Verification of FLASH Cache Coherence Protocol By Aggregation of Distributed Transactions *

Seungjoon Park *David L. Dill*
Computer Systems Laboratory
Stanford University
park@turnip.stanford.edu dill@cs.stanford.edu

Abstract

To verify cache coherence protocols for distributed multiprocessor architectures, we compare a state graph of the implementation with a specification which is a state graph representing the simplified behavior. The steps in the specification correspond to atomic transactions, which are not atomic in the implementation. The method relies on an abstraction function which aggregates the implementation steps of each transaction into a single atomic transaction in the specification. The key idea in defining the abstraction function is that it must *complete* transactions which have committed but are not finished.

This approach is applied to verification of the cache coherence protocol in the FLASH multiprocessor system. We illustrate how to determine an abstraction function which reduces the protocol to an atomic specification. The protocol consisting of more than a hundred implementation steps has been reduced into six kinds of atomic transactions. Based on the reduced behavior, it is very easy to prove crucial properties of the protocol including data consistency of cached copies at the user level. Moreover, the reduced model allows us to write a simple executable memory model of the protocol. The aggregation method is also used to prove that the reduced protocol satisfies a desired memory consistency model.

1 Introduction

In shared-memory multiprocessor architectures, cache coherence protocols maintain consistency of multiple copies of cached data. The protocols control a number of readable and writable copies of each memory line for multiprocessors. Modification of one copy of a datum may require updating of other copies to maintain consistency among them. Several coherence protocols have been proposed for distributed multiprocessor architectures but few are formally verified [1, 15, 2, 10].

*This research was supported by the Advanced Research Projects Agency through NASA grant NAG-2-891.

SPAA'96, Padua, Italy
© 1996 ACM 0-89791-809-6/96/06 ..$3.50

Formal verification is desirable because there could be subtle bugs as the complexity of protocols increases. Although finite-state methods (e.g. [3, 5]) can solve many verification problems with little effort, they are basically limited to finite-state protocols. The finite-state techniques we have applied do not scale especially well for the implementation-detailed cache coherence protocols. For example, Murφ verifier can barely handle the protocols with 3 processors and 2 memory lines, using 100 megabytes of memory in the process.

Symbolic state models proposed by Pong and Dubois [14] use symbolic states which abstract away from exact number of configurations of replicated identical components by recording only whether there are zero, one, or more than zero replicated components. However, there still remains a specification problem of the protocol as in model checking: It is not easy to find a set of properties, say in temporal logic or in their notation, which completely describes the correct behavior of the protocols. Moreover, their method requires the user to write an abstract description of the protocol to be verified, which raises another verification problem that the abstract description and the actual protocol are equivalent.

Finite-state methods have been applied to non-finite-state systems in various ways, but these techniques typically require substantial pencil-and-paper reasoning to justify. Theorem-provers make sure that such manual reasoning is indeed correct, in addition to making available the full power of formal mathematics for proof, so they can routinely deal with problems that cannot yet be solved by any finite-state methods. However, the major problem with theorem proving is that considerable labor is required. Consequently, previous theorem proving approaches have not been able to verify a problem of the scale of a full multiprocessor cache coherence protocol.

We have recently developed a method for simplifying automatic proofs of cache coherence protocols and similar distributed algorithms using "aggregation" [13]. Using this method, we have formally verified the cache coherence protocol of the FLASH multiprocessor at the level of its formal description [8, 7]. The protocol consisting of more than a hundred different implementation steps has been reduced to a model with only six kinds of atomic transactions. It is much easier to prove important properties of the reduced model, such as the consistency of data at the user level, than the original protocol description. This paper overviews the general method and its application to the FLASH protocol.

Protocols for distributed systems often simulate atomic transactions in environments where atomic implementations are impossible. We believe that this observation can be

exploited to make formal verification of protocols and distributed algorithms using a theorem-prover much easier than it would otherwise be.

The method proves that an implementation state graph is consistent with an specification state graph that captures the ideal abstract behavior of the protocol, in which each transaction appears to be atomic. The method involves constructing an abstraction function which maps the distributed steps of each transaction to the atomic transaction in the specification. We call this *aggregation*, because the abstraction function reassembles the distributed transactions into atomic transactions.

This method addresses the primary difficulty with using theorem proving for verification of real systems, which is the amount of human effort required to complete a proof, by making it easier to create appropriate abstraction functions. Although our work is based on using the PVS theorem-prover from SRI International [11], the method is equally appropriate for other theorem-proving systems.

For our method to be applicable, the description must have an identifiable set of *transactions*. Each transaction must have a unique *commit step*, at which a state change first becomes visible to the specification. Usually, a commit step corresponds to the transition which first modifies one of the distributed copies, resulting in a temporarily inconsistent state. Once a part of distributed data is modified, the nodes that are concerned with the change should wait for the steps which restores the data consistency. The protocol consists of a set of rules which cooperate to maintain consistency of data in distributed nodes.

The most important idea in the method is that the abstraction function can be defined by completing transactions that have committed but not yet completed. In our experience, this guideline greatly simplifies the definition of an appropriate abstraction function.

The same idea of aggregating transactions can be applied to reverse-engineer a specification where none exists, because the specification with atomic transactions is usually consistent with the intuition of the system designer. We construct a reduced model which performs transactions atomically at their commit steps in the implementation, and does nothing at other steps. The reduced model may aggregate distributed steps for each transaction so that it can provide an illusion that the transaction takes effect instantaneously at the commit step in the implementation.

