

Formal Verification of a Novel Snooping Cache Coherence Protocol for CMP

Xuemei Zhao, Karl Sammut, and Fangpo He

School of Informatics and Engineering
Flinders University, Australia
{zhao0043, karl.sammut, fangpo.he}@flinders.edu.au

Abstract

The Chip Multiprocessor (CMP) architecture offers dramatically faster retrieval of shared data which is cached on-chip rather than in an off-chip memory. Remote cache requests are handled through a cache coherence protocol. In order to obtain the best possible performance with the CMP architecture, the cache coherence protocol must be optimized to reduce time lost during remote cache and off-chip memory accesses. This paper proposes a novel snooping cache coherence protocol for CMP in which each processor has both private and shared L2 caches. The cache coherence protocol is proven by means of formal verification methods.

1. Introduction

With the emergence of the CMP design paradigm, it has become possible to integrate multiple processors and large caches on a single chip to increase computation performance for a wide variety of workloads. The bus-based shared-memory multiprocessor architecture, formerly developed for the Symmetric Multiprocessor (SMP), is still commonly used for the design of small-scale CMPs. It facilitates the use of snooping protocols and achieves low-latency cache-to-cache misses by broadcasting requests for shared data on the bus. However, in comparison with the SMP, the access time of remote cache requests on CMP is reduced dramatically, and is faster than the off-chip memory access time. Therefore, directly employing the traditional SMP memory hierarchy and coherence protocol with the CMP would diminish its performance advantage. The optimal design of the CMP's on-chip cache presents a key challenge in efficiently using limited on-chip cache capacity while keeping the number of off-chip accesses to a minimum. Some CMP systems employ private L2 caches [1][2] to attain fast average cache access latency by placing data close to the requesting processor. To prevent replication and improve the CMP's performance, IBM Power 4 [3] and Sun Niagara [4] use shared L2 caches to maximize the on-chip capacity. Recently, several hybrid L2 organizations have been proposed to reduce access latency by a compromise between the low latency of private L2 and the low off-chip access rate of shared L2. For instance, CMP-NuRapid [5], Victim replication [6], Adaptive Selective Replication scheme [7], and Cooperative Caching [8] are mainly based on either private L2 or shared L2. These schemes represent modifications to the basic architecture and coherence protocol (snooping or directory-based), however, no formal verification of the modified cache coherence

protocols is available. We propose an alternative novel L2 cache architecture, in which each processor has **Split Private** and **Shared L2** (SPS2). When data is loaded it is located in private L2 or shared L2 according to its state (exclusive or shared). This scheme makes efficient use of on-chip L2 capacity and has low average access latency.

The use of formal verification techniques to prove the functional correctness of CMP cache structures is becoming an increasingly essential part of the design process, particularly since most cache performance improvements come at the cost of greater hardware design and cache protocol complexity. It is especially useful for discovering and correcting any errors at the early stage of protocol design before expending valuable time and resources on a novel strategy. Several formal verification methods have been applied to verify the correctness of some cache coherence protocols [9][10][11].

The principal contributions presented here include: (i) a novel L2 cache architecture - SPS2, and (ii) a novel snooping cache-coherence protocol accompanied with its formal verification method. Due to the distinctive architecture of SPS2, we use a new state graph method to express state transactions in our cache coherence protocol, and verify its correctness and properties using the formal verification method.

2. Cache Architecture

In this paper, we limit our consideration to a 2-level on-chip cache hierarchy, although the proposed SPS2 architecture can be readily extended to include more than 2 levels. A traditional bus-based shared-memory multiprocessor has either private L1s and private L2s, or private L1s and a shared L2. Both cache architectures have their pros and cons. The private L2 architecture has fast L2 hit latency but can suffer from large amounts of replicated shared data copies which reduce on-chip capacity and increase the off-chip access rate. Conversely, the shared L2 architecture reduces the off-chip access rates for large shared working datasets by avoiding wasting cache space on replicated copies. The banked shared L2 cache organization is a well-known method to reduce access latency. However, average L2 access latency is still influenced by relative placement on the die and network congestion.

