Concurrent Verification Experience of Cache Protocol in Real Development of Large SMP Server Product by Using Model Checking

Toru Shonai
Hitachi, Ltd.
Kokubunji, Tokyo 185-8601
toru.shonai.hs@hitachi.com

Shoichi Hanaki
Okano Electric Co., Ltd
Higashi-kurume, Tokyo 203-0003
syoichi.hanaki@okano-denki.co.jp

Yoshiaki Kinoshita
Hitachi, Ltd.
Shinagawa, Tokyo 140-8572
Yoshiaki.kinoshita.zw@hitachi.com

Abstract – We have verified the cache protocol by using model checking in real development of the highly multiple-CPU server product. A formal verification engineer abstracted the models for model checking several times through the design process from the protocol specifications written in natural language by the architect team. We discovered actual nine complicated protocol bugs acknowledged by the architects in advance of logic simulation. Some bugs we found were too complicated to be replicated in logic simulation. This effort surely shortened the total design duration. We proved the effectiveness of formal verification of cache protocols in early design phase of real server product development.

I. Introduction

Actual design process of digital system products consist of micro-architecture design, logic design, and logic verification. Logic verification usually consists of logic simulation and laboratory test using real machines manufactured. There are two problems concerning to them.

The first problem is that micro-architecture specifications, which are written through the micro-architecture design by architects, are checked by no more than inspection and walkthrough. When bugs are passed over, they are probably not detected until logic verification phase. The more lately bugs are detected, the more man-months and cost to change the design are needed. In particular, bugs of micro-architecture specifications detected in the later phase of product development may invoke immense negative influence.

The second problem is that because it is impossible to verify the designed system with all the possible test cases in logic simulation, we cannot detect all the bugs. When bugs are discovered at the laboratory test after logic simulation, several weeks or a few months are necessary to change and reproduce the LSIs. Therefore it is important to discover bugs before the laboratory test in order to decrease LSI reproduction and the duration of product development. In addition, the verification by using laboratory test itself is not perfect and bugs may be found after product shipment.

The increased complexity of the present digital system makes more difficult to detect all the bugs. For example, the complicated bugs of high-end servers with multiple CPUs have the sequence of tens of particular operations before malfunctions occur. Because the combination of the input data increases by the exponent of the number of operations in the sequence, it is extremely difficult to discover these bugs.

Some of these bugs need so strict and so rare conditions to occur where many very rare events occur in the particular order and at the particular timing that it is difficult to replicate the bugs even if the conditions are well known.

To cope with these problems, model checking, a kind of formal verification, was proposed. Although model checking has a problem that the size of finite state machine applicable is small, by using abstraction technology to decrease its state size and by dividing the target to be verified, SGI[2], IBM[3] and Intel[4] have successfully applied the formal verification to products development. In [2], model checking was applied to actual server development of directory-based cache protocol and found numerous problems that would have been extremely difficult to find with conventional simulation techniques. Other activities are also reported. Cache control protocol in JUMP-1, research-level directory-based server machine, was formally verified with model checking after fabrication[5]. Experimentally designed SCI-cache protocol was formally verified[6]. KBUS specifications in 128way server product is verified inspection and formal verification, but no bugs were found by formal verification and no further details of formal verification was described[7]. Cache coherency protocol for speculative multithreading was experimentally designed and verified with model checking[8]. HAL S1 system directory-based cache coherence protocol was verified with model checking in real product development process, but no real bugs of the real design were found[9]. Early protocol design errors of directory-based cache coherent protocol of ASURA multiprocessors at Fujitsu Inc. were detected by statically checking protocol properties using SQL[10]. Intel's multicore cache coherent protocol LCP (Larrabee coherence protocol) was verified with parametric protocol verification using flows[11], and Intel's hierarchical cache protocol with off-chip protocol QPI and on-chip coherence protocol was tried to verify with refinement checking[12] , but both had no explanation of real protocol bug detection. There are very few model checking activities applied to real cache protocol bugs of real server products in real development process.