If the reduced model is not considered as a complete specification, or is not obviously correct, it can instead be regarded as a model of the protocol with an enormously reduced number of states. The amount of reduction is much more than other reduction methods used in model checking, such as partial order reduction, mainly because the reduced system is based on the only state variables relevant to the specification, without variables such as local states and communications buffers.

This method has been successfully applied to verification of the FLASH protocol. In this paper, we illustrate how the protocol is reduced to an atomic model by an abstraction function. Based on the reduced atomic behavior, it is very easy to reason about the protocol: checking safety properties and data consistency of cached copies. Moreover, the reduced model allows us to write a simple executable memory model supported by the protocol. The executable description is automatically able to generate all the possible outcomes of test programs for each mode that the FLASH protocol supports. The aggregation method can also be used to verify that the reduced protocol satisfies a desired memory consistency model. The detailed proofs are confirmed by a theorem prover and some techniques to simplify the proof are presented.

This paper is organized as follows. Section 2 overviews the verification procedure. Section 3 presents an informal description of the FLASH protocol. Section 4 illustrates how we construct a reduced model and determine an abstraction function for the proof. Section 5 proves that a specific mode of the protocol implements a sequential consistency memory model. Section 6 shows an executable description of the memory models based on the reduced behavior of the protocol.

2 The Verification Method

The verification method begins with a logical description in higher-order logic of the state graph of the implementation of a distributed computation, and a logical description of the state graph of the specification. The implementation description contains a set of *state variables*, which are partitioned into *specification variables* and *implementation variables*. The set of *states* of the implementation, Q, is the set of assignments of values to state variables. The description of the implementation also includes a logical formula defining the relation between a state and its possible successors. The relation is represented by a set of functions $\mathcal{F}$, each of which maps a given implementation state to its next state. The implementation is nondeterministic if this set has more than one function.

The description of the specification state graph is similar. The specification states are assignments of values to the *specification variables* of the implementation (implementation variables do not appear in the specification). Also, every state in the specification has a transition to itself. We call these *idle* transitions. The idle transitions are necessary for following implementation steps that do not change specification variables. We adopt the convention that components of the specification are primed, so the set of states of the specification is Q', the set of functions is $\mathcal{F}'$, etc.

The verification method is based on the usual notion of an abstraction function, which we call *abs*, which maps implementation states to specification states and which satisfies a commutativity property

$$\forall q \in Q \quad \forall N \in \mathcal{F} \quad \exists N' \in \mathcal{F}' : \\ abs(N(q)) = N'(abs(q)). \qquad (1)$$

The most interesting part of the method is how this abstraction function is defined.

The method relies on the notion that there is a set of *transactions* which the computation is supposed to implement, which are atomic at the specification level (meaning that a transaction occurs during a single state transition in the specification), but non-atomic at the implementation level. Indeed, the transactions in the implementation may involve many steps that are executed in several different components of the implementation. Formally, the transactions in the specification are the specification transition functions.

The method requires that each transaction in the implementation have an identifiable *commit step*. Intuitively, when tracing through the steps of a transaction, the commit step is the implementation step that first causes a change in the specification variables. Implementation states that occur before the transaction or during the transaction but before the commit step are called *pre-commit states* for that

transaction. The transaction is *complete* when the last specification variable change occurs as part of the transaction. The states after the commit step but before the completion of the transaction are called *post-commit states* for the transaction. A state where every committed transaction has completed is called a *clean* state.

Formally, all of the above concepts can be derived once the post-commit states are known for each transaction. The pre-commit states for the transaction are the states that are not post-commit; the commit step for a transaction is the transition from a pre-commit state to a post-commit state for that transaction; and the completion step is the transition from a post-commit state to a pre-commit state. A state is clean if it is a pre-commit state for *every* transaction.

An abstraction function consists of two parts: a *completion function* which changes the state as though the transaction had completed, and a *projection* which hides the implementation variables, leaving only the specification variables.

Once a purported abstraction function has been defined, the user must prove that it meets the commutativity requirement (1). Substituting universally quantified variables with symbolic constants and then case-splitting on the transition functions generates a finite set of subgoals in the form,

$$abs(N_j(\bar{p})(q)) = N'_{j'}(\bar{p}')(abs(q)), \tag{2}$$

where each N_j is a transition function in $\mathcal{F}$, and $\bar{p}$ is a tuple of parameters (perhaps ranging over an unknown number of components). Given j and $\bar{p}$, the user is responsible for finding a proper transition function $N'_{j'} \in \mathcal{F}'$ and $\bar{p}'$.

In most cases, the required specification step $N'_{j'}(\bar{p}')$ is the idle transition; indeed, the only non-idle transition is that which corresponds to the commit step in the implementation. We have no global strategy for proving these theorems, although most are very simple.

The above discussion omits an important point, which is that not all states are worthy of consideration. Theorem (1) will generally not hold for some absurd states that cannot actually occur during a computation. Hence, it is usually necessary to provide an *invariant* predicate, which characterizes a superset of all the reachable states. If the invariant is *Inv*, Theorem (1) can then be weakened to

$$\begin{aligned} &\forall q \in Q \quad \forall N \in \mathcal{F} \quad \exists N' \in \mathcal{F}' : \\ &\quad Inv(q) \Rightarrow \big(abs(N(q)) = N'(abs(q))\big) . \end{aligned} \tag{3}$$

($\Rightarrow$ stands for logical implication.) In other words, *abs* only needs to commute when q satisfies the *Inv*.