We propose a new scheme, SPS2, to organize the placement of data. All data items are categorised into one of two classes depending on whether the data is shared or exclusive. Correspondingly, the L2 cache hardware organization of each processor is also divided into two parts, private and shared L2. The proposed scheme places

exclusive data in the private L2 cache and shared data in the shared L2 cache. This arrangement provides fast cache accesses for unique data from the private L2. It also allows large amounts of data to be shared between several processors without replication of the data and thus makes better use of the available shared L2 cache capacity which is distributed between the processors.

The proposed SPS2 cache scheme is shown in Figure 1. Each processor has its own private L1 (PL1), private L2 (PL2) and shared L2 (SL2). The shared L2 is a multi-banked cache that could be accessed by all the processors directly over the bus. The corresponding cache coherence protocol is referred to as the SPS2 protocol. In this paper, we define a node as an entity comprising a single processor and three caches, i.e., PL1, PL2 and SL2. In a physical realization, SL2 could have the same number of banks as the number of processors, where each SL2 bank is physically located close to its respective processor. This regular structure is very amenable to VLSI implementation although other bank arrangements could also be used. The role of the SL2 is to store shared data, while the role of PL2 is to store modified exclusive data. All new data for reading is fetched from memory and simultaneously loaded into PL1 and SL2. Data for writing will be placed in PL1 only. PL1 and PL2 are exclusive, but PL1 and SL2 are inclusive. If a data item exists in PL1 then it cannot also exist in PL2, however, if exclusive data is evicted from PL1, it will be placed in PL2, from where it may later be evicted again back to memory. If a shared data item exists in PL1, then a copy must exist in SL2, however, the existence of a data item in SL2 does not imply that a copy must also exist in a PL1 cache.

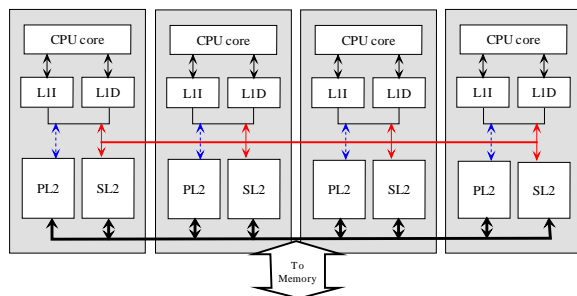


Figure 1. SPS2 cache architecture

Unlike the unified L2 cache structure, the SPS2 system with its split private and shared L2 caches can be flexibly and individually designed according to demand. First, PL2 could be designed as a direct-mapped cache to provide fast access and low power, while SL2 could be designed as a set-associative cache to reduce conflict. Second, PL2 and SL2 do not have to match each other in size, and they could have different replacement policies. We can evaluate the advantages of dividing shared data and modified data. Take for example an application comprising a large quantity of shared data that needs to be used by several threads from different processors and also a large quantity of data that is exclusive to one processor. In the case of a private L2 architecture several copies of the same shared data set will exist in the L2 caches of the different processors. This architecture will suffer from

wasted cache space and incur higher numbers of on-chip misses for large data sets and/or multiple concurrent processes/threads. In the case of a shared L2 architecture, any processor can access all of the shared data, however, since many of the requested data blocks would not be available in the local bank, it will result in high access latencies while blocks are fetched from other banks. In the case of the SPS2 architecture, each processor has two separate L2 caches (PL2 and SL2), which could be individually and simultaneously accessed. This scheme reduces access latency and contention between shared data and exclusive data. It imposes a low L2 hit latency because most of the exclusive data should be found in the local PL2. Shared data will be placed in SL2 which collectively provide high storage capacity to help reduce off-chip access. The SPS2 cache system does not need any new additional CPU instructions to support its protocol, and hence, no changes to the instruction-set or CPU hardware interface are required to enable the cache system to work with a conventional multiprocessor design. The only parts that need to be modified comprise the cache architecture and the cache controller that includes the realization of cache coherence protocol.

