Formal Verification of Delayed Consistency Protocols

Fong Pong
Sun Microsystems Computer Corporation
Technology Development Group
fong.pong@eng.sun.com

Michel Dubois
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles CA 90089-2562
dubois@paris.usc.edu

Abstract

In a cache-coherent, shared-memory multiprocessor system, data consistency among cached copies can be delayed until synchronization points under relaxed memory consistency models. Some protocols called delayed consistency protocols take advantage of this flexibility to reduce cache miss rates and memory traffic. However, they are very complex and validating their correctness, even at the behavior level, is a challenge. We have successfully applied a new verification tool to verify the delayed consistency protocol at the behavior level. The method is called SSM [22] for Symbolic State Model. The contribution of this paper, besides verifying the protocol, is to demonstrate how to deal with relaxed memory models and latency tolerance hardware in the context of SSM.

1 Introduction

Verification models of cache coherence protocols for sequentially consistent systems are relatively simple because the status of a single memory location is traced and a memory block is assimilated to a single word since the coherence unit of cache protocols is a memory block [5, 17, 22]. Coherent accesses to a memory block, even to different memory words, are serialized and tracking a single memory block is enough to verify the property of data consistency.

Verification of cache protocols is much more complicated in the context of relaxed memory models [1, 6, 11, 25]. Consider the execution of figure 1 in a weakly-ordered system. The write by p0 and the read by p1 are ordered by paired Test&Set and Unset synchronization accesses. Since the read of p1 cannot be completed before the write of p0 due to the explicit synchronization in between, p0 does not necessarily block at the write operation while waiting for the invalidation of p1’s copy. The only requirement for a correct execution is that the value written by p0 must be visible to p1 before p1 reads it.

To enforce this requirement the hardware usually relies on the lock accesses. The value could propagate from p0 to p1 when p0 releases the lock. Thus, cache coherence is not enforced on-the-fly and is delayed. To verify cache protocols in such systems, the state expansion process which searches for all reachable system states must take into account the synchronization accesses as well as the regular data accesses. In the execution sequence of figure 1, p1 is allowed to issue its read only after p0 and p1 have performed their Unset and Test&Set operations respectively.

Figure 1. Explicit synchronization in a weakly-ordered memory model.

In this paper, we show how to verify a particular protocol called the delayed consistency protocol [7] designed for relaxed memory models. Delayed consistency supports all the performance improvement techniques (e.g., pipelining of memory accesses and effective use of store buffers [9, 16, 25]) offered by relaxed memory models. More interestingly, stores are always local operations. There is no need to request ownership when a shared block is modified and, on a write miss, the new value is recorded locally without blocking the processor. The ping-ponging of blocks associated with large block sizes and due to false sharing [8, 10, 24] is reduced. Because the delayed protocol incorporates hardware features found in a wide spectrum of designs, the verification model is general enough to apply to other protocols under weak ordering.

The validation of delayed protocols is complex because the verification model must be built upon a synchronization model assumed by the memory model. Since the enforcement of data consistency is deliberately delayed, caches may have data copies with inconsistent values for long periods of time. Even if a block is still accessible in a cache, some of its words may be stale and, in general, the state of individual words are
different from the state of the block containing these words; thus, the verification method must track the state of individual words as well as of blocks. Verification techniques based on proving that all reached states are permissible, in the sense that individual cache states are compatible, must be ruled out [18].

We have developed a new verification method based on a symbolic state model (SSM) [22]. The SSM method verifies a cache protocol by exploring the entire state space of the protocol, but it exploits a unique feature of cache protocols: it only needs to keep track of whether there are 0, 1, or several copies in a particular state. This symbolic state representation leads to a drastic reduction of the system state space, reducing verification time and consuming less memory space to store states. Protocols are verified for any system size, unlike current approaches which can only verify small-scale systems [20, 21].

In the next section, we present an execution model for programs that obey a synchronization model called data race free (DRF) [1, 2]. This execution model is used to restrict the state expansion process of the SSM method so that only legal sequences of memory accesses are generated with respect to the synchronization model. In section 3, we briefly overview the SSM method in order to make the paper self-contained; more details can be found in [19-22]. We then describe the main features of the delayed protocol in section 4. Because our primary goal is to demonstrate that delayed consistency does not compromise the system correctness for performance, we assume atomic memory accesses. Non-atomic memory accesses can be modeled in the same way by adding transient states to the protocol and by tracking messages exchanged among protocol components as was done in [20, 21].