Use of an invariant incurs some additional proof obligations. First, we must prove that the initial states of the protocol satisfy *Inv*, and second, that the implementation transition functions all preserve *Inv*.

3 FLASH Cache Coherence Protocol

This section informally describes the cache coherence protocol used in the Stanford FLASH multiprocessor [8, 7]. The cache coherence protocol is directory-based so that it can support a large number of distributed processing nodes. Each cache line-sized block in memory is associated with *directory header* which keeps information about the line. For a memory line, the node on which that piece of memory is physically located is called *home*: the other nodes are called *remote*. The home maintains all the information about memory lines in its main memory in the corresponding directory headers.

The system consists of a set of nodes, each of which contains a processor, caches, and a portion of global memory of the system. The distributed nodes communicate using asynchronous messages through a point-to-point network. The state of a cached copy is in either invalid, shared (readable), or exclusive (readable and writable).

If a read miss occurs in a processor, the corresponding node sends out a GET request to the home (this step is not necessary if the requesting processor is in the home). Receiving the GET request, the home consults the directory corresponding to the memory line to decide what action the home should take. If the line is *pending*, meaning that another request is already being processed, the home sends a NAK (negative acknowledgement) to the requesting node. If the directory indicates there is a dirty copy in a remote, then the home forwards the GET to that node. Otherwise, the home grants the request by sending a PUT to the requesting node and updates the directory properly. When the requesting node receives a PUT reply, which returns the requested memory line, the processor sets its cache state to *shared* and proceeds to read.

For a write miss, the corresponding node sends out a GETX request to the home. Receiving the GETX request, the home consults the directory. If the line is *pending*, the home sends a NAK to the requesting node. If the directory indicates there is a dirty copy in a third node, then the home forwards the GETX to that node. If the directory indicates there are shared copies of the memory line in other nodes, the home sends invalidations (INV) to those nodes. At this point, the protocol depends on which of two modes the multiprocessor is running in: EAGER or DELAYED. If it is in EAGER, the home grants the request by sending a PUTX to the requesting node; if it is in DELAYED, this grant is deferred until all the invalidation acknowledgements are received by the home. If there are no shared copies, the home sends a PUTX to the requesting node and updates the directory properly. When the requesting node receives a PUTX reply which returns an exclusive copy of the requested memory line, the processor sets its cache state to *exclusive* and proceeds to write.

During the read miss transaction, an operation called sharing write-back is necessary in the following "three hop" case. This occurs when a remote processor in node R_1 needs a shared copy of a memory line of which an exclusive copy is in another remote node R_2. When the GET request from R_1 arrives at the home H, the home consults the directory to find that the line is dirty in R_2. Then H forwards the GET to R_2 with the source of the message *faked* as R_1 instead of H. When R_2 receives the forwarded GET, the processor sets its copy to *shared* state and issues a PUT to R_1. Unfortunately, the directory in H does not have R_1 on its sharer list yet and the main memory does not have an updated copy when the cached line is in the shared state. The solution is for R_2 to issue a SWB (sharing write-back) conveying the dirty data to H with the source *faked* as R_1. When H receives this message, it writes the data back to main memory and puts R_1 on the sharer list.

When a remote receives an INV, it invalidates its copy and then sends an acknowledgement to the home. There is a subtle case with an invalidation. A processor which is waiting for a PUT reply may get an INV before it gets the shared copy of the memory line, which is to be invalidated if the PUT reply is delayed. In such a case, the requested line is marked as invalidated, and the PUT reply is ignored when it arrives.

A valid cache line may be replaced to accommodate other

memory lines. A shared copy is replaced by issuing a replacement hint to the home. An exclusive copy is written back to main memory by a WB (write-back) request to the home. Receiving the WB, the home updates the line in main memory and the directory properly.

The above description of the protocol traces through individual transactions. However, the formal description of the protocol is written for each component, not each transaction, to make sure that the description is complete. Appendix A presents an English version of the formal description of the FLASH protocol in EAGER mode.

4 Verification of the FLASH Protocol

Verification requires two descriptions of same behavior: an implementation and a specification. Sometimes, there is an *a priori* specification as in the memory model verification in the next section. However, in most practical instances, there is only an implementation. In such cases, we extract a reduced model of the implementation using aggregation, which serves as a specification.

To apply the verification method, we first decide which state variables should be considered specification variables. In cache coherence protocols, the consistency of multiple copies of a memory line is a function of the values and states of cached copies, and the corresponding value in main memory. Therefore, the specification variables should be the state variables representing the data and states of cached copies and the data in main memory.

4.1 Extracting reduced model of the protocol

We construct a reduced model of the protocol, which we use for a specification. The reduced model is a much simpler version of the protocol which reads and writes only the specification variables. The specification steps update the values and states of cached copies in multiple nodes *atomically*.

Constructing a reduced model can be done by tracing through a transaction: 1) concatenating the implementation steps, 2) simplifying by substituting values forward through intermediate assignments, 3) eliminating statements that only change implementation variables.