3. Description of Coherence Protocol

The protocol employed in SPS2 is based on the MOSI (Modified, Owned, Shared, Invalid) protocol and introduces no new states. Data contained in PL1 may have all four possible states (M, O, S, I), while shared data contained in SL2 has only two states (S, I), and evicted data contained in PL2 has three possible states (M, O, I). The SPS2 protocol uses the write-invalidate policy on write-back caches. To keep consistency and coherency between the three different caches, the cache coherence protocol should also be modified accordingly.

The protocol works roughly as follows. Initially, any data entry in the three caches (PL1, PL2 and SL2) should be Invalid (I). When node i makes a read access for an instruction or data block at a given address, $PL1_i$ will be searched first. Since $PL1_i$ is empty, then $PL2_i$ and $SL2$ will be searched next. Again, neither $PL2_i$ nor $SL2$ have the requested data, so a GetS message will be sent on the bus. Since all the caches in all the processors are initially invalid, the memory will put the data on the bus, and $SL2$ and $PL1_i$ will both store the data and change their states from I to S. If another node j requires this same data shortly after, the data will be copied from $SL2$ to $PL1_j$ without needing to fetch the data from memory or to place another copy in $PL2_j$. If a read request finds the data in the local $PL1_i$, then no bus transaction is needed and data will be supplied to the processor directly.

When node i needs to make a write access and a write miss is detected because the data block is not present in $PL1_i$, $PL2_i$, and $SL2$, the cache controller starts a block replacement cycle, possibly evicting a dirty block from the cache. A GetX message will be sent on the bus to fetch data from the other nodes or memory and place the requested data in the recently vacated slot. All the other nodes will check their own PL1 and PL2 caches for the requested data. If any node, for example j , finds Modified

(M) data with same address as the requested data, the contents will be sent to PL1_i and all the caches (including *j* and excluding *i*) should invalidate data with this address. Once the data is placed in PL1_i and updated, its state will be changed to M. Since the SPS2 scheme employs a write-back policy, modified data will not be written back to memory until it is replaced. A data item with state M means this node exclusively retains the most recent version of the data and that the data in memory is obsolete. If, conversely, the write operation finds the data block in PL1_i or PL2_i with state M (implying a write hit), then the write process will proceed with no bus transactions involved.

Suppose that after node *i* executes a write command, another node *j* needs to read data or fetch instruction from same address, it will check PL1_j first, then PL2_j and SL2. Since any copy that node *j* may have previously held would have been invalidated by the write operation from node *i*, it will be unable to find the data locally. Therefore a GetS message will be placed on the bus requesting the other nodes to send back the data. Node *i* will check its own PL1_i and PL2_i, and find the requested data block with state M in PL1_i. The modified data will then be placed on the bus and stored in both SL2 and PL1_j. The cache state in PL1_i will be changed from M to O and that in PL1_j will be set to S.

If a free slot is not available in any of the caches, then the existing data block will need to be swapped out and replaced with the new block. The old data block in PL1 will be evicted to PL2, if its state is M or O. Any data evicted from PL2 will be written back to memory. If the state of the data evicted from PL1 is S, which means that SL2 must have a copy, then the data need not be copied back to SL2, but simply invalidated. When data in SL2 is evicted, write back is not needed.

4. Formal Verification of Cache Coherence Protocol

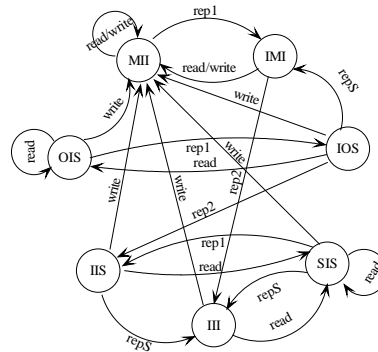
Cache coherence is a critical requirement for correct behavior of a memory system. The goal of a formal protocol verification procedure is to prove that it adheres to a given specification. The specification is a list of correctness properties required from the protocol. The cache protocol verification procedure includes checking for data consistency, incomplete protocol specification, and absence of deadlock and livelock. Using formal verification in the early stage of the design process is helpful in finding consistency problems and eliminating them before committing to hardware.