Section 5 describes the verification model and section 6 shows the results. We finally conclude in section 7.

2 Execution Model of Data Race Free Programs in Weakly-Ordered Systems

2.1 Data Race Free Synchronization Model

The data race free (DRF) model [1, 2] is a simple characterization of programs that forbid data races among conflicting accesses [11]. Two accesses are conflicting if they access the same memory location and at least one of them is a write. The core of the DRF model is a partial order relation called happen-before which governs the ordering of accesses in the execution of a DRF program. happen-before is built upon program order (po) and synchronization order (so).

Since sequences of memory accesses observed in executions of correct DRF programs may not contain data races, we must restrict the state expansion of a state enumeration method to generate legal orderings of memory accesses only. This is achieved by building an execution model for DRF programs which adheres to the program order and the synchronization order.

2.2 An Abstract View of Synchronization Accesses

To build the execution model, we take an abstract view of synchronization accesses. Synchronization accesses are classified into two categories: lock and unlock operations. Conflicting accesses are only ordered by lock and unlock operations. In actual systems, conflicting accesses are sometimes ordered by high-level synchronization instructions. Figure 2 shows a barrier synchronization implemented with high-level synchronization instructions given in [2, 12]. In this example, p₀ and p₁ read a shared variable x in a semi-critical section; before p₂ is allowed to write a new value to location x, it must wait until both p₀ and p₁ terminate the semi-critical section. Since p₁ is the last processor checking at the barrier, the last write to Flag by p₁ is indeed a synchronization write [2] or a release operation [11] and the reads to Flag by p₀ and p₂ are synchronization reads or acquire operations. There is no direct synchronization order between the read(x) of p₀ and the write(x) operation of p₂. These two conflicting accesses are ordered by a chain of synchronization accesses (as circled in figure 2), including the two fetch&inc operations of p₀ and p₁. If we wish to model all sequences of memory operations in a verification model, we would have to include the read-modify-write cycle embedded in a fetch&inc instruction. The verification model would be uselessly complex.

Rather, there is no need to generate all memory accesses of figure 2 to construct an execution model. We can take a more abstract view of synchronization accesses as in [2]:

A synchronization between processors pₓ and pᵧ is formed by a write access of pₓ and a read access of pᵧ to the same location so that pᵧ’s read

Figure 2. Barrier synchronization using fetch&inc.
returns the value stored by \( p_x \)'s write, and the value is used by \( p_y \) to conclude that all operations of \( p_x \) before the write is complete.

We simply call \( p_x \)'s write an unlock operation and \( p_y \)'s read a lock operation because \( p_x \) lifts the barrier such that \( p_y \)'s execution can progress (this is a typical release-acquire synchronization). We can also generalize this abstraction to the cases of barrier synchronizations. Given \( N \) processors to synchronize, the barrier is terminated only when every processor executes an unlock operation. This abstract view of the example in figure 2 is displayed in figure 3. In this case, the unlocks actually behave as operations increasing the value of a counter and the lock succeeds when the value of the counter reaches the barrier count.

Taking a high-level view of synchronization orders has the advantage that the model is independent of the exact set of synchronization instructions (e.g., semaphore, test&set, barrier, or spinning on shared flag variables [12]) supported in the system. The model also provides simpler insight into the synchronization orders among conflicting accesses. In the example of figure 3, \( p_0 \)'s read and \( p_2 \)'s write are now ordered by a direct synchronization relation rather than by a chain of synchronization accesses.

![Figure 3. An abstract view of barrier synchronization.](image)

Note that this abstract view of synchronization does not change the semantics and execution results of DRF programs. To ensure a correct model of the target system architecture, the state expansion process simply executes all operations that need to be performed at synchronization points when locks and unlocks are executed.

### 2.3 Execution Model of DRF0 programs in Weakly-Ordered Systems

With respect to a single shared location, we build the execution model by assuming first that no synchronization accesses are needed. Then, they are added whenever data races are detected, so that the traces of accesses conform to the hypothesis of data race free. The execution model characterizes the execution of a processor in three modes: (1) **Semi-Critical Section** SCS (read only; concurrent readers), (2) **Critical Section** CS (exclusive reader and writer), and (3) **Out Mode** OUT (not accessible).