The reduced model of the protocol obtained by the above procedure is shown in Table 1. Atom-WB invalidates an exclusive copy and writes back the data to main memory atomically. Atom-Invalidate simply invalidates a shared copy. There are two kinds of transactions for a read miss: Atom-Get-1 corresponds to the transaction that the home grants a shared copy to the requester when there is no dirty copy of the memory line; Atom-Get-2 corresponds to the transaction that a node with an exclusive copy grants a shared copy. For the transaction for a write miss, Atom-GetX-1 sends an exclusive copy of a memory line from the home if there is no other copies in remotes; Atom-GetX-2 transfers an exclusive ownership from a dirty node to the requester.

4.2 Abstraction function

4.2.1 Commit steps

To define the abstraction function *abs*, we should first identify commit steps of each transaction in the protocol. The transaction for a read miss begins with sending a GET request to the home. Depending on the directory state of the memory line, the request may be forwarded to a remote which contains a dirty copy of the line. These steps do not modify the specification variables, so they are pre-commit steps of the transactions. The transaction for a write miss is similar.

The commit step occurs when the home, or a remote with an exclusive copy, sends a PUT or PUTX reply, granting the request. In each case, the state of the cache line or main memory in the granting node is modified. Any future request for the memory line is processed as if the committed reply has been processed by the requesting node, even if that has not actually happened. For instance, if a GETX request arrives from R_1 right after a grant of an exclusive ownership to R_2, the home forwards the GETX to R_2 regardless of whether the PUTX sent to R_2 has arrived there or not. If a request is NAKed, then there is no change in specification variables by the transaction, so, in effect, no action occurs.

The write-back transaction begins with invalidating an exclusive copy and sending a WB request to the home. This is the commit step of the transaction because a part of the specification variables are already updated at this moment and the write-back request can not be denied by the home. The invalidation transaction is similar to this case.

4.2.2 Per-node abstraction function

Once a transaction is committed, the abstraction function *abs* simulates the post-commit steps of the transaction to complete it. The post-commit steps in the protocol are the steps that process a PUT and SWB for the transaction for a read miss, and that process a PUTX for the transaction for a write miss, and that process a WB for the write-back transaction. Therefore, to complete all the committed transactions, the *abs* should process all the messages of types PUT, PUTX, WB, and SWB.

The key question is how to complete all committed transactions in the current state, especially since the number of distributed nodes, and hence the number of committed transactions, is unknown. The general strategy, which has worked for other examples as well, is to define a *per-node* completion function for a node indexed by an unbounded variable i; the per-node function is then *generalized* to define a completion function for all of the nodes in the system.

It is quite simple to complete a committed transaction for a particular node. If a PUT message destined for node i exists, the transaction for a read miss in node i must be completed by simulating the effect of node i processing the PUT message it receives at the end of the transaction: putting the data in the message into its cache and setting the state to *shared*. The transaction for a write miss is similarly completed by processing a PUTX to node i. If node i is the home, there are two more kinds of messages possibly generated at commit steps: SWB and WB. Note that there exists at most one message of the four types destined to a particular node at any time.

This processing changes values and states of cached copies, and values in main memory. Changes to implementation variables, such as removing messages from the network, and reseting the waiting flag in the processor can be omitted from the completion function, as they do not affect the corresponding specification state. All of this computation is done solely in node i, without the involvement or interference of other nodes.

4.2.3 Global abstraction function

It is easy to generalize the per-node completion function to a completion function for all of the nodes because the

	Condition	Atomic Transaction
Atom-WB (p, a)	cache$[p][a]$.state=exclusive	cache$[p][a]$.state := invalid memory$[a]$:= cache$[p][a]$.data
Atom-Invalidate (p, a)	cache$[p][a]$.state=invalid $\vee$ cache$[p][a]$.state=shared	cache$[p][a]$.state := invalid
Atom-Get-1 (p_2, a)	$\neg\exists i$: cache$[i][a]$.state=exclusive	cache$[p_2][a]$.state := shared cache$[p_2][a]$.data := memory$[a]$
Atom-Get-2 (p_1, p_2, a)	cache$[p_1][a]$.state=exclusive $\wedge$ $p_1 \neq p_2$	memory$[a]$:= cache$[p_1][a]$.data cache$[p_1][a]$.state := shared cache$[p_2][a]$.state := shared cache$[p_2][a]$.data := cache$[p_1][a]$.data
Atom-GetX-1 (p_2, a)	$(\neg\exists i$: cache$[i][a]$.state=exclusive) $\wedge$ $(\neg\exists i$: cache$[i][a]$.state=shared $\wedge$ $i \neq p_2)$*	cache$[p_2][a]$.state := exclusive cache$[p_2][a]$.data := memory$[a]$
Atom-GetX-2 (p_1, p_2, a)	cache$[p_1][a]$.state=exclusive $\wedge$ $p_1 \neq p_2$	cache$[p_1][a]$.state := invalid cache$[p_2][a]$.state := exclusive cache$[p_2][a]$.data := cache$[p_1][a]$.data

* is an additional constraint for DELAYED mode.

Table 1: Reduced model of the FLASH protocol obtained by aggregation of distributed transactions

completions do not interact. The completion functions are simply performed in parallel.

Formally, the global implementation state is an array of node state records, indexed by the node indices. Let $cc(q[i])$ be a completion function for node i, which modifies the state variables for i (in the record $q[i]$), and returns a new record of the state variables as modified by the completion of the transactions. If $cc(q[i])$ completes committed transactions on node i, the completion function for all nodes is $\lambda q.\lambda i.cc(q[i])$. When this function is supplied a state q, it returns $\lambda i.cc(q[i])$,[1] which is an array of the completed node states, i.e., the desired clean global state. The abstraction function is simply the completion function, followed by a projection which eliminates all implementation variables.