In this paper we describe the concurrent verification of cache protocol by using model checker VIS(Verification Interacting with Synthesis)[13,14] in real development of the large SMP server of global-snooping-based cache-coherent protocol rather than directory-based with at most 32 Intel Itanium processors in design. The aims of this research are to find bugs which are difficult to detect using conventional logic simulation, and to find bugs in the micro-architecture specifications as early as possible by performing

micro-architecture design and verification concurrently.

To these ends we assigned a formal verification engineer of micro-architecture other than the architects team of micro-architecture specifications. The architects wrote the specification by Japanese language, figures, tables and time-chart and so on. The formal verification engineer read the specifications carefully, made verification models and verified them using the model checker.

We designate the time when preliminary version of the specifications was issued as a first result of micro-architecture design by the architects—Dec. 1997—as the starting point. The 0.0 version was issued two months later, the 1.0 version four months later, the 2.0 version seven months later, the 3.0 version eight months later and the 3.1 version twelve months later from the starting point. Micro-architecture design completed then and logic design was started. Even if the revised version was not issued, the specifications were partially revised at any time when some error, unclearness or contradiction were discovered. Preparation for model checking was started four months later from the starting point. Techniques, tools, their estimation and understanding of micro-architecture specifications were completed until six months later from the start point. Model checking was started from the seventh month and continued to the twenty-first month with successive model updates when the specifications were changed or when the models are refined. Model checking continued from the latter phase of micro-architecture design to the end point of logic design, i.e., the starting point of logic simulation.

II. Micro-architecture for cache control

A. System Organization

Figure 1 sketches the configuration of the system. It consist of symmetric multiprocessing (SMP) nodes connected by proprietary interconnect based on Node Controller and Crossbar Switch chips. Each processor has L1 and L2 cache. Each node has four processors, L3 cache and distributed global memory. Based on this design, we manufactured and shipped the 8way server, Hitachi Advance server HA8000-ex/880 in 2001.

B. Cache Protocol

The system has to ensure that all processors access no old data. In order to achieve this, we choose global-snooping-based cache-coherent protocol, rather than directory-based in term of memory access latency performance. In the following we first explain basic behavior without L3 cache, then explain the effect of L3 cache and L2 cache tag memory for reducing snooping transactions.

1) Basic Behavior

In general, processors have L1 and L2 cache. But when considering cache coherency, we can view that processors have only L2 cache. L2 cache has four state: M(modified), E(exclusive), S(shared), and I(invalid) of MESI protocol[15]. Two types of memory request transactions exist: read and read-invalidate.

A processor which has issued a read request caches its memory line in the S state when a processor has issued a read request. In case of a read-invalidate request, it caches its line as E or M state. Table 1 shows the state changes of cache lines when the processor has been snooped.

Snooping has to be performed on the processor in other nodes, the node controller sends a read request to all nodes via the crossbar switch. Each node makes all four processors snooped with this request. When M line hits as the result of the request, the line is write-backed into memory and is set to the requesting processor. We call sending the line from the hit processor to the requesting processor as cache-to-cache (c2c) transfer.

2) Transaction reduction mechanism

Although cache consistency is maintained with the mechanism explained in the above, increased transactions by snooping all processor in all nodes every time makes system performance lower. To cope with this, we adopt L3 cache and L2 cache tag to the system. [16]

i) L3 Cache

Each node has L3 cache. When a processor in the node caches the line, it is also write into L3 cache. L3 cache has the same four states as L2 cache. When a processor request a read and L3 cache has its line, the line is read form the L3 cache. When a processor requested a read-invalidate and L3 cache has its line in the E or M state, the line is also read from the L3 cache. As a result, transactions sent to the crossbar switch are reduced.

L3 cache is also snooped by other nodes and the state of its lines change as L2 cache. When L3 cache is snooped and it has the latest M line, the line is write-backed from L3 cache.

ii) L2 Cache Tag

It's unnecessary to perform snoop requests from other nodes, if it is known that all processors in the node do not have the requested line in all cache. To achieve this, each node have L2 cache tag.