The state transitions modeling the execution modes of processors are described in the following and are illustrated in figure 4:

1. When a processor \( p_i \) is in the OUT mode, \( p_i \) is not allowed to access the shared memory location. To access the shared data, \( p_i \) must execute a lock operation first.
2. When \( p_i \) successfully executes a lock operation, \( p_i \) enters either the SCS mode or the CS mode. If the state transition leads \( p_i \) to the SCS mode, there must be no other processors in the CS mode. If \( p_i \) wants to enter the CS mode, it must wait until all other processors terminate their SCS or CS via an unlock operation.
3. The processor in the CS mode is the only one allowed to issue writes to the shared memory location, whereas concurrent reads are allowed for processors in SCS.

![Figure 4. Processor execution modes from the standpoint of local processor \( p_i \).](image)

We will prove that the model correctly characterizes execution sequences of memory accesses of DRF programs in weakly-ordered systems. But first we briefly describe the SSM verification method.

### 3 Symbolic State Model

Researchers in protocol verification have recently exploited the symmetry of cache-coherent, shared-memory multiprocessor systems. Namely, contexts of processors can be swapped without affecting the correctness of the system [14]. In addition to the system symmetry, the SSM method exploits a unique property of cache protocols. 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 dirty copy). As a result, the SSM uses four repetition constructors \([0, 1, +, *]\) to map system states to more abstract states which do not keep track of the exact number of copies [19-22]. The + indicates one or multiple instances and the * indicates zero, one, or multiple instances. According to the possible states specified, repetition constructors are ordered as \( 1 < + < * \) and \( 0 < * \).

In a system with an unspecified number of caches,
we group all caches in the same state into a state class and specify the number of caches in the class by one of the replication 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 (Inv+, RO*). Thus, a global state (called composite state) has the general form:

\[ \{ r_1, r_2, \ldots, r_n, q_1, q_2, \ldots, q_m \} . \]

By extending the order among repetition constructors, a monotonous state containment relation exist between composite states [19]. Specifically, composite state \( S_2 \) contains composite state \( S_1 \), or \( S_1 \subseteq S_2 \), if

\[ \forall q_1 \in S_1 \exists q_2 \in S_2 \rightarrow q_1 \leq q_2 \quad \text{i.e.} \quad r_1 \leq r_2. \]

Also, if \( S_1 \subseteq S_2 \), for every \( S_1 \) reachable from \( S_1 \), there exists \( S_2 \) reachable from \( S_2 \) such that \( S_1 \subseteq S_2 \). Therefore, during the expansion process, 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, which are mutually non-contained by each other.

**Theorem 1** Given a DRF program, all possible sequences of memory accesses during executions of the program on a weakly-ordered system are characterized by the proposed execution model.

**Proof:** Starting with a state in which all processors are in the OUT mode, the resulting state diagram for a system of arbitrary number of processor is shown in figure 5. (Note that \( S_1 \) is not contained by \( S_2 \) in order to distinguish the two cases of one or multiple processors in SCS. This modeling technique is the same as we did for the Illinois protocol in [22].) Theorem 1 derives from the global state diagram. The system allows only one processor in the CS mode and the processor holds the most recent value of the shared memory location according to the happen-before relation. When the processor exits the CS, it must execute an unlock operation. Processors in the OUT mode need to execute a lock operation before they are allowed to access the shared memory location, so that a correct synchronization order is imposed between the writes by the processor in CS and the accesses by processors in OUT. When a processor wants to write, it must wait until all other processors in CS or SCS change their mode to OUT by an unlock. The writing processor then executes a lock operation entering the CS mode. Thus, the write is correctly ordered with respect to preceding reads. \( \square \)

The execution model assumes that no initial synchronization access is needed, and synchronizations are added when data races are detected. Therefore, every possible sequence of memory accesses that can be generated by a DRF program can be characterized by this execution model.

**4 Delayed Consistency Protocol**

**4.1 System Configuration**

The delayed protocol we have verified was first published in [7]. (Of course, many variations to this protocol are possible.) It is an ownership-based, write-invalidation protocol. The effect of outgoing (incoming) invalidations are delayed until the next unlock (lock) instruction is executed by the local processor.