4.2.4 Specification steps

The corresponding specification steps are simply idle transitions for pre-commit steps and post-commit steps. The only non-idle transitions are those which correspond to the commit steps of transactions.

Complete assignments of atomic transactions of the reduced model to the implementation steps of the protocol are shown in Table 3 in Appendix B. Each of the assignment corresponds to a subgoal (2) in section 2. The condition of an atomic transaction should be true at the corresponding commit step in the implementation, which is included in the invariant of the system.

4.3 Invariant

The subgoals corresponding to pre-commit steps are simple to prove because the specification variables are not modified at all. PVS can handle them automatically. However, some of the other subgoals need an invariant about the system, as discussed in section 2, to satisfy the requirement (2). If an invariant is inadequate, the proof of the requirement will fail. Analysis of the failed proof generally provides insight about additional conditions that should be added to the invariant.

[1] The notation may be a bit confusing. $\lambda i.cc(q[i])$ is a function, which when applied to a particular value of i, say i_0, returns $cc(q[i_0])$, which is the completed state for node i_0. This is effectively the same as indexing into an array of completed node states

To check those assertions, we write an invariant which is the logical "and" of the assertions, and prove that it is preserved by every step of the protocol. If the invariant is not strong enough to be preserved by all the implementation steps, we need to strengthen it. Although not intellectually difficult, this was the most time-consuming part of the proof.

The invariant we eventually derived includes the following assertions that for each memory line:

- there is at most one exclusive copy.
- there is at most one message to each node of type PUT, PUTX, WB, or SWB.
- a node is waiting for a PUT reply if there is a GET request from the node, a PUT reply to the node, or an invalidation marked.
- a node is waiting for a PUTX reply if there is a GETX request from the node, or a PUTX reply to the node.

4.4 Tricks for using theorem prover

Part of the reason that the proof is not more difficult than it was is that we have chosen to represent the network in a non-obvious way. We observe that there is at most one request/reply message for a memory line pertaining to any particular node at any time. So the network can be represented with one variable per node per memory line (sometimes associated with the source, sometimes with the destination) for relevant kinds of messages. Hence, instead of proving that there is only one message of a certain type in the network for node i at any time, we register an error whenever a message in a variable is about to be overwritten, and verify that no error occurs. The description can read a message by accessing the variable instead of removing it from a set, which is a bit more difficult to deal with in PVS. It is possible to use similar tricks in the other examples we have done.

5 Delayed Mode Implements Sequential Consistency

As mentioned before, the FLASH protocol supports two memory model modes: EAGER and DELAYED. The difference between the two modes lies in when the reply is sent

for a GETX request of a processor trying to write. In EAGER mode the reply can be sent before all the invalidation acknowledgements have been collected, while DELAYED mode only sends the reply after invalidation acknowledgements have been collected. Therefore, EAGER mode supports a more aggressive memory model which grants exclusive ownership when there are still old copies valid for reads. This difference is visible to users and may affect the correctness of synchronization code.

In this section, we show that the DELAYED mode implements the sequential consistency memory model [9], if the processors execute instructions in a sequential order one at a time, stalling at each cache miss [6]. For the proof, we use the aggregation method again. This time, the reduced behavior of DELAYED mode shown in Table 1 is considered the implementation instead of the specification as in the proof of section 4, and the specification is a state graph that models a collection of processors doing atomic loads and stores. The composition of two abstraction functions is an abstraction function, so this also implies the existence of an abstraction function from the full protocol to a sequential consistency memory model.

The sequential consistency memory model is specified in the rightmost column of Table 2. The model consists of two transactions Load_SC and Store_SC which read and write data between the registers and main memory, atomically. The specification variables model the main memory and registers. The caches are now implementation variables, which are not visible to the memory model specification.

In order to model registers in the implementation, we add a couple of steps to the reduced model which load and store a cached copy respectively. The step Load_Delayed in Table 2 simulates a processor loading a memory location by reading a cached datum into a designated register if the copy is in a shared or an exclusive state. The step Store_Delayed simulates a processor storing a memory location by writing a datum into a cache line if it has an exclusive ownership of the memory line.

The commit step of the load transaction in the protocol is Load_Delayed and that of the store transaction is Store_Delayed. The abstraction function should simulate delayed update of main memory by immediately writing back an exclusive copy, if it exists. Table 2 shows assignments of specification steps for each step of the reduced model for DELAYED mode. All the rest six steps correspond to idle transitions.

The proof involves proofs of property (2) for eight implementation transition functions with the following invariant of the system: if a cached line is in shared state, then main memory has a same data as in the cache and there is no exclusive copy; and there exists at most one exclusive copy. It is easy to see that the invariant is true in the system which consists of the eight transitions of the reduced model for DELAYED mode.

6 Executable Memory Model

We have proved that the DELAYED mode implements the sequential consistency memory model. However, there does not exist a well-defined memory model for EAGER mode, though we know that the EAGER mode supports a weaker memory model than sequential consistency. Moreover, the different behavior between the memory models is important to the users, especially to programmers, because the outputs of programs could be different depending on the modes the multiprocessor is running in.

