690-691.

[9] Lenosky, D., *et al.*, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor", *Proc. of the 17th Int'l Symposium on Computer Architecture*, June 1990, pp. 148-159.

[10] McMillan, K.L. and Schwalbe, J., "Formal Verification of the Gigamax Cache Consistency Protocol", *Proc. of the ISSM Int'l Conf. on Parallel and Distributed Computing*, Oct. 1991.

[11] Nowaztyk, A. and Parkin, M., "The S3.mp Interconnection System and TIC Chip", Hot *Interconnects* 1993.

[12] Nowatzyk, A., Aybay, G., Browne, M., Kelly, E., Parkin, M., Radke, B., and Vishin, S., "The S3.mp Scalable Shared Memory: Current Status and Future Directions", *Workshop on Shared Memory Multiprocessors, ISCA*, May 1994. [also as USC Technical Report #...]

[13] Pong, F. and Dubois, M., "The Verification of Cache Coherence Protocols", *Proc. of the 5th Annual Symp. on Parallel Algorithm and Architecture*, pp.11-20, June 1993.

[14] Pong, F. and Dubois, M., "Formal Verification of Complex Coherence Protocols Using Symbolic State Models", *Technical Report CENG-94-01, University of Southern California.*

[15] Sindhu, P.S., Frailong, J-M. and Cekleov, M., "Formal Specification of Memory Models", In Dubois M. and Thakkar, S., Editors. *Scalable Shared Memory Multiprocessors*. Kluwer, Norwell, MA, 1992.

[16] Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 12-24, June 1990.

[17] The SPARC Architecture Manual, Version 9, Prentice Hall.

[18] Thapar, M. and Delagi, B., "Stanford Distributed-Directory Protocol", *IEEE Computer*, June 1990, pp. 78-80.