![Figure 5. The composite behavior of processors accessing a shared memory location.](image)

Using the SSM, we can now prove that the execution model of section 2 correctly characterizes sequences of memory accesses of DRF programs.

**Theorem 1** Given a DRF program, all possible sequences of memory accesses during executions of the program on a weakly-ordered system are characterized by the proposed execution model.

**Proof:** Starting with a state in which all processors are in the OUT mode, the resulting state diagram for a system of arbitrary number of processor is shown in figure 5. (Note that \( S_1 \) is not contained by \( S_2 \) in order to distinguish the two cases of one or multiple processors in SCS. This modeling technique is the same as we did for the Illinois protocol in [22].) Theorem 1 derives from the global state diagram. The system allows only one processor in the CS mode and the processor holds the most recent value of the shared memory location according to the happen-before relation. When the processor exits the CS, it must execute an unlock operation. Processors in the OUT mode need to execute a lock operation before they are allowed to access the shared memory location, so that a correct synchronization order is imposed between the writes by the processor in CS and the accesses by processors in OUT. When a processor wants to write, it must wait until all other processors in CS or SCS change their mode to OUT by an unlock. The writing processor then executes a lock operation entering the CS mode. Thus, the write is correctly ordered with respect to preceding reads. \( \square \)

The execution model assumes that no initial synchronization access is needed, and synchronizations are added when data races are detected. Therefore, every possible sequence of memory accesses that can be generated by a DRF program can be characterized by this execution model.

The system architecture model, illustrated in figure 6, consists of processors each with an associated cache in a shared-memory configuration with a directory-based protocol [4]. Each cache has an Invalidation Send Buffer (ISB) in which modified non-owned blocks are buffered (using one dirty bit \( d \) per word in each block) and an Invalidation Receive Buffer (IRB) in which received invalidations are buffered. Cache blocks with pending invalidations in the IRB are considered as Stale copies and are still accessible until the next lock-
acquire executed by the local processor; the updated blocks in the ISB are propagated to memory at lock-release points (at the latest). In such a system, synchronization variables are not subject to delay and are assumed to be stored in different memory regions than other shared data.

4.2 Cache States and Algorithm

In the delayed protocol, every cache block can be in any one of four states: Invalid, Keeper, Stale, and Owner. A Keeper copy was loaded in cache on a read miss; a Stale copy is a valid block with pending invalidation in the IRB and an Owner copy implies that the local copy has been modified. An Owner copy does not necessarily mean an exclusive copy; the Owner is simply responsible for providing the latest copy of the block in response to misses.

Another unique feature of the delayed protocol is that local modifications can be buffered in the ISB even if the block is invalid in the cache, which necessitates that a miss in the cache must first check the ISB. This leads to complex definitions for cache access misses and hits.

Definition 1 (Cache Access Misses and Hits in the Delayed Protocol) In the delayed consistency protocol, cache access misses and hits are defined on the local cache state and the ISB state. Assume that accesses are made to location \( a \) in block \( A \).

1. A read miss means \( A \) is invalid in the cache and \( a \) is not valid in the local ISB.
2. A read hit means \( A \) is valid in the cache or \( a \) is valid in the local ISB.
3. A write miss occurs when \( A \) is invalid in the cache and no entry in \( A \) is in the local ISB.
4. A write hits when \( A \) is valid in the cache or an entry of \( A \) is in the local ISB.

A brief description of the delayed protocol is given in the Appendix; detailed explanations of coherence messages and transactions can be found in [7].

5 Verification Model

5.1 Modeling Data Words

In the verification model, the abstract model splits a data block in two parts denoted by \( wd_1 \) and \( wd_2 \). \( wd_1 \) is a word in the block subject to data consistency verification. Concurrent accesses to \( wd_1 \) are restricted by the execution model of section 2. The rest of the block is abstractly represented by \( wd_2 \). We assume that the block has an arbitrary size greater than one word. Modeling \( wd_2 \) is important because accesses to \( wd_2 \) can affect the state of \( wd_1 \). By symmetry, the verification results obtained by tracking and checking the copies of \( wd_1 \) can be generalized to all other words [14].