To maintain data consistency between caches and memory, each node is equipped with a finite-state controller that reacts to the read and write requests. Abstracting from the low-level implementation details of read, write, and synchronization primitives, one may consider cache coherence protocol as families of identical finite-state machines. The following section will illustrate how the SPS2 protocol works using a state machine description.

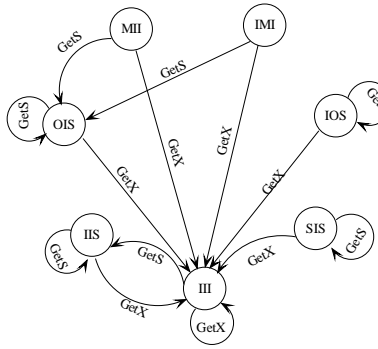
4.1 State graph of SPS2 cache protocol

Each node in the SPS2 architecture has three caches

(PL1, PL2, and SL2), each of which has its own state representation. These three states can be concatenated to form a single vector {XYZ} representing the state of a PL1 cache block in a node. X indicates the state of a PL1 cache block with four possible states (I, S, M, O), Y indicates the state of a PL2 cache block with three possible states (I, M, O), and Z indicates the state of a SL2 cache block which has only two states (I and S). Therefore, for each node a cache block could have up to $4 \times 3 \times 2 = 24$ different states, although some of these states are unreachable. For example, SMS is an impossible state, because if PL2 has a modified block, it is impossible for both PL1 and SL2 to have shared copies of this block. Another impossible state for a data block is IMS, which implies that the block in PL1 is invalid, while PL2 owns a modified version, and SL2 has a shared copy. Excluding the set of invalid states, there are only seven possible states, i.e., III, IIS, SIS, MII, IMI, OIS, and IOS. III is the initial state of each data block.



(a) State transition graph (from processor perspective)



(b) State transition graph (from bus perspective)

Figure 2. State transition graph of SPS2 cache protocol

Our coherence protocol requires two different sets of commands. All the transition arcs in Figure 2(a) correspond to access commands issued by a local processor. These commands are labeled as *read*, *write*, and replacement (*rep1*, *rep2* and *repS*). All the arcs in Figure 2(b) correspond to commands issued by processors via the snooping bus. They include *GetS*, *GetX*, *inv*, *invS*. All these commands are defined below:

- read*: issued when a processor needs to fetch instruction or load data;
- write*: issued when a processor needs to write data;
- rep1*: issued when PL1 needs room for new data;
- rep2*: issued when PL2 needs room for new data;
- repS*: issued when SL2 needs room for new data;

GetS: issued when requesting other nodes or memory to share data, following a read miss;

GetX: issued when requesting other nodes or memory for exclusive data, following a write miss;

inv: issued to invalidate copies in other nodes when a local write access results in a hit.

invS: issued to invalidate other copies in PL1 when a cache block in SL2 is replaced.

As shown in Figure 2, the cache state of any node will change to the next state according to its current state and the received command.

4.2 Verification using HyTech

One of the new techniques described in the research literature relating to formal verification of coherence protocols, is to validate protocols independent of the number of processors in the system. This technique is referred to as parameterized verification. In this section, HyTech [12], an abstraction-level model checker, is used to verify the SPS2 cache coherence protocol.

The first step is to define the SPS2 protocol using a finite-state machine model. According to [13][14], we limit ourselves to consider protocols controlling single memory blocks and single cache blocks although the procedure could be easily extended to encompass the whole memory and cache system.

