

# A Low-Cost Cache Coherence Verification Method for Snooping Systems \*

Demid Borodin and B.H.H. (Ben) Juurlink  
Computer Engineering Laboratory  
Faculty of Electrical Engineering, Mathematics, and Computer Science  
Delft University of Technology  
Mekelweg 4, 2628 CD Delft, The Netherlands  
Telephone: +31 15 2786623, Fax: +31 15 2784898.  
E-mail: {demid,benj}@ce.et.tudelft.nl

## Abstract

*Due to modern technology trends such as decreasing feature sizes and lower voltage levels, fault tolerance is becoming increasingly important in computing systems. Shared memory in modern multiprocessor systems is supported by cache coherence mechanisms. The correctness of cache coherence of the system is crucial for the data integrity. This work proposes an error detection scheme for snooping-based cache coherence protocols. For the widely used MESI coherence protocol, the proposed method does not introduce any performance overhead. Only a limited amount of additional hardware is required. Existing systems can be easily extended to support the proposed technique. Almost all single faults that are able to affect data integrity in the system are covered, with the exception of a few very rare cases. Experimental results involving fault injection do not encounter any undetected faults leading to corrupted application output.*

Keywords: error detection, cache, coherence, snooping, safety, low-cost

## 1 Introduction

The importance of *fault tolerance (FT)* of computing systems is increasing instantly nowadays [14]. This is a consequence of the technology trends which try to follow Moore's law. Smaller feature size, greater chip density, and minimal power consumption lead to increasing device vulnerability to external disturbances such as radiation, internal problems such as crosstalk, and other reliability problems. This results in an increasing number of faults, especially transients, in computing systems.

After the switch from tubes to more reliable transistors and until recently, strong FT used to be a requirement only of special-purpose high-end computing systems. The technology reliability was considered sufficient, and only a few FT techniques, such as Error Correcting Codes [13] in memory, were usually used. However, according to [14, 6], the technology trends will pose more and more reliability issues in future. This means, in turn, that FT features are required even in PCs.

Current trends in computer architecture demonstrate an increasing interest in multiprocessor systems [3]. Multiprocessors feature a distributed and/or shared memory hierarchy. To enhance programmability, the details of the memory structure are typically hidden from the application developer, for whom the memory structure appears as a single shared memory accessible from all the nodes. This is achieved by employing cache coherence protocols which guarantee a consistent memory view for different nodes [7].

Cache coherence plays a very important role in multiprocessors, providing data integrity. For example, it guarantees that, when one processor has changed data at a certain memory address, other processors in the system get a fresh copy of these data when they read them. Thus, faulty cache coherence hardware can lead to data integrity violation. In the above example, if the cache coherence hardware fails to notify a reading processor about the changes another processor has made to the data, it will work with wrong (outdated) data, produce other wrong data on this basis, etc.

This work proposes a concurrent error detection technique addressing cache coherence in multiprocessors. The concept of watchdog processors [8] is used. A watchdog processor is an external logic that dynamically verifies the operation of a controlled logic based on certain information received from it. The proposed technique addresses multiprocessors with snooping-based cache coherence. The operation of the coherence logic and storage elements holding coherence states in every cache is dynamically verified

---

\*This work was partially supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

by external logic (*checker(s)*) located on other cache(s) or on special circuit(s) snooping the system network. Each checker has a local store with tag information for all the cache lines residing in the checked cache, but not the data. The checkers snoop all messages on the network and filter those related to the checked cache. Based on these messages and the tag information they contain, the checkers maintain correct state of each cache line, and check the correctness of its state in the checked cache. Wrong states, which might result in violated data integrity in the system, are detected and signaled. In addition to this, the checkers partly verify the operation of the communication logic, making sure, for example, that every request is answered.

The main features that distinguish our proposal from the related approaches discussed in Section 2 are:

- Low or zero performance overhead. There is no performance overhead for the MESI protocol verification discussed in this paper.
- Limited hardware overhead comparable to the cheapest alternatives.
- The design is scalable in that the overhead is linear to the number of nodes in the system.
- It is relatively easy to extend an existing system with the coherence checking. The checkers have to be attached to the network, and a few bits have to be added to the messages.
- The benefits come at the price of a reduced fault coverage in comparison with some other approaches which introduce considerably more overhead.