5.2 Automaton States

With respect to the data block, the finite state automaton for the behavior of each cache consists of three elements:

1. Cache state \( q \), where \( q \in \{ \text{Owner, Keeper, Stale, Invalid} \} \).
2. Invalidation Send Buffer (isb) state, where \( \text{isb} \in \{ \text{isb00, isb01, isb10, isb11} \} \). These four states correspond to four distinct possibilities: 1) no entry for the block in the ISB, 2) modifications of \( wd_2 \) stored in the ISB, 3) modifications of \( wd_1 \) stored in the ISB, and 4) an entry in the ISB with updates of both \( wd_1 \) and \( wd_2 \).
3. Process mode \( (ps) \), where \( ps \in \{ \text{OUT, SCS, CS} \} \). Mode \( \text{CS} \) indicates that the local process is in the critical section and has the right to modify \( wd_1 \). Processors in \( \text{SCS} \) are allowed to read \( wd_1 \) only, while processors in \( \text{OUT} \) cannot access \( wd_1 \).

Note that the state of IRB is already embedded in the cache state, i.e., a stale copy indicates a pending invalidation in the IRB. By combining these three elements, we characterize the automaton by its state \( q_{isp} \).

5.3 Automaton Operations

The set of operations triggering state transitions are:

1. read and write accesses to \( wd_1 (wd_2) \) are denoted as \( rd_1 (rd_2) \) and \( wt_1 (wt_2) \).
2. lock and unlock are synchronization accesses.
3. \( \text{remisb} \) corresponds to the removal of the block from the ISB.
4. \( \text{remstale} \) removes the entry in the IRB and invalidates the block in the cache.
5. \( \text{repl} \) is a regular replacement operation.

In the expansion process, a processor must execute lock before it can make any access to \( wd_1 \). Terminations of \( \text{SCS} \) or \( \text{CS} \) are via unlocks. All other operations including \( \text{remisb} \), \( \text{remstale} \), and \( \text{repl} \) can be executed at arbitrary times. Thus, the state expansion process simulates arbitrary interleaving of events. Executions of \( \text{remisb} \) and \( \text{remstale} \) trigger the same actions as unlock and lock (resp.) and are needed in any realistic implementation with ISB and IRB of finite capacity. A special instruction “Prepare for Synchronization” [7] that removes the ISB entries is also equivalently modeled by remisb.

5.4 Model for Data Consistency

Within this context, our approach proves the validity of the delayed protocol by showing that the protocol guarantees data consistency for all possible interleaving of cache events. For correctly written programs, the delayed protocol must always return the latest value on
each load. The latest value is defined by the latest write prior to the load in the happen-before relation. We formulate this condition within the framework of the reachability expansion as follows.

**Definition 2 (Data Consistency)** With respect to a particular memory location, the protocol preserves data consistency if and only if the following condition is always true during the reachability analysis: the family of global states originated from \( G' \), including \( G' \) itself, always observe the value written by a STORE transition \( \tau \) which brings a global state \( G \) to \( G' \) or the value written by STORE transitions after \( \tau \). That is, states reached by expanding \( G' \) are not allowed to access any (old) value defined before \( \tau \).

To verify the condition for data consistency in definition 2, the verification method keeps track of the values of data copies explicitly [19-22]. We associate every cache to a variable of data copies explicitly. We associate every variable \( \text{isb} \) corresponding cache. Another variable \( \text{remstale} \) keeps the value of \( \text{wd} \) in the \( \text{ISB} \). These variables take values from domain \( \{ \text{nodata, fresh, obsolete} \} \). A single global variable \( \text{mwd} \in \{ \text{fresh, obsolete} \} \) holds the value of \( \text{wd} \) in the memory. When a processor modifies \( \text{wd} \), its own \( \text{cwd} \) takes the value fresh while copies of \( \text{cwd} \) in caches and in memory become obsolete. The assignment of the value of one variable to other variables emulates data transfers. The value of \( \text{wd} \) is not modeled because the role of \( \text{wd} \) is just to model the changes in the states of \( \text{wd} \) caused by accesses to \( \text{wd} \). The check of data consistency on \( \text{wd} \) can be generalized to all other memory locations by symmetry arguments [14]. Also, because the \( \text{remstale} \) and \( \text{remstale} \) operations (which have the same effect as executing \( \text{unlock} \) and \( \text{lock} \)) are unconditionally applicable in the state expansion process, arbitrary interleaving of synchronization accesses are correctly modeled.