Let k be the number of nodes of a given multiprocessor system. The behavior of the caches N_i in node i is represented as a finite system $\langle Q, \Sigma_i, f_i, \delta_i \rangle$ where Q is a finite set of states of a node (such as IIS, IOS, etc.), Σ_i is the set of operations (*read*, *write*, *GetS*, *GetX*, etc.) causing state transitions, $f_i: Q^k \rightarrow \{true, false\}$ is the characteristic function from the perspective of N_i , and δ_i defines the state transition $Im(f_i) \times Q \times \Sigma_i \rightarrow Q$ (where $Im(f_i)$ is the image of f_i).

Similarly, we could use EFSM (extended finite-state machine) [14] to model parameterized cache coherence protocol. The behavior of system is modeled as the global machine $M_G = \langle Q_G, \Sigma_G, F, \delta_G \rangle$ which is associated with protocol P , where $Q_G = \{s_1, \dots, s_n\}$. s_i is the possible states of cache blocks in one node. $\Sigma_G = \bigcup_{i=1}^k \Sigma_i$, $F = \langle f_1, \dots, f_k \rangle$,

$\delta_G: Im(F) \times Q_G \times \Sigma_G \rightarrow Q_G$. We model M_G via an EFSM with only one location and n data variables $\langle x_1, \dots, x_n \rangle$ (denoted as x) ranging over the set of positive integers. For simplicity, location could be omitted, hence the EFSM-states are tuples of natural numbers $\langle c_1, \dots, c_n \rangle$ (denoted as c) where n_i denotes the number of nodes in states $s_i \in Q$ during a run of M_G . Transitions are represented via a collection of guarded linear transformations defined over the vector of variables $\langle x_1, \dots, x_n \rangle$ (denoted as x) and $\langle x'_1, \dots, x'_n \rangle$ (denoted as x'), where x_i and x'_i denote the number of nodes in state s_i , respectively, before and after the occurrence of an event. Transitions have the form $G(x) \rightarrow T(x, x')$, where $G(x)$ is the *guard* and $T(x, x')$ is the *transformation*. The transformation $T(x, x')$ is defined as $x' = M \cdot x + c$ where M is an $n \times n$ -matrix with unit vectors as columns. Since the number of nodes is an invariant of the system, we require the transformation to satisfy the

condition $x_1 + \dots + x_n = x'_1 + \dots + x'_n$.

The following gives an informal definition of how the transitions of a cache coherence protocol can be modeled via guarded transformations.

- ♦ *Internal action*. Caches in a node move from state s_1 to state s_2 : $x_1' = x_1 - 1$, $x_2' = x_2 + 1$ with the proviso that $x_1 \geq 1$ is part of $G(x)$. For example, a read miss makes the state of a node move from IIS to SIS.
- ♦ *Synchronization*. Two nodes synchronize on a signal: a node N_1 in state s_1 changes to state s_2 , and another node N_2 in state s_3 changes to state s_4 . This is modeled as $x_1' = x_1 - 1$, $x_2' = x_2 + 1$, $x_3' = x_3 - 1$, $x_4' = x_4 + 1$, with the proviso that $x_1 \geq 1$, $x_3 \geq 1$ is part of $G(x)$. For instance, a read miss may not only make a node change from III to SIS, but also make another node change from MII to OIS.
- ♦ *Re-allocation*. The state of all nodes C_1, \dots, C_k is a constant number λ of nodes whose state changes to C_z for $z > k$ and to state C_i for $i > k$: $x_1' = 0, \dots, x_k' = 0$, $x_1' = x_1 + \dots + x_k - \lambda$, $x_z' = \lambda$. This feature can be used to model bus invalidation signals.

Some of the transition rules are listed in Figure 3. Because SPS2 protocol has seven different states (III, IIS, SIS, MII, IMI, OIS, and IOS), we use these seven variables of integer type to indicate seven states respectively. Rule $r1$ corresponds to a *read hit* event: i.e., no action is needed; and rules $r2 - r7$ correspond to *read miss* events. For the sake of brevity, other events (such as write hit, write miss, replacement, etc.) are omitted in Figure 3.