Section 2 describes related work and compares it with the proposed method. Section 3 provides the details of the proposed technique, discusses possible design options, and gives an example implementation for the MESI coherence protocol. Section 4 analyzes the fault coverage of the proposed method, and presents the experimental results. Section 5 draws conclusions and discusses future work.

## 2 Related Work

Several dynamic cache coherence verification techniques have been proposed. All of them introduce a certain hardware overhead needed for the additional logic. Most techniques increase the communication network bandwidth requirements, which means either a performance penalty or a hardware overhead. Unlike previous proposals, our method introduces almost no network bandwidth overhead (only a few bits must be added to each message). However, this is achieved at the expense of a limited coverage of network faults that can lead to global errors (conflicts in coherence

states of different nodes). We rely on the communication network to provide a reasonable robustness. In addition, in comparison with existing methods, ours requires minimal effort for incorporating coherence errors detection into a system. The cache controllers have to be extended to add the required bits to the coherence messages, and the checkers have to be connected to the network.

Cantin, Lipasti and Smith [4] proposed a method based on the watchdog processor concept. A checker circuit is placed on each cache and implements a simplified version of the coherence protocol used. The checker maintains its own copy of the tags. When a state of a cache line changes, the necessary information is sent to the checker. The checker recomputes the coherence transaction and verifies the states delivered by the cache. In addition to a local, the checker performs a global verification. The new state is broadcast to the checkers of other caches in the system, using a special logical network. The other checkers make sure that the new state of the broadcasting cache does not conflict with their own states.

Compared to [4], our scheme requires significantly less hardware and performance overhead. In [4], every state change requires the controller to submit the new state and some other related information to the checker. We only require the controller to add a few bits to the broadcast messages. Furthermore, the global verification in [4] requires a dedicated logical network. This network can use the existing hardware communication resources, increasing the network traffic and potentially introducing performance overhead. Alternatively, this network can use additional, dedicated hardware resources, increasing the cost. Our approach does not feature a global verification. Consequently, no hardware and/or time is required for that. In addition, unlike [4], our approach not only verifies the state transitions of cache lines, but also makes sure that appropriate messages appear on the network.

Meixner and Sorin [10] developed the Token Coherence Signature Checker (TCSC) scheme. TCSC addresses memory systems utilizing Token Coherence [9], but can also be applied to the traditional invalidation-based snooping and directory coherence protocols, if they are interpreted in terms of token coherence. TCSC keeps two signatures on every cache and memory controller. One signature represents the history of coherence states for all the blocks in cache or memory, and the other incorporates the history of data values stored. Periodically these signatures are sent to the verifier(s). The verifier determines if any conflicting coherence states appeared in different places and if data propagated incorrectly. TCSC requires additional information to be included in the coherence messages and introduces extra traffic by submitting the signatures to the verifier. Implementing the snooping MOSI protocol requires even more additional messages. TCSC introduces a theoretical worst-

case bandwidth overhead of 4.54% for the Token Coherence, 10% to 5% (with some optimizations) for snooping-based protocols, and 15% to 10% for directory protocols. TCSC also requires additional hardware to compute the signatures and for the verifier(s).

TCSC can be considered more general than our scheme, targeting not only snooping-based coherence protocols. As [4], TCSC performs global conflicts checking, while we do it only to a limited extent. However, our scheme introduces almost no network bandwidth overhead, which means a minimal cost and no performance overhead. TCSC requires verifiers, which can become a bottleneck in systems with a large number of caches. In this case, TCSC adds additional verifiers, possibly creating hierarchies of them. The hardware overhead of our approach is always linear in the number of caches. However, in our scheme, like in [4], the hardware overhead per cache depends on the size of the cache, because the state of the cache lines is kept in the checkers. In TCSC the overhead per cache does not depend on the size of the cache.

In a work prior to TCSC, Sorin, Hill, and Wood [15] computed signatures locally and submitted them periodically to a central verifier, similar to TCSC. They used a different algorithm for computing the signatures and had different signatures for the coherence and messages history. The technique is limited to snooping systems and provides slightly less fault coverage than TCSC, but introduces less network bandwidth overhead. Our method requires even less bandwidth overhead at the expense of a weaker interconnect fault coverage.

Fernandez-Pascual, Garcia, Acacio, and Duato [5] proposed a fault tolerant token coherence protocol capable of tolerating transient errors in the CMP interconnection network. This is achieved by running several different timers that start the “token recreation” process when a corresponding timeout is detected. Unlike this work, we mostly target the faults in the cache coherence logic, not in the interconnection network.