We have previously developed executable descriptions of memory models [4, 12], derived from axiomatic specifications of memory models. We can apply the same technique for this protocol using the reduced behavior of the FLASH protocol in Table 1. The executable description automatically generates all the possible outcomes of test programs so that we can analyze the programs running on the two different modes of the protocol.

We write the executable model using a high-level description language for finite-state concurrent systems called Murφ. The description consists of a set of rules, each of which has an enabled condition and atomic transaction statements. Execution of a Murφ program begins with one of a set of initial states specified by the user. Then the following loop is executed forever: some rule whose condition is satisfied by the current state is chosen and its action evaluated, yielding a new current state. If there are no rules whose conditions are true, the execution halts. When several rule conditions are true at the same time, a choice is made arbitrarily, resulting in several possible executions. The Murφ verifier tries them exhaustively by depth-first or breadth-first search. It can prints out the value of system variables at user-specified points while exploring all the reachable states of the system.

We present a simple test program which shows different behavior between the two modes of the protocol.

```
Proc[0] :   st #1, A;  ld  B, %r1;
Proc[1] :   st #1, B;  ld  A, %r2;
```

The following is excerpted from the Murφ description for the above test program.

```
Rule  -- Proc[0] does  < st #1, A >
      -- condition to store
pc[0] = 0 & cache[0][A].state = exclusive
==> begin store(0, 1, A); end;
      -- stores the value into memory

Rule  -- Proc[0] does  < ld  B, %r1 >
      -- condition to load
pc[0] = 1 & cache[0][B].state != invalid
==> begin load(0, B, r1); end;
      -- loads the data in memory to the register

      -- Other rules are omitted.
Rule
'condition that pc[0], pc[1] are in final state'
==> begin 'print out memory and registers'; end;
```

The list of all possible outcomes of the test program generated by the description is shown below. As expected, the output of DELAYED mode is equivalent to that of the sequential consistency memory model. The output of EAGER mode is a superset of the that of DELAYED mode; the first output of EAGER mode is not possible in DELAYED mode. This confirms that the EAGER mode supports a weaker memory model than sequential consistency. The results of other test programs demonstrates that the memory model with FLASH protocol in EAGER mode is as weak as the PSO SPARC memory model.

```
EAGER:: A:1 B:1 r1:0 r2:0
EAGER:: A:1 B:1 r1:0 r2:1     DELAY:: A:1 B:1 r1:0 r2:1
EAGER:: A:1 B:1 r1:1 r2:0     DELAY:: A:1 B:1 r1:1 r2:0
EAGER:: A:1 B:1 r1:1 r2:1     DELAY:: A:1 B:1 r1:1 r2:1
```

	Condition	Action	SC Model
Load_Delayed	cache[p][a].state = shared $\vee$ cache[p][a].state = exclusive	register[p][r] := cache[p][a].data	*Load_SC* register[p][r] := memory[a]
Store_Delayed	cache[p][a].state = exclusive	cache[p][a].data := register[p][r]	*Store_SC* memory[a] := register[p][r]
Atomic Transactions DELAYED mode in Table 1	See Table 1	See Table 1	ε

Table 2: DELAYED mode conforms Sequential Consistency memory model

7 Concluding Remarks

For several years, we had firmly believed that proving the correctness of algorithms of the complexity of the FLASH cache coherence protocol was well beyond the capability of a general-purpose theorem prover. The aggregation method has broken through this barrier.

We have not considered the important problem of proving liveness properties here. However, showing the liveness using the strong fairness assumption is not difficult, because the implementation steps for each transaction in the protocol are successively enabled in sequence.

From this and many other efforts, it has become clear that finding invariants is the most time consuming part of many verification problems. More computer assistance is needed, especially for large problems.

Acknowledgements

We would like to thank Sam Owre and Natarajan Shankar at SRI International for their help with PVS system.

References

[1] J. Archibald and J. Baer. An economic solution to the cache coherence problem. In *Proc. 11th International Symposium on Computer Architecture*, pages 355–362, June 1984.

[2] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[3] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design: VLSI in Computers*. IEEE Computer Society, 1992.

[4] David Dill, Seungjoon Park, and Andreas Nowatzyk. Formal specification of abstract memory models. In *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 38–52. MIT Press, March 1993.

[5] Ásgeir Eiríksson and Ken McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *Computer Aided Verification, 7th International Conference, CAV'95*, pages 367–380, July 1995.

[6] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, 1991.

[7] Mark Heinrich. *The FLASH Protocol.* Internal document, Stanford University FLASH Group, 1993.

[8] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[9] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[10] D. Lenosky, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[11] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[12] Seungjoon Park and David Dill. An executable specification, analyzer and verifier for RMO (Relaxed Memory Order). In *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 34–41, July 1995.

[13] Seungjoon Park and David Dill. Protocol verification by aggregation of distributed actions. In *Computer Aided Verification, 8th International Conference, CAV'96*, July 1996.

[14] Fong Pong and Michel Dubois. The verification of cache coherence protocols. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 11–20, 1993.

[15] W. C. Yen and W. L. Yen. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, 34(1), January 1985.

A Detailed Description of the FLASH Protocol (EAGER Mode)

The following is a brief description extracted from the FLASH protocol documents [7].