<pre>(r1) SIS+OIS+MII ≥ 1 → — (r2) III ≥ 1, MII=0, IMI=0 → III'=0, SIS'=SIS+1, IIS'=IIS+III-1 (r3) III ≥ 1, MII ≥ 1 → III'=0, SIS'=SIS+1, MII'=MII-1, OIS'=OIS+1, IIS'=IIS+III-1 (r4) III ≥ 1, IMI ≥ 1 → III'=0, SIS'=SIS+1, IMI'=IMI-1, IOS'=IOS+1, IIS'=IIS+III-1 (r5) IIS ≥ 1 → IIS'=IIS-1, SIS'=SIS+1 (r6) IMI ≥ 1 → MII'=MII+1, IMI'=IMI-1 (r7) IOS ≥ 1 → OIS'=OIS+1, IOS'=IOS-1</pre>

Figure 3. EFSM for SL2 protocol

Possible sources of data inconsistency are outlined below.

(i) $OIS \geq 1$ & $MII \geq 1$. This indicates that data is inconsistent if a node with state OIS coexists with other cache blocks in other nodes with state MII. OIS implies that PL1 modified the data and has ownership, and that other nodes share this data through SL2. This is contradicted by the corresponding block label MII for another node which informs that PL1 has modified data, and that PL2 and SL2 have no valid data. We can see there are two conflicts. The first inconsistency is that if a cache in one node has modified data, no other cache should have valid copies (labeled O or S). The second inconsistency is that since the state of one node is OIS then all the corresponding states of the other nodes could only be IIS or SIS. Since SL2 is a common shared cache, then the state of SL2 should be coherent.

(ii) $OIS \geq 2$. If more than one node has state OIS for

the same block, then data integrity will not hold, because it is impossible for two or more PL1s to own the same block of data.

(iii) $IIS \geq 1 \ \& \ IMI \geq 1$. If a block in one node is shared as indicated by state IIS, then no other cache should have a modified copy, i.e. with state IMI, of the shared block.

In order to verify data consistency all possible sources of data inconsistency must first be defined. As proven in [13] [15][16], whenever both the guards of a given EFSM and the target states are represented via constraints that represent upward-closed sets, a symbolic reachability algorithm using integer constraints for representing sets of states always terminates. In this way we automatically verified the properties of our SPS2 protocol using the HyTech tool.

4.3 Verification using SMV

Computation Tree Logic (CTL) is a very simple subset of model tense logic defined by Clark and Emerson [17]. It can express the mandatory properties of the protocols. It has three components: atomic propositions, boolean connectives, and temporal operators. Atomic propositions talk about the values of individual state variables. The boolean connectives are the standard ones (\wedge, \vee, \neg). Each temporal operator consists of two parts: a path quantifier (A or E) and a temporal modality (F, G, X, or U). Quantifier A indicates that the operator denotes a property that should be true of all execution paths from a given state whereas quantifier E denotes that the property is exclusive to one path. The modalities describe the ordering of events in time along an execution path and have the following intuitive meanings:

(i) $F \phi$ (ϕ holds sometime in the future) is true of a path if there exists a state on the path for which the formula ϕ is true.

(ii) $G \phi$ (ϕ holds globally) means that ϕ is true for every state along the path.

(iii) $X \phi$ (ϕ holds in the next state) means that ϕ is true in the second state along the path.

(iv) $\phi U \psi$ (ϕ holds until ψ holds) means that there exists some state along the path for which ψ is true, and for all preceding states, ϕ is true.

The following examples illustrate the expressive power of the logic.

(i) $AG (req \rightarrow AF ack)$: it automatically follows that if the signal *req* is high, then eventually *ack* will also be high.

(ii) $AG AF enabled$: enabled holds infinitely on every computation path.