### 3 Cache Coherence Verification

This section presents a design implementing the proposed technique. General design decisions, which are suitable for many snooping-based coherence protocols, and their motivation are described in Section 3.1. A detailed implementation example, specific for the widely used MESI coherence protocol, is discussed in Section 3.2.

#### 3.1 General Design Decisions

The cache coherence verification technique proposed in this work addresses multiprocessor systems with a number

of nodes, each with a private cache, connected with a network, such as a bus. Coherence is achieved by means of a snooping-based protocol [7], such as MESI [11]. The checkers can reside on other than the controlled caches, or on separate bodies connected to the network. In the first case, some logic processing the incoming messages can be shared between the checker and its host cache, reducing the hardware overhead introduced by the checkers. Thus, this configuration is preferred. Figure 1 depicts the proposed system structure with a bus as the interconnection network. Note that in Figure 1, cache number  $i$  incorporates a checker for the cache number  $i-1$ . However, this is not necessary, any distribution of the checkers is possible. Later it will be shown that placing checkers as far as possible from the controlled caches is even desirable for increasing the network fault coverage.

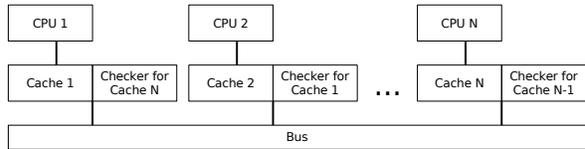


Figure 1. Proposed system structure

A checker needs some information about each line of the controlled cache, such as its block address and current coherence protocol state. Hence, the checker keeps the tag of each line, but not the data. Keeping the data provides no benefit. It implies a large hardware overhead, but does not help in error detection. Verifying the data would require every local processor write to be broadcast on the network (to inform the checker), which implies a network traffic overhead that we do not consider feasible. It is more practical to apply conventional protection methods to the data, such as error detection/correction codes [13].

The controlled cache could notify the checker about all state transformations through a special connection between them. However, this would require special logic added to the cache, as well as a dedicated communication line between the cache and its checker. These communication lines would complicate the final design, requiring to place and route them, and prevent locating checkers on arbitrary places. Furthermore, adding extra hardware not only increases the system cost, but also introduces new potential faulty points. To minimize the hardware and performance overhead, we aim at designing the checkers as autonomous as possible. Any special communication between the caches and the checkers is avoided. The checker only snoops the network, processing all the messages related to the data in the controlled cache. Based on this information, it maintains its own states, and verifies the states on the controlled cache. A mismatch of the checker’s and the con-

trolled cache's states for a cache line signals an error. In addition to this, by snooping the messages from other caches, the checker makes sure that the states in other caches do not conflict with the states on the controlled cache. Besides the states, the checkers verify the network transactions, initiated both by the controlled cache and the rest of the system (addressed to the controlled cache). For example, it makes sure that every memory request of the controlled cache is answered, that the controlled cache does not attempt to write back an invalid cache line, etc. Thus, by connecting the checkers to the system network instead of a dedicated communication line, we gain some additional control of the network interface (both of the checked cache and of other entities communicating with it), and the network itself.

The effectiveness of the proposed checker design can be compared to the traditional logic duplication with comparison of the results. Inside the cache, the logic implementing coherence, the coherence protocol state and tags could be duplicated. Every state transition would require an agreement between the replicated logic. The independent checker has several advantages over this scheme. An independent checker is most probably physically located further away on the die (or on a different die) from the checked logic. This reduces the probability of common faults. A duplicated logic without modifications inherits possible design faults. The checker should be designed independently, thus it has the potential to detect design and implementation faults. Furthermore, as discussed earlier, the checker can implement additional functionality, such as verification of the network operation. The price to be paid for these advantages is a possible increase of hardware cost (if the checker needs more logic than the duplicated version). If the design requires additional network traffic, a certain performance overhead is also likely to appear.

The proposed technique is able to detect errors in the cache coherence logic and in the storage elements where the current states are kept. Upon error detection, depending on the design, the system can be halted, a graceful degradation may take place, or a recovery may be initiated. In the case of graceful degradation, the erroneous node is switched off in a multiprocessor environment, and its task is reassigned to a different processor. Optionally, assuming the fault could be transient, the node can be switched on again and tested for permanent faults. If the system supports a recovery mechanism, such as [16, 12], it can restart operation from the last state which is known to be correct. Note that the technique as presented in this paper does not favor recovery, because the error detection latency is not fixed. Errors are detected when a message holding the erroneous state appears on the network. Checkpointing requires that at some moment the system state is known to be correct. Additional activities are required for this in the proposed scheme. For example, the caches can send the current states of all the lines to the

checkers for verification. If the verification result is positive, a checkpoint can be created.