**6 The Verification Results**

**6.1 Correctness of the Delayed Protocol**

The delayed protocol is proven to preserve the data consistency by exploring the state space exhaustively. We start the expansion process by an initial state \( \left( \text{Invalid}_{out}^{\text{jsb00+}} \right) \) in which no cache has a block copy and all processors are in the \( \text{OUT} \) mode and prevented from accessing \( \text{wd} \).

In the SSM, we specify correctness conditions as in-line assertions in state expansion rules [21]. A protocol error is detected when a violation of the in-line assertions occurs. With respect to \( \text{wd} \) and processor \( P_i \) in state \( q_{ps_i} \), the delayed protocol has the following properties:

1. \[ (ps_i = cs) \lor (ps_i = scs)) \land (q_i \neq \text{Invalid}) \land (\text{cwd}_i = \text{fresh}) \]

This property partially proves that the delayed protocol preserves data consistency. If processor \( P_i \) can access its local copy of \( \text{wd} \), the copy must have a fresh value.

2. When there is a modified copy stored in the local \( \text{ISB} \), all other copies are obsolete. We have

\[
(\text{isb}_i = \text{isb}10) \lor (\text{isb}_i = \text{isb}11) \lor \text{(isbw}_i = \text{fresh}) \land (m\text{wd}_i = \text{obsolete}) \land \\
(\neg \exists j \neq i, (\text{cwd}_j = \text{fresh}) \lor (\text{isbw}_j = \text{fresh}))
\]

This property confirms that the delayed protocol ensures data consistency. Since a read access may miss in the local cache but hit in the \( \text{ISB} \), the word present in the \( \text{ISB} \) copy must have the most recent value. The new value stored in the \( \text{ISB} \) is not yet visible to other processors and therefore, multiple copies of the same memory location in different \( \text{ISB} \)s are not allowed.

**6.2 Performance Results**

In order to assess the efficiency of the SSM method, we have applied the Mur\( \varphi \) verification system [5, 14] to verify the protocol. The Mur\( \varphi \) implements an explicit state enumeration method, incorporating state encoding to reduce memory usage and hash tables to speed up the state-search and state-comparison operations. There are two Mur\( \varphi \) systems: one exploits the system symmetry (Mur\( \varphi \)-s) and one does not (Mur\( \varphi \)-ns).

Table 1 shows the performance of the three tools for the verification of the delayed protocol on a SPARCstation 10 Model 30 with 128 MBytes of memory. All verification runs start with an initial state in which no processor has a copy, all \( \text{ISB} \) entries are empty and all processors are in the \( \text{OUT} \) mode and prevented from accessing the memory location \( \text{wd} \).

The size of the state space of the delayed protocol quickly increases with model size, even with the assumption of atomic memory accesses. Using the same assumption of atomic memory accesses, the delayed protocol has a much higher level of complexity as compared with the verification of other protocols [19]. The number of essential states reported by the SSM is 58 which is much larger than the number of essential states (usually under 8, cf. [22]) reported in traditional protocol designs.

The most complex essential state characterizing the delayed protocol consists of 14 processors in different states. Therefore, the SSM effectively verifies models with more than 14 processors. For conventional protocols [3], a model of 3 or 4 processors is able to cover all possible cases (under atomic memory accesses). As a
result, we can assume that the size of the state space will be excessive for the delayed protocol if non-atomic memory accesses are modeled.

The size of the state space increases drastically with the model size for state enumeration methods. The symmetric Murϕ-s shows significant reductions in both the global state space and the search state space over non-symmetric Murϕ-ns. However, Murϕ-s takes twice as much time as Murϕ-ns takes for the model size with five processors. The computation of canonical states for symmetrically equivalent states may be the cause for this abnormality.

7 Conclusion

In this paper, we verified the delayed consistency protocol, which has high complexity even at the behavior level. This protocol is shown to be correct in the sense that processors can delay the sending and receiving of invalidations until synchronization points in a weakly-ordered system. Because the design of delayed protocol covers a wide range of techniques that can be exploited in relaxed memory models for good performance, the verification approach can be applied to similar protocols. This includes protocols designed for systems using write caches [9] which can be verified in the same way because the functionality of write caches is essentially the same as the ISB employed in the delayed protocol. Although the delayed protocol is complicated, the symbolic verification method reduces the complexity of the verification process to a point where a complete and reliable verification is possible.