However, because L2 caches may change their states without notifying its change to the bus, all their state changes cannot be reflected in L2 cache tags. For this reason, L2 cache tags have three states: EM, S and I. I designates that all

Table 1 L2 state transition when processor snooped

transaction	L2 state before snoop	L2 state after snoop	response
read	I	I	OK
	S	S	HIT
	E	S	HIT
	M	I	HITM(writeback)
read-invalidate	I	I	OK
	S	I	OK
	E	I	OK
	M	I	HITM(writeback)

processors in the node have no cache line, S designates that some processors may have S-state lines, and EM designates that some processors may have E-state or M-state lines.

When L2 cache tag state is I, it is unnecessary for other nodes to snoop the corresponding bus. Also when L2 cache tag state is S for read request, snoop in unnecessary. Therefore, bus transactions are reduced.

C. Cache Line Replacement

When cache have no room for caching a new line, an old data line is get out from the cache to make room. This is called cache line replacement. S-state or E-state lines in L2 cache can be replaced without notifying it to the bus. When M-state lines are replaced, data lines are write-backed. Into where write-back is performed is L3 cache, write-back is performed into L3 cache and L3 cache becomes also M-state.

In case of replacement of L3 cache, L2 caches have to be invalidated, by snooping the processor bus. If M-state line of L2 cache hits at this time, write-back is done from L2 to the memory. Otherwise, write-back is done from L3 to the memory when L3 has the M-state.

III. Formal Verification Method

A. Model Checking System VIS

Although model checking is so powerful tool that can check all the states of the verification model, the size is limited and complicated specification cannot be verified directly. To reduce the size of states dramatically, the verification model must be made by simplifying and abstraction. The problem is that making the verification model which has the checkable size of state and which can detect subtle bugs.

Two model checking systems could be used at that time. SMV(Symbolic Model Verifier)[1] developed at CMU and VIS[13,14] at UCB, we chose VIS for its various features. In VIS, the verification models are described in extended Verilog and verification specifications are written in CTL.

Figure two show the formal verification flow of the micro-architecture specifications. Firstly, the verification models are written by modeling the specifications. Furthermore, the verification specifications are written in CTL. These two are input to VIS. After VIS runs, OK or NG of the verification specifications are output, and in case of NG, sequences from initial states to illegal states are listed as counterexample.

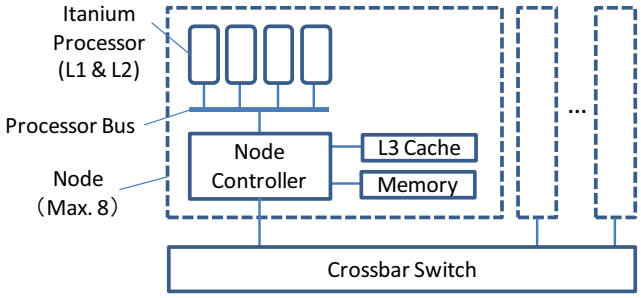

Figure 1 System configuration

B. Verification Models

The verification models consist of one to three nodes and one to four processors each node.

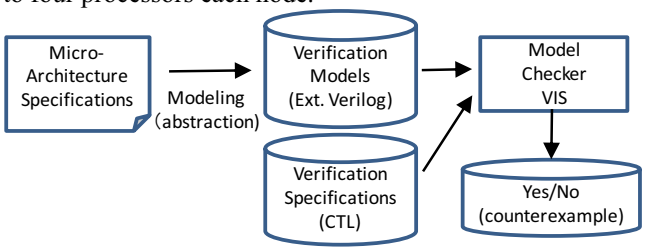

Figure 2 The Formal Verification Flow

A data is only one bit, and cacheable and writable. Every cache can hold only one line with one bit. Because data is one bit, only one node has a memory, and the others have no memory. The initial value of data is 0 and the value can be rewritten from 0 to 1 only. Therefore, consistency checks can check the data with the value 1 is always newer than one with the value 0.