The proposed technique relies on the correctness of the system network. Faults leading to network messages loss, for example, are not reliably covered. If a request from one cache neither reaches another cache nor its checker, the error is not going to be detected, unless it will lead to further conflicting coherence actions. This is another reason why it is better to place checkers further away from the checked caches in the network: the chance that a lost message will reach at least one of them increases.

### 3.2 Protocol-Specific Implementation

In this section the proposed idea is applied to the widely used MESI cache coherence protocol [11]. The MESI protocol features four possible states for a cache line: *Modified (M)*, *Exclusive (E)*, *Shared (S)*, and *Invalid (I)*. In the *M* state, a valid copy of the line is present only in the owner's cache. The local processor has modified the line and can write it again without notifying the other nodes, so it should be invalidated in all the other caches. The *E* state means that both the memory and the cache have a valid copy, but other caches do not. The local processor can write it, switching its state to *M*, without notifying other caches. The *S* state indicates that the line is present in multiple caches and in memory. The caches use it in read-only mode. A write access requires sending a notification (invalidation) to other nodes. The *I* state signals that the data in the line are invalid.

The implementation of the MESI protocol which is discussed in this work, with the required state transitions in response to different requests, is demonstrated in Table 1. The first column contains the request, coming either from the local processor or from the network. The following requests and network messages appear:

- *PrRd* – read request from the local processor.
- *PrWr* – write request from the local processor.
- *BusRd* – read request snooped from the network.
- *BusRdX* – read exclusive request from the network.
- *BusWB* – write back from a cache.
- *Flush* – flush request snooped from the network.

The second column defines the current state at which the request is received. The third column shows the resulting state of the transition induced by the request. The fourth column lists messages that the cache sends to the network in response to the received request. The fifth column defines the corresponding actions required from the coherence checker of this cache. The following syntax is used:

**Table 1. MESI protocol state transitions and corresponding actions of the cache and its checker**

Request	Current State	Next State	Bus Message	Checker Activity
<i>PrRd</i>	<i>M</i>	<i>M</i>	-	-
<i>PrRd</i>	<i>E</i>	<i>E</i>	-	-
<i>PrRd</i>	<i>S</i>	<i>S</i>	-	-
<i>PrRd</i>	<i>I</i>	<i>E/S</i>	<i>BusRd</i>	$== I; \text{Answer}; \rightarrow E/S$
<i>PrWr</i>	<i>M</i>	<i>M</i>	-	-
<i>PrWr</i>	<i>E</i>	<i>M</i>	-	-
<i>PrWr</i>	<i>S</i>	<i>M</i>	<i>Flush</i>	$== S; \rightarrow M$
<i>PrWr</i>	<i>I</i>	<i>M</i>	<i>BusRdX</i>	$== I; \text{Answer}; \rightarrow M$
<i>BusRd</i>	<i>M</i>	<i>S</i>	<i>BusWB</i>	$\rightarrow S; \text{Answer}$
<i>BusRd</i>	<i>E</i>	<i>S</i>	<i>BusWB</i>	$\rightarrow S; \text{Answer}$
<i>BusRd</i>	<i>S</i>	<i>S</i>	<i>BusWB</i>	<i>Answer</i>
<i>BusRd</i>	<i>I</i>	<i>I</i>	-	<i>No Answer</i>
<i>BusRdX</i>	<i>M</i>	<i>I</i>	<i>BusWB</i>	$\rightarrow I; \text{Answer}$
<i>BusRdX</i>	<i>E</i>	<i>I</i>	<i>BusWB</i>	$\rightarrow I; \text{Answer}$
<i>BusRdX</i>	<i>S</i>	<i>I</i>	<i>BusWB</i>	$\rightarrow I; \text{Answer}$
<i>BusRdX</i>	<i>I</i>	<i>I</i>	-	<i>No Answer</i>
<i>Flush</i>	<i>M</i>	-	-	<i>Signal Error</i>
<i>Flush</i>	<i>E</i>	-	-	<i>Signal Error</i>
<i>Flush</i>	<i>S</i>	<i>I</i>	-	$\rightarrow I; \text{No Answer}$
<i>Flush</i>	<i>I</i>	<i>I</i>	-	<i>No Answer</i>

- $== \text{state}$  – make sure the current state is *state*.
- $\rightarrow \text{state}$  – change the state to *state*.
- *Answer* – make sure the request is answered.
- *No Answer* – make sure request is not answered.
- *Signal Error* – report an error and stop operation.