(iii) $AG EF restart$: it is possible to reach the *restart* state from any state.

(iv) $AG (send \rightarrow A(send U recv))$: it is always the case that if *send* occurs, then eventually *recv* is true, and until that time, *send* must remain true.

To avoid state explosion, recent model checkers use an implicit representation for finite-state systems based on *ordered binary decision diagrams* (OBDD). SMV (symbolic model verifier) [14] is one such tool for checking that finite-state systems satisfy specifications

given in CTL. Applications of the symbolic model checking method have been used to successfully verify the snooping protocols of the Gigamax [18] and the Futurebus+ [19]. The formal verification of the Stanford FLASH cache coherence protocol via SMV proof assistant is given in [9]. The following section show how SMV is used to model and verify the proposed SPS2 protocol.

4.3.1 Modeling the protocol using SMV

Figure 4 shows a portion of the SMV program used to model the state transactions of a single block in a node in the SPS2 cache system. In lines 1-3, the ASSIGN declaration initially assigns the state of the cache block the value III and then proceeds to assign it the next state depending on the command received. Lines 5-9 reveal that when no command is present the next value of *state* is a random value. Lines 10-27 express that if this node is master, the next corresponding value of *state* differs depending on the command (read, write, GETS, ...). Lines 28-35 reveal the next value of *state* when the current node is not master and command is GETS.

```

ASSIGN                                1
init(state) := III;                   2
next(state) :=                          3
  case                                  4
    CMD=none:                           5
      case                               6
        ...                              7
        1: state;                        8
      esac;                              9
  master:                                10
    case                                  11
      CMD=gets:                           12
        case                             13
          state=III: SIS;                 14
          1: any;                         15
        esac;                             16
      CMD=read:                           17
        case                             18
          state=IIS: SIS;                 19
          state=IOS: OIS;                 20
          state=IMI: MII;                 21
          state in {SIS, MII, OIS}: state; 22
          1: any;                         23
        esac;                             24
      ...                                  25
    1: any;                                26
  esac;                                   27
  CMD=gets:                                28
    case                                  29
      state=III: IIS;                     30
      state=MII: OIS;                     31
      state=IMI: IOS;                     32
      state in {IIS, SIS, OIS, IOS}: state; 33
    1: any;                                34
  esac;                                    35
  CMD=read: state;                        36
  ...                                      37
  1: any;                                  38
  esac;                                    39

```

Figure 4. Modeling SPS2 protocol using SMV

4.3.2. Verifying the protocol

To verify data integrity, we introduce another variable M to indicate if the content of a cache block has been modified. Therefore, we could check whether the bus has had any errors using formulas, such as $AG(CMD=replacement \ \& \ M=1)$. As defined in Section 4.3, this

formula means that it is always true that when the command is *replacement*, M must be high because if this data block has not been modified, there is no need to replace it. Another form of error is processor error which typically reveals when state and command conflict. The specification could be shown in the following formulae:

AG (p1.shared \rightarrow p2.shared)
 AG (p1.state=SIS & p2.state=SIS \rightarrow p1.data = p2.data)
 AG (p1.state=OIS & p2.state=IIS \rightarrow p1.data = p2.data)
 AG (p1.readable & !M \rightarrow p1.data=mem.data)
 AG (p1.excl \rightarrow !p2.readable)

The first formula states that it is always true that if one node has a shared data block, then eventually another node (for example, node 2) will also have this shared data. The second formula declares that if the states of two nodes are both SIS, the value of the data in both nodes must be the same. Similarly, the third formula declares if the state of one node is OIS and the state of another node 2 is IIS, the value of the data in both nodes must be the same. The fourth formula means that if the data in node 1 is readable and not modified, then data should have same value as that stored in the memory. The fifth formula means that if node 1 has exclusive data, then no other node has corresponding data.