VIS permits non-deterministic descriptions which can express plural behaviors verification with one description. Non-determinism is used for several purposes. In general, type of the transaction which a processor issues depends on the memory access which a processor instruction issues and the state of the processor's cache. However, so as to verification of bus protocols, detailed processor's behavior make no sense. In such a situation, the type of transaction the processor issues should be described as non-deterministically determined.

Furthermore, by using abstraction with non-deterministic description, the verification under the more general conditions can be performed. For example, when an arbitration logic is modeled in detail, bugs which don't occur under this arbitration never be verified. By abstracting away the detailed arbitration logic and describing that any resource can have priority non-deterministically, the verification independent from the arbitration logic can be performed.

In addition, by using non-deterministic description, the events made unhappened by abstraction can be happened For example, although cache line never be replaced by other line in the model with one cache line only, replacement occurring at the arbitrary timing can be written to verify the cases relating replacement behaviors. Delay or wait of the arbitrary time interval are also described non-deterministically. VIS system verifies all combination of non-deterministic behaviors.

In short, it is important to describe both to reduce the number of the state by abstract and to simulate the behaviors similar to the actual behavior.

C. Verification Specifications

1) Cache Consistency Check

It is verified that processors never read old data. The verification specifications are that a processor never read data 0 after it reads data 1.

2) Deadlock Check

It is verified that a transaction which a processor issues will be certainly complete. The verification specifications are that a processor can transit from arbitrary any state to the state where a read or a write can be issued.

3) Invariant Check

Invariant is the condition that satisfied at all states. Because invariant check is faster than other CTL checks as (1) or (2), we verified a lot of invariant check. For example, we verified that when a processor has E/M line, other processors have no line, that the data of L2 cache in S/E state is equal to the data of L3 cache and that the data of L3 cache in S/E state is equal to the data in memory.

IV. Verification Results and Considerations

A. Verification Results

Table 2 shows model checking execution results of three verification models with different functions and abstraction levels. Each model has two nodes with one to three processors. The numbers of Verilog source code lines and latches represent the approximate size of models. CPU times and memory usage of three checks are presented in the table. The machine we used had 400MHz Xeon and 1G byte main memory. We quit the execution at the limit of CPU time 10,000sec. Deadlock and consistency checks consume so much time that they could not complete the run. Invariant checks are so faster that they could complete when deadlock and consistency checks could not complete.

Table 3 shows the protocol bugs variation found by the formal verification. Nine protocol bugs have been found in the verification period. Some have been found in inspection relating formal verification. Architects have acknowledged that all bugs were protocol bugs and the specifications documents were correctly revised by them. Eight bugs were found by consistency check and no bugs were found by the deadlock check. As to finding situation, two bugs were found only by model checking, four were by model checking and relating inspection and three were by inspections. Model checking and relating inspection means that counterexamples VIS showed were not bugs but genuine bugs were found through examination of counterexamples. They were because the models had some faults. If the model would be corrected, these bugs could be found by the model checking.

All consistency errors could be found by some invariant check. Therefore they could be verified. On the contrary, deadlock check errors could not be found by any invariant check. The following is a complicated protocol bug example found.

Table 2 Execution Results

Models	The number of Verilog codes (The number of latches)	The Number of Proc-essors	Reachable state		Invariant Check		Deadlock Check		Consistency Check	
			depth	The Number Of State	CPU (sec)	Mem. (MB)	CPU (sec)	Mem. (MB)	CPU (sec)	Mem. (MB)
Model 1	994 (120)	1	44	8.14E04	88.6	12.5	147.6	14.1	175.2	17.1
		2	53	4.39E06	342.9	19.6	1045.6	32.4	1421.3	41.9
		3	64	4.60E07	592.1	26.4	>10000	—	8850.6	155.1
Model 2	1133 (126)	1	42	1.48E05	28.1	13.2	72.4	14.4	185.4	17.2
		2	40	4.48E06	45.3	14.1	229.8	23.7	710.1	38.2
		3	40	4.50E07	64.5	15.0	836.6	48.8	>10000	—
Model 3	2130 (383)	1	91	5.68E06	1170.5	32.8	>10000	—	>10000	—
		2	93	2.52E08	1704.1	40.3	>10000	—	>10000	—
		3	93	2.25E09	1997.3	44.4	>10000	—	>10000	—