Local processor read requests (*PrRd*) for cache lines in any state except *I* are serviced within the node and do not induce any network activity. They also do not change the states, thus the checker does not need to be notified about these events. *PrRd* for an invalid line leads to a *BusRd* requesting the line from the memory. Thus, when the checker snoops a *BusRd* request from the checked cache, it knows that this request can only appear for an invalid cache line, and checks if the corresponding line is in the *I* state. Further, the checker makes sure that the request is answered. The architecture discussed here uses the conventional approach in which other caches answer read requests if possible. If other caches do not contain the requested data, the

memory answers. If the *BusRd* request is answered by the memory, the cache line is not present in other caches, so its state becomes *E*. Otherwise, other caches share the line, so the state becomes *S*.

Local processor writes (*PrWr*) to lines in the state *M* do not alter the state and do not produce any bus messages, thus the checker is idle. *PrWr* to a line in the state *E* is the only event which changes the state (to *M*), but which is not associated with any network traffic. The checker can, therefore, not know about this transition without a special notification from the checked cache. Thus, a design decision has to be made in this case. The checked cache could broadcast a special notification message for the checker. However, this implies that the cache controller has to be redesigned, and some additional network traffic appears, which could possibly degrade performance if the network throughput is an issue. There exists another solution, acceptable in this case. The *E* to *M* transition is considered safe, and it is not communicated to the checker. This is reasonable because an erroneous *E* to *M* transition will never affect the data integrity in the system, as can be seen from Table 1. Thus, to

minimize the performance and cost overhead, this transition is allowed. If, after a silent  $E$  to  $M$  transition in a cache, its checker receives a line in state  $M$  instead of the expected  $E$ , it does not signal an error, but changes the state to  $M$  to reestablish coherence with the checked cache. If an external request arrives for a line whose  $M$  and  $E$  states are not synchronized between the cache and the checker yet, it is not problematic. As can be seen from Table 1, the checker actions in response to external requests  $BusRd$ ,  $BusRdX$  and  $Flush$  are the same for the states  $M$  and  $E$ . For the external read requests, the coherence states are changed to the same one ( $S$  in the case of  $BusRd$  request and  $I$  in the case of  $BusRdX$  request), thus the cache and the checker become coherent again. Hence, no error will be reported or propagated.  $PrWr$  on a line with the  $S$  state changes the state to  $M$  and broadcasts an invalidation ( $Flush$ ) request on the bus. Hence, when a checker snoops a  $Flush$  request from the checked cache, it makes sure that the old state is  $S$  (there are no other states that produce the  $Flush$  request), and updates the state. When a local write miss happens, the cache sends the  $BusRdX$  request, to receive the necessary line and invalidate it in other caches. In this case the checker makes sure the current state is  $I$ , that an answer arrives, and changes the state to  $M$ .

When the checker snoops a read request ( $BusRd$ ) for a line which is present and valid in the checked cache, it changes the state to  $S$  if necessary, and makes sure that the checked cache answers with a  $BusWB$  message. If the requested line is invalid in the checked cache, the checker makes sure it does not answer. On a  $BusRdX$  request for a valid line, the checker invalidates the line, and makes sure the checked cache sends a  $BusWB$  answer.  $BusRdX$  for an invalid line should produce no answer from the checked cache.

Upon receiving a  $Flush$  request from the network, the checker invalidates the corresponding line if it is in the state  $S$ , and makes sure the checked cache does not answer. If a  $Flush$  request is received for a line in the  $M$  or  $E$  state, an error is reported, because it is an illegal situation: only lines in the  $S$  state can send a  $Flush$  request.

Table 1 demonstrates the checker's actions corresponding to MESI transitions in the checked cache. It only partially specifies the actual checker implementation, and does not reflect some details like cache line evictions induced by local misses. A full event-driven implementation specification is given in Table 2. The checker snoops a network message, and if it relates to data contained in the checked cache, classifies it as a checked cache request, answer for the checked cache, or request from another cache. It is further processed depending on the particular message, as shown in Table 2. The following additional actions complete the specification of the checker:

- for all messages (requests and answers) snooped from

the checked cache, the checker verifies the current state of the line, if available. This is the action which actually detects illegal state transitions and reports errors.