Each cache line-sized block in main memory is associated with directory header which keeps information about the line. The directory header consists of several boolean flags: Local, Dirty, Pending, Head_Valid, List; pointers to other nodes: Head_Pointer, Sharer_List; and a number of sharers

in Real_Pointers. The Local bit is used to indicate if the local processor contains a cached copy of the line in either shared or exclusive state. The Dirty bit is set if the home thinks that there is a dirty copy of the line in the system. The Pending bit is set if the current request for the memory line is being processed by a third node. The Head_Valid bit indicates whether the Head_Pointer contains a valid pointer to a node. The Head_Pointer entry is simply a cache pointer that is packed into the directory header as an optimization. It keeps a pointer to a remote cache with a dirty copy if exists, or one of the nodes with a shared copy. The List bit indicates whether Sharer_List contains one or more pointers. Sharer_List is represented abstractly as a set[2] of pointers to the nodes that have a shared copy of the memory line. Real_Pointers contains the count of the number of sharers on the list. This count excludes the Head_Pointer and is mainly used to count invalidation acknowledgements.

The FLASH protocol consists of a set of rules which are called *handlers*. Each handler is prefixed with NI (Network Interface) or PI (Processor Interface) to indicate where the requests are generated from. PI handlers are initiated by a requesting processor and NI handlers are initiated by a message from the network. The additional notation 'Local' or 'Remote' indicates whether the processing node is the home of the requested memory address.

- **PI.Local.Get:** this handler describes actions of the home when the local processor needs a shared copy of a memory line. If Pending[3], the local processor is NAKed. Otherwise, if Dirty, the home sends a GET request to Head_Pointer and Pending is set. Otherwise, the data in main memory is copied into the local cache (in *shared* state) and Local is set.

- **PI.Local.GetX:** this handler describes actions of the home when the local processor needs an exclusive copy. If Pending, the processor is NAKed. Otherwise, if Dirty, the home sends a GETX request[4] to Head_Pointer and Pending is set. Otherwise, the data in main memory is copied into the local cache (in *exclusive* state) and Local and Dirty is set. In the last case, if Head_Valid, which indicates there are shared copies in remote nodes, the home sends INVs to Head_Pointer and the nodes in Sharer_List, Pending is set, Head_Valid is reset, and the number of invalidations is written in Real_Pointers.

- **PI.Remote.Get(X):** this handler describes actions of a remote node when the processor needs a shared (or an exclusive) copy. The remote sends a GET (or GETX) request to the home.

- **PI.Local.PutX:** this handler writes back a cached exclusive copy in the home. Dirty is reset (and Local, if not Pending) and the cached copy to the main memory is written back.

- **PI.Remote.PutX:** this handler writes back a cached exclusive copy in a remote. The remote sends a WB request to the home.

- **PI.Local.Replacement:** this handler replaces a shared copy in the home. Local is reset.

- **PI.Remote.Replacement:** this handler replaces a shared copy in a remote. The remote sends a RPL request to the home.

- **NI.NAK:** this handler describes actions of a node receiving a NAK reply. The processor clears its waiting flag and invalidation mark.

- **NI.NAKC:** this handler describes actions of the home receiving a NAKC. Pending is reset.

- **NI.Local.Get:** this handler describes actions of the home receiving a GET request from a remote. If Pending, the home sends a NAK to the source. Otherwise, if Dirty and not Local, Pending is set and the home forwards the GET to Head_Pointer with source *faked* as the original requester. Otherwise, if Dirty and Local, then writes back the exclusive copy in the local cache to main memory, sends a PUT reply to the source, and Dirty is reset, Head_Valid is set, and Head_Pointer is set to the source. Otherwise, the home sends a PUT reply to the source; If Head_Valid, List is set, Real_Pointers is incremented, the source is added to Sharer_List. If not Head_Valid, Head_Valid is set, Head_Pointer is set to the source.

- **NI.Local.GetX:** this handler describes actions of the home receiving a GETX request from a remote. If Pending, the home sends a NAK to the source. Otherwise, if Dirty and not Local, then Pending is set and the home forwards the GETX to Head_Pointer with source *faked* as the original requester. Otherwise, if Dirty and Local, the home sends a PUTX reply to the source with the exclusive data from the local cache, and Local is reset, Head_Valid is set, and Head_Pointer is set to the source. Otherwise, the home sends a PUTX to the source with the data in main memory.

  In the last case, if not Dirty and Head_Valid, Dirty is set, List is reset, and if Head_Pointer is not equal to the source, Pending is set, and the home sends an INV to the Head_Pointer, and Head_Pointer is set to the source. If Local, invalidates the local copy, and if List, the home send INVs to all the nodes in Sharer_List and set Real_Pointers to the number of invalidations. Otherwise, if not Dirty and not Head_Valid, Head_Valid and Dirty are set, Local is reset, and Head_Pointer is set to the source.

- **NI.Remote.Get:** this handler describes actions of a remote receiving a GET request. If the cached data is in the *exclusive* state, it is changed to *shared* and the node sends a PUT reply to the source (and also SWB to the home if the source is not the home). Otherwise, the node sends a NAK to the source and a NAKC to the home.

- **NI.Remote.GetX:** this handler describes actions of a remote node receiving a GETX request. If the cached data is in *exclusive* state, it is invalidated and the node sends a PUTX reply to the source (and a forward acknowledgement FWAK to the home if the source is not the home). Otherwise, the node sends a NAK to the source and a NAKC to the home.

[2] The FLASH protocol uses a linked list for sharers (inside a local node) by dynamic pointer allocation.