Table 3 Bugs Found

Type	Situation when bugs are found	The number of
Consistency Error	Model check	2
	Inspection after model check	4
	Inspection	2
Deadlock	-	0
Others	Inspection	1
The Total	-	9

B. Bug Example

Figure 4 shows the cache consistency error example. It shows the behavior of a node. L3Q is the queue to hold transactions relating L3 cache, NPQ is the queue to hold snoop requests from other nodes and snoop requests relating L3 replacements until both request are issued to the bus, and OOQ in the figure is the queue to hold read transactions to other nodes until their responses arrive. There exists two processors P0 and P1

(0) Initially P0 has M line with data=0, L2 tag is EM state and L3 is E state. (1) P1 issues read-invalidate request and P0 is snooped. (2) Because P0 has M line as a result of snooping, P0 performs a write-back. P0 gets a line as M state. (3) As a result of a write-back by P0, a write transaction to L3 enters into L3Q. The L3 state of the node becomes M. (4) P1 replaces data 0 with 1, and the state of its cache becomes M. (5) A read transaction arrived from another node. (6) The processor bus is snooped because L3 tag is EM. (7) Because P1 holds M line, P1 perform write-back into memory and c2c transfer. The states of L2, L2 tag and L3 all become I. (8) P0 issues read and P1 is snooped. (9) Because the node has no line,

a read transaction enters into the OOQ. (10) A read transaction is transferred via the crossbar switch. (11) After snooping is done in another node, data arrived from the node. Data=1 because the data is replaced at (4).

(12) Data is transfered to P0. At the same time, this data line is registered into the L3 cache because the state of L3 is I. Data=1. (13) Data 0 is written into the L3 cache from the L3Q.

At this time, an error occurs because the old data is written into L3 cache. The root cause of the bug is because L3 write at (3) and L3 write at (12) are reversed. This reverse does not happen in general, but it happens when the snoop from the other node is done at (5) with L3 write in the L3Q and when the states of L2 and L3 become both I. The architect corrected the bug by denying to receive the snoop from the other node at (5) and by retrying the snoop.

C. Considerations

It is really impossible to find the previous bug with logic simulation because the frequency of the bug is extremely few and because the sequence of operations until the erroneous situation is extremely long. The bug is an extremely rare case because the particular events sequence on the same line has to happen exactly and at the particular timing sequence. Furthermore, the operations including those in the other nodes until the bug appears are 20 to 30 steps, and the input data to invoke the bug is extremely large.

Another reason why it is really impossible to find these bugs with logic simulation is because the operation sequence have to happen at the strict timing conditions while it happens at any arbitrary timing conditions by non-determinism in the model checking. For example, the replacement of L3 cache happens only when a write happens to L3 cache, when no room in L3 cache invokes replacement and when the line under the attention of the model checking is selected to be replace. The bug example happens only when write transaction to L3 cache is stalled in the L3Q from (3) to (12). This situation can be happened by logic simulation only when particular conditions all have hold. Therefore, it is actually impossible for these particular conditions to happen in the particular timings in the logic simulation.

As a result, if the bug described above would not been found in the model checking, it is probable that the bug could not be found until the latter part of the logic simulation period, or it is possible that they could not found through all the logic simulation period.

By doing model checking of the micro-architecture of the cache control concurrently with the design of the micro-architecture design, the complicated micro-architecture bugs can be found before the logic simulation period and we can achieve the initial aims.