- the checker makes sure that every request from other caches, if it hits, is answered by the checked cache.
- the checker makes sure that the checked cache never answers with invalid data (lines with the state  $I$ ).
- the checker makes sure that every answer to the checked cache comes in response to a corresponding request.

There are several necessary design decisions not mentioned so far. To be able to check the current state of a line in the checked cache, the checker needs to receive this state. This could be achieved by letting the cache send notifications on every state transition to its checker. However, as discussed earlier, for optimization reasons this is avoided. In this work the problem is solved by extending the network message with the current state of the addressed cache line. All messages sent by the cache include the current state information of the addressed line, which is used by the checker.

Another problem appears if the cache is not direct-mapped, but features associativity. Upon a miss, the checker needs to know where the incoming data will be placed, to maintain the correct information about the lines in the checked cache. The set can be easily determined by the address which is present in the request. However, depending on the replacement policy used, the way number is determined randomly or based on the access history. For line replacement policies such as Least Recently Used (LRU), the checker needs the whole cache access history to determine the way which will be used. It is not feasible to send all this information from the cache, thus another solution is needed. In this work it is solved by further extending the network message with the way number of the cache line.

Thus, to support the proposed technique, the network message format is extended with 2 bits for the current MESI state, and a certain number of bits for the way where a cache line resides (depends on the cache associativity). For example, 1 bit is sufficient for a 2-way set-associative cache. Hence, a total of 3 additional bits are added to the message for the MESI protocol and 2-way set-associative caches.

## 4 Fault Coverage

This section discusses the fault coverage of the proposed coherence verification method. The MESI-specific design presented in Section 3.2 is considered. Section 4.1 analytically evaluates the fault coverage, and Section 4.2 presents experimental results.

**Table 2. Checker actions in response to snooped bus traffic**

<b>Bus Message</b>	<b>Checker Activity</b>
<p><b>Requests from the checked cache:</b></p> <p><i>BusRd</i> or <i>BusRdX</i></p> <p><i>BusWB</i></p> <p><i>Flush</i></p>	<p>== <i>I</i>: the line is either already invalid, or has previously been evicted and invalidated <i>Answer</i>.</p> <p>→ <i>I</i>: this should be an eviction. If another request follows, make sure it is <i>BusRd</i> or <i>BusRdX</i>.</p> <p>== <i>S</i>; → <i>M</i>.</p>
<p><b>Answers for the checked cache:</b></p> <p><i>BusRd</i> answer from memory</p> <p><i>BusRdX</i> answer from memory</p> <p><i>BusWB</i> as an answer from another cache</p>	<p>→ <i>E</i>.</p> <p>→ <i>M</i>.</p> <p>If <i>BusRd</i> was requested, → <i>S</i>. If <i>BusRdX</i> was requested, → <i>M</i>.</p>
<p><b>Requests from other caches:</b></p> <p><i>BusRd</i></p> <p><i>BusRdX</i></p> <p><i>Flush</i></p>	<p>If state is <i>I</i>: <i>No Answer</i>. If state is not <i>I</i>: → <i>S</i>; <i>Answer</i>.</p> <p>If state is <i>I</i>: <i>No Answer</i>. If state is not <i>I</i>: → <i>I</i>; <i>Answer</i>.</p> <p>== <i>S</i> or <i>I</i>; → <i>I</i>; <i>No Answer</i></p>

#### 4.1 Analytical Evaluation

Assuming an error-free communication network, the proposed method detects all coherence errors visible to the checker. Note that by coherence errors we mean illegal changes of the coherence protocol states in cache lines. These errors can be due to faults in the new state generation logic, or faults in the storage elements holding the current states. In addition, some communication errors such as the absence of answers to issued requests are covered.

A coherence state error is visible to the checker when the controlled cache broadcasts any message related to the corresponding cache line, either a request or a reply. The message contains the current coherence state of the line, which is verified by the checker. In addition, the absence of an expected reply to requests from other nodes in the system can signal errors. For example, an illegal transition of a valid line to the *I* state will lead to the absence of a reply to a read

request. Thus, coherence state errors are not detected in the lines which, after the errors appear, never send and receive requests, and are never written back to the memory.