[3] I.e., the Pending bit is set in the directory.

[4] The original protocol uses a different request upgrade for an exclusive copy, rather than using getx, when the cache has a shared copy The reason is to enhance performance by avoiding unnecessary data transfer. However, the two requests are processed in the same manner except whether the reply contains the cached data or not. We did not model the upgrade request in the verified description

- **NI.Local.Put:** this handler processes a PUT reply to the home. Local is set, Dirty and Pending are reset, and the shared copy is put into the local cache.
- **NI.Remote.Put:** The shared copy is put into the cache.
- **NI.Local.PutX:** this handler processes a PUT reply to the home. Local is set, Head_Valid and Pending are reset, and the exclusive copy is put into the local cache.
- **NI.Remote.PutX:** The exclusive copy is put into the cache.
- **NI.Inval:** Receiving a INV, the remote invalidates the cached copy and sends an INVAK to the home. If the node was waiting for a PUT, it marks the line invalidated.
- **NI.InvalAck:** this handler describes actions of the home receiving an INVAK. Real_Pointers is decremented. If it reaches to zero, Pending is reset (and Local if not Dirty).
- **NI.Writeback:** this handler describes actions of the home receiving a WB request. Dirty and Head_Valid are reset and the data is written back into the main memory.
- **NI.ForwardAck:** this handler describes actions of the home receiving a FWAK. Pending is reset. If Dirty, Head_Pointer is set to the source.
- **NI.SharingWriteback:** this handler describes actions of the home receiving a SWB. Dirty and Pending are reset, List is set, Real_Pointers is incremented, the source is added to Sharer_List, and the data is written back into main memory.
- **NI.Replacement**: this handler describes actions of the home receiving a RPL. The source is removed from Sharer_List if found and Real_Pointers is adjusted.

B Assignments of Implementation/Specification Steps

In the FLASH protocol, some handlers perform commit steps in some cases and not in others. In order to establish the necessary correspondence between implementation steps and specification steps in a proof of property (2), we need to split these handlers into multiple transition functions, each of which either always commit or never commit. For example, the PI.Local.Get handler simply NAKs the local processor if the requested line is pending (pre-commit step), or sends a request to a remote if there is a dirty copy (pre-commit step), otherwise, it updates the state and data of the local cache which are specification variables (commit step). In the first two cases, the reduced model should take idle transitions, but in the last case, a corresponding atomic transaction should be taken.

The PI.Local.Get handler is decomposed into two different transition functions PI.Local.Get.else and PI.Local.Get.put with disjoint enabling conditions, where the first includes the pre-commit steps, and the latter corresponds to the commit step. In the same manner, other handlers are decomposed if necessary. In Table 3, the protocol steps named with suffix 'ex' (with superscript 1) correspond to the decomposed handlers when the home holds an exclusive copy. The protocol steps named with suffix 'inv' (with superscript 2) correspond to the decomposed handlers when the requesting node is invalidation marked. Note that these decompositions do not change the original protocol implementation.

Protocol Step at node p	Atomic Transaction (Reduced model)
PI.Local.Get.else	ε
PI.Local.Get.put	Atom-Get-1(*home*)
PI.Remote.Get	ε
PI.Local.GetX.else	ε
PI.Local.GetX.putx	Atom-GetX-1(*home*)
PI.Remote.GetX	ε
PI.Local.PutX	Atom-WB(*home*)
PI.Remote.PutX	Atom-WB(p)
PI.Local.Replace	Atom-Invalidate(*home*)
PI.Remote.Replace	Atom-Invalidate(p)
NI.NAK	ε
NI.NAK.Clear	ε
NI.Local.Get.else	ε
NI.Local.Get.put	Atom-Get-1(GET.src)
NI.Local.Get.put.ex[1]	Atom-Get-2(*home*, GET.src)
NI.Local.Get.put.inv[2]	Atom-Get-1(GET.src); Atom-Invalidate(GET.src)
NI.Local.Get.put.ex.inv	Atom-Get-2(*home*, GET.src); Atom-Invalidate(GET.src)
NI.Remote.Get.else	ε
NI.Remote.Get.put	Atom-Get-2(p, GET.src)
NI.Remote.Get.put.inv[2]	Atom-Get-2(p, GET.src); Atom-Invalidate(GET.src)
NI.Local.GetX.else	ε
NI.Local.GetX.putx	Atom-GetX-1(GETX.src)
NI.Local.GetX.putx.ex[1]	Atom-GetX-2(*home*, GETX.src)
NI.Remote.GetX.else	ε
NI.Remote.GetX.putx	Atom-GetX-2(p, GETX.src)
NI.Local.Put	ε
NI.Remote.Put	ε
NI.Local.PutXAcksDone	ε
NI.Remote.PutX	ε
NI.Inv	Atom-Invalidate(p)
NI.InvAck	ε
NI.WB	ε
NI.FAck	ε
NI.ShWB	ε
NI.Replace	ε

Table 3: Correspondence of transition functions in the protocol with atomic transactions (EAGER mode)

Table 3 lists all the transition functions of the protocol in EAGER mode and the corresponding atomic transactions of the reduced model. The atomic transactions are listed with properly instantiated parameters. The table for DELAYED mode would be the same as Table 3 except that ownership transfer (GetX-Atom) corresponds to the protocol step which processes the last invalidation acknowledgement.