Furthermore, by analyzing the characteristics of the bugs we found by model checking, we suggested the architect team the transaction patterns they tended to make bugs, and this information is also given to the test program developing team to make many test cases for these patterns.

However, the modeling processes depend on the human expertise, so that the completeness of the test coverage is unknown. The micro-architecture specifications are more than 100 pages with figures, tables, time-charts and Japanese descriptions. It is the formal verification engineer's expertise to which parts are to be modelled with simplification and abstraction and what style in order to run the models in allowable times. Completeness is the future problem. However, we insist that model checking is very valuable method compensating logic simulation in real develop processes of real server products those days and now.

V. Conclusions

We have verified the cache protocol in real development of the highly multiple-CPU server product. We have verified formally the design model that we abstracted from the protocol specifications concurrently with their design in time. We abstract the verification models with one bit and one cache line from the micro-architecture specifications, and verified cache consistency checks, deadlock checks and invariant checks. We discovered actual nine complicated protocol bugs in advance of logic simulation. The bugs we found have too complicated conditions to be occurred in logic simulation. This effort surely shortened the total development duration. We proved the effectiveness of formal verification of cache protocols in early design phase of real server product development.

Acknowledgements

We thank to H. Akashi, Y. Tsushima, K. Uehara and other architects. We also thank to Prof. Fujita, Prof. and Prof. Sakai for valuable advice on the paper. Thanks to the anonymous reviewers for their comments for improving the quality of the paper.

References

[1] K. L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[2] Asgeir Th. Eiriksson, "Integrating Formal Verification Methods with A Conventional Project Design Flow", DAC95, 1995.

[3] Ilan Beer, et al., "RuleBase: an Industry- Oriented Formal Verification Tool", DAC96, 1996.

[4] M. Agaard, et al., "Combining Theorem Proving and Trajectory Evaluation in an Industrial Environment", DAC98, 1998.

[5] Fukushima , Hamagkuchi, Tomata, Yajima , Formal Verification of a Cache Protocol Design : A Case Study in the Massively Parallel Computer JUMP-1, IPSJ SIG Technical Report, ARC-117-1, 1996.

[6] Morihiro, Yoneda, Formal Verification System Based on Simulation, IEICE Transaction on Information and System D-I, J84-D-1, No. 4, pp.367-377, 2001.

[7] Shimizu, Watabe, Kobayashi, Ishihata , Kaiser: 128-CPU SMP Server Design and Evaluation, IPSJ Journal, Vol. 24, No. 4, 2001.

[8] Monma, Hun, Tashiro, Sakai, Verification of Cache Coherency Protocol for Speculative Multithreading, IPSJ SIG Technical Report, 2005-ARC-164, pp. 103-108, 205.

[9] A. Hu, M. Fujita and C. Wilson, Formal Verification of the HAL S1 System Cache Coherence Protocol, ICCD '97, pp.438-444, 1997.

[10]R. Brayton, et. al., "VIS: A System for Verification and Synthesis", in Proc. Conference on Computer Aided Verification, 1996.

[11]M. Subramanian, "Early Error Detection in Industrial Strength Cache Coherence Protocols Using SQL", in IPDPS 2003.

[12]J. O'Leary, M. Talupur, and M. Tuttle, "Protocol verification using flows: An industrial experience", in FMCAD 2009.

[13]J. Bingham, J. Erickson, G. Singh, and F. Andersen, "Industrial Strength Refinement Checking," in FMCAD 2009.

[14]The VIS Group, "VIS User's Manual", http://embedded.eecs.berkeley.edu/research/vis

[15]J. Archibald and Jean-Loup Baer, Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, ACM Trans. On Computer Systems, Vol. 4, No.4 Nov. 1986, pp.273-298.

[16]USP-6438653, H. Akashi, Okochi, Shonai, Kashiyama 、 Cache memory control circuit, 2002.8.20.

[17]D. E. Culler, J. P. Singh and A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufman Publishers, Inc., 1999.