Similarly, the SPS2 protocol could be verified for liveness using the following formula by proving that from any state it is possible to get to the states (OIS, MII, SIS) or that the block is readable or writable:

AG EF p1.state=OIS or AG EF p1.state=MII or
 AG EF p1.state=SIS or
 AG EF p1.readable or AG EF p1.writable

The SMV verification process has been used to check 250 OBDD nodes. The safety properties of the SPS2 protocol are thus proved conclusively.

5. Summary

To improve the CMP cache performance, we propose a new cache architecture SPS2 with split private and shared L2 cache, which takes advantages of the low latency of L2P and the high capacity of L2S. We also propose a corresponding SPS2 cache coherence protocol described by means of new state transition graphs in which each node has three states to indicate the states of private L1, split private L2 and split shared L2 respectively. Using the state transition graphs, the functional correctness of coherence protocol is proven through two formal verification methods.

The use of formal design verification methods helps identify coherence problems in the early stage, and provide assurance of the correctness of the protocol before commencing on the hardware development.

References

[1] K. Krewell. UltraSPARC IV Mirrors Predecessor. *Microprocessor Report*, pages 1-3, Nov. 2003.
 [2] C. McNairy and R. Bhatia. Montecito: A Dual-core Dual-thread Intelium Processor. *IEEE Micro*, 25(2): 10-20, March/April 2005.
 [3] K. Diefendorff. Power4 Focuses on Memory Bandwidth.

Microprocessor Report. 13(13): 1-8, Oct. 1999.
 [4] P. Jibgetura. A 32-way Multithreaded SPARC/E processor. In *proceedings of the 16th HotChips Symposium*, Aug. 2004.
 [5] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication and Capacity Allocation in CMPs. In *proceedings of 32th International Symposium on Computer Architecture*, pages 357-368, June 2005.
 [6] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *proceedings of 32th International Symposium on Computer Architecture*, pages 336-345, June 2005.
 [7] B. M. Beckmann, M. R. Marty, and D. A. Wood. Balancing Capacity and Latency in CMP Caches. Univ. of Wisconsin Computer Sciences Technical Report CS-TR-2006-1554, February 2006.
 [8] J. Chang, and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *proceedings of 33th International Symposium on Computer Architecture*, pages 264-2765, June 2006.
 [9] K. L. McMillan, Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *CHARME'01*, Livingston, Scotland, LNCS 2144, Springer-Verlag, 2001, pp.179--195.
 [10] D. J. Sorin, M. Plakai, A. E. Condon, et. al. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transaction on Parallel and Distribution System*. 13(6): 556-578, 2002
 [11] Milo M. K. Martin. Formal Verification and its Impact on the Snooping versus Directory Protocol Debate. In *Proceedings of the 2005 International Conference on Computer Design*. 2005: 543-449
 [12] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In Orna Grumberg, editor, *In Proceedings of 9th Conf. on Computer Aided Verification (CAV'97)*, LNCS 1254:460-463. Springer-Verlag, 1997.
 [13] G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. *12th International Conference 2000*, LNCS 1855, Chicago, IL, USA, pp.53-68.
 [14] K.L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, Ph.D. Dissertation, Carnegie Mellon University, May 1992.
 [15] J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, 352-359, 1999.
 [16] G. Delzanno, J. Esparza, and A. Podelski. Constraint-based Analysis of Broadcast Protocols. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'99)*, LNCS 1683, 50-66, Springer-Verlag, 1999.
 [17] E. M. Clarke and E. A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In Dexter Kozen, editor, *Logic of Programs: Workshop*, LNCS 131, Springer-Verlag, 1981
 [18] K.L. McMillan, and J. Schwalbe, Formal Verification of the Gigamax Cache Consistency Protocol, In *Proceedings of the ISSM Int'l Conf. on Parallel and Distributed Computing*, Oct. 1991.
 [19] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness, Verification of the Futurebus+ Cache Coherence Protocol, In *Proceedings of the 11th Int'l Symp. on Computer Hardware Description Languages and Their Applications*, Apr. 1993.