Consider a cache line which, after its coherence state becomes erroneous, is never addressed by requests from other nodes in the system and is never evicted. When the node's task execution completes, this line will be written back only if it has been changed by the local CPU, i.e. if its current state is *M*. The lines in other states do not have to be written back, because they have a valid copy in the memory. In this situation, the erroneous *M* state will be detected by the checker, when the line is written back. Illegal transitions from the state *M* to another state will also be detected, because it does not write back, as expected. Illegal transitions from any state to *I* will also be detected, if the local processor tries to access the line, because the corresponding read request will be issued to the memory system. If the local processor does not access the line anymore, it has no

influence on the data integrity, and the error can be safely ignored. Only the illegal transitions from a non- $M$  state to the states  $E$  and  $S$  are not covered so far. Their possible combinations and the probable consequences are listed below:

- **Illegal transition from  $E$  to  $S$ .** This transition is harmless, because the local processor still reads valid data from the line, and does not write to this line.
- **Illegal transition from  $S$  to  $E$ .** This transition is also harmless. The local processor still reads valid data from the line. If the line is written, however, it will silently change the state to  $M$ . Then, the line in the state  $M$  will eventually try to write back, and the checker will detect the error.
- **Illegal transition from  $I$  to  $E$  or  $S$ .** These transitions are the most dangerous for the proposed method. They can lead to local reads receiving garbage. On the base of this garbage other (correct) cache lines can be updated, and the data integrity will be violated. This case will not be detected if the line is never evicted or referenced again from outside. Experimental results in Section 4.2 demonstrate that this rare situation is not likely to appear in practice.

To summarize, the proposed method covers all harmful single errors in cache coherence states, except illegal transitions from state  $I$  to state  $E$  or  $S$  in cache lines which are never referenced from the memory system and are never evicted later. Many multiple errors are also covered, except those that assign back the correct state to the cache line before it is checked.

The undetectable single error cases described above are due to the fact that the cache never broadcasts any messages related to the erroneous lines. This problem could be solved by forcing a broadcast of the line states at a certain time. Because the local CPU does not know which lines are in erroneous states, it has to broadcast the states of all lines. The checker then verifies all the lines. This operation introduces a network bandwidth overhead, and should be performed as rarely as possible. An error may propagate from a node to the system only when the node updates the shared memory, i.e. writes back. Thus, a possible solution is to broadcast the states of all the cache lines at every cache write back.

## 4.2 Experimental Evaluation

To evaluate the fault coverage of the proposed method experimentally and to validate its correctness, a cycle-accurate multiprocessor simulator based on the UNISIM environment [2, 1] has been used. The original simulator available in the UNISIM public repository has been extended to support the proposed technique and to inject faults

into the cache coherence states. Simulations have been conducted on a shared memory CMP with two PowerPC 405 32-bit RISC cores. Every CPU has a private data cache, connected to a bus. The cache contains 128 lines, each of the size 32 bytes. The cache is 2-way set-associative. A DRAM memory is also connected to the bus. The simulator implements the MESI protocol to maintain cache coherence.

The checkers implementing the logic described in Section 3.2 have been designed and integrated into the simulated multiprocessor as shown in Figure 1. A fault injector has been implemented which changes the current coherence state of a cache line to a different state, chosen randomly. This simulates both the faults of the cache controller logic and the faults in the storage elements holding the current state. The injection time and place is controlled by a random number generator. The injection frequency is controlled by a user-defined injection period. A synthetic benchmark has been used which lets different nodes perform read and write accesses to shared memory locations. This ensures that many coherence actions take place. The output produced in the presence of faults has been compared to the correct output to determine if the faults have affected the final result, which is of most importance for the end user. The fault injection frequency was varied from extremely high (roughly one fault per 100 clock cycles) to low (one fault per simulation).

14380 simulations with one or more injected faults have been performed. 6290 times the checker detected the error(s), 2466 times the simulator detected the error(s), and 5624 times the simulation finished with undetected faults, but the faults did not propagate to the output and did not corrupt it (these are escapes). We see the following possible causes of the escapes: the faults were injected into cache lines which were never referenced again in such a way that wrong data propagated in the system (see Section 4.1), or the faults were masked, i.e. the following fault(s) returned the correct state, or the state was changed by future coherence transactions. None of the simulations finished with undetected faults and corrupted output. The theoretically possible situations in which undetected faults violate the system data integrity and affect the output (see Section 4.1) never occurred in our simulations. This provides additional evidence that they are, indeed, very rare.

The storage overhead needed for the tags and coherence states of each checked cache line in the checker for the simulated cache configuration can be calculated as follows. The 32-bit address consists of a 21-bit tag, a 6-bit index, and a 5-bit block offset. Only the 21-bit tag and 2-bit MESI state need to be saved on the checker for each cache line. Since every line stores 32 bytes of data, the overhead is  $\frac{21+2}{21+2+32 \cdot 8} = 0.08$ , which is 8%.

## 5 Conclusions

This work has presented a technique to dynamically verify the cache coherence protocol operation on snooping-based multiprocessor systems. The widely used MESI coherence protocol has been analyzed in detail. A careful design and a limited speculation (such as accepting illegal  $E$  to  $M$  state transitions which are considered safe for system data integrity) allowed to completely avoid any performance overhead for the MESI protocol. This suggests that applying a similar approach to other protocols can also be done with limited or no performance overhead. The hardware cost is constant per processing node: for every cache, a fixed amount of redundant logic (the checker) is required. Thus, the technique is scalable. In addition, every checker contains copies of all the tags of all cache lines in the checked cache, which introduces approximately 8% overhead for the simulated architecture.

Only a few simple changes to an existing system are required to support the technique. The checkers can be easily integrated by connecting them to the communication network so that they can snoop all the messages. The message has to be extended with a few additional bits (3 in case of the MESI protocol and 2-way set-associative caches). The cache controllers need a simple modification to integrate the information about the way number and the current coherence state into every message they send.

The technique addresses faults in the logic controlling the coherence transactions, and in the storage elements holding cache line states. Almost all single-fault scenarios are covered, except for a few rare cases described in Section 4.1. Experimental results show that these cases are very rare. None of over 14 thousand simulations finished with undetected faults leading to corrupted output. Furthermore, the technique can be strengthened by using multiple checkers per cache if very high error rates are expected.

To provide a full system coverage, the proposed technique is assumed to be used in combination with other detection and/or correction techniques. These techniques would cover the faults within the nodes, and the network faults which are only partly covered by the proposed scheme. Note that while the logic (except the cache coherence controllers) and memory on all the nodes need extra protection, the coherence checkers proposed in this work do not necessarily need to be error-free. This is because the cache controllers are essentially duplicates of the checkers, thus they check the operation of each other.

The technique presented in this work aims at a fail-safe operation. To enable recovery, some modifications are required, because in the fail-safe mode error detection latency is unpredictable. For example, to implement checkpointing, the caches can submit the states of all the lines to the checkers for verification. If no errors are detected, a checkpoint

can be made. This, however, is likely to introduce performance overhead. Extensions of the proposed technique to support recovery as well as applications to other than MESI coherence protocols are planned as future work.

## References

- [1] UNISIM: UNIted SIMulation environment webpage. <http://unisim.org/>.
- [2] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE Computer Arch. Letters*, 6(2):45–48, 2007.
- [3] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. *Technology@Intel Magazine*, 2005.
- [4] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Dynamic Verification of Cache Coherence Protocols. In *Proc. ISCA Workshop on Memory Performance Issues*, 2001.
- [5] R. Fernandez-Pascual, J. Garcia, M. Acacio, and J. Duato. A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures. In *HPCA-2007*, pages 157–168, 2007.
- [6] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai. Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes. *VLSI Technology. Digest of Technical Papers*, pages 73–74, 2001.
- [7] J. Hennessy and D. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann, 3 edition, 2003.
- [8] A. Mahmood and E. McCluskey. Concurrent Error Detection Using Watchdog Processors—A Survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.
- [9] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *ISCA-03*, pages 182–193, 2003.
- [10] A. Meixner and D. Sorin. Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures. In *HPCA-2007*, pages 145–156, 2007.
- [11] M. Papamarcos and J. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA-84*, pages 348–354, 1984.
- [12] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *ISCA-02*, pages 111–122, 2002.
- [13] T. Rao and E. Fujiwara. *Error-Control Coding for Computer Systems*. Prentice-Hall, Inc., 1989.
- [14] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *DSN-02*, pages 389–398, 2002.
- [15] D. Sorin, M. Hill, and D. Wood. Dynamic Verification of End-to-End Multiprocessor Invariants. In *DSN-03*, pages 281–290, 2003.
- [16] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *ISCA-02*, pages 123–134, 2002.