8 References


<table>
<thead>
<tr>
<th>Method</th>
<th>Number of processors</th>
<th>Size of global state space</th>
<th>Size of search state space</th>
<th>Verification time (seconds)</th>
<th>Memory (Mbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Murϕ-ns</td>
<td>2</td>
<td>484</td>
<td>4,184</td>
<td>0.9</td>
<td>0.007</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6,228</td>
<td>76,170</td>
<td>14.7</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>75,088</td>
<td>1,202,544</td>
<td>268.2</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>905,312</td>
<td>18,132,640</td>
<td>3,303.1</td>
<td>17.35</td>
</tr>
<tr>
<td>Murϕ-s</td>
<td>2</td>
<td>248</td>
<td>2,148</td>
<td>0.8</td>
<td>0.004</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1,206</td>
<td>14,844</td>
<td>14.9</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4,500</td>
<td>72,575</td>
<td>330.1</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>14,366</td>
<td>288,932</td>
<td>7,949.3</td>
<td>0.28</td>
</tr>
<tr>
<td>SSM</td>
<td>any n &gt; 1</td>
<td>58</td>
<td>6,864</td>
<td>0.67</td>
<td>0.01</td>
</tr>
</tbody>
</table>
Appendix

This appendix contains an outline of the delayed protocol. We denote $C_i$ as the local cache and $C_j$ is a remote cache.

1. **Read hit.** No coherence action is taken.
2. **Read miss.** If there exists a $C_i$ in the Owner state, $C_i$ is requested to update memory and the memory then sends a copy to $C_j$. If no owner exists, the memory provides its copy to $C_j$ directly. In all cases, $C_j$ receives the block in the Keeper state. If an entry of the block is in the local ISB, the ISB copy is merged with the received data and stays in the buffer.
3. **Write hit.** If $C_i$ is in Owner state, the write is performed to the local cache. Otherwise, $C_i$ must have a Keeper copy, or a Stale copy, or an Invalid copy but with a valid ISB entry. An ISB block is allocated if it doesn’t exist when the access hits on a Keeper or on a Stale copy. The written value is always stored into the ISB block and the local cache copy (if the cache block is not Invalid). The dirty bit associated with the written word in the ISB block is set.
4. **Write miss.** A request for an owner copy is issued by $C_j$ to the memory. The memory then multicasts invalidations to caches with their presence bits set (these caches must be in either Keeper or Owner state). When a cache $C_j$ receives the invalidation, a memory-update request is sent to the memory if $C_i$ has an Owner copy; otherwise an invalidation acknowledgement is sent back to the memory. The invalidation message received by $C_j$ is buffered in the local IRB and $C_i$ still maintains an accessible Stale copy. The actual invalidation of $C_j$ is delayed until the next lock access. After all copies have been staled, $C_j$ gets a copy from memory and becomes the new Owner.
5. **Lock (Remove IRB).** Before acquiring a lock, the local processor must ensure that invalidations buffered in the local IRB are completed in the local cache. In other words, all Stale copies are invalidated.
6. **Unlock (Remove ISB).** Before the processor executes an unlock operation, all entries in the local ISB must be purged first. The operation is somewhat complicated. If the corresponding cache entry of the purged ISB block is in the Keeper state, the memory performs three operations: (1) update the memory copy with the modified words in the ISB block, (2) invalidate all other Keepers, which become Stale, and (3) notify the cache purging the ISB block that it is the new Owner. If the local cache purges an ISB entry in the Stale or Invalid states, the cache state remains Stale or Invalid. The memory is updated with the received ISB block.
7. **Replacement.** When an Owner is victimized the memory is updated. If the victim is in the Keeper state, two cases are possible depending on the state of the ISB. If there is no entry of the block in the ISB, a request is issued to the memory to clear the presence bit. Otherwise, the ISB block is written back to memory; the memory is updated and sends invalidations to all other Keepers. If the replaced block is in the Stale state, the local state changes to Invalid (removing the invalidation from the IRB). Moreover, if an ISB entry exists, the same operations of updating the memory and invalidating all other Keepers are executed by the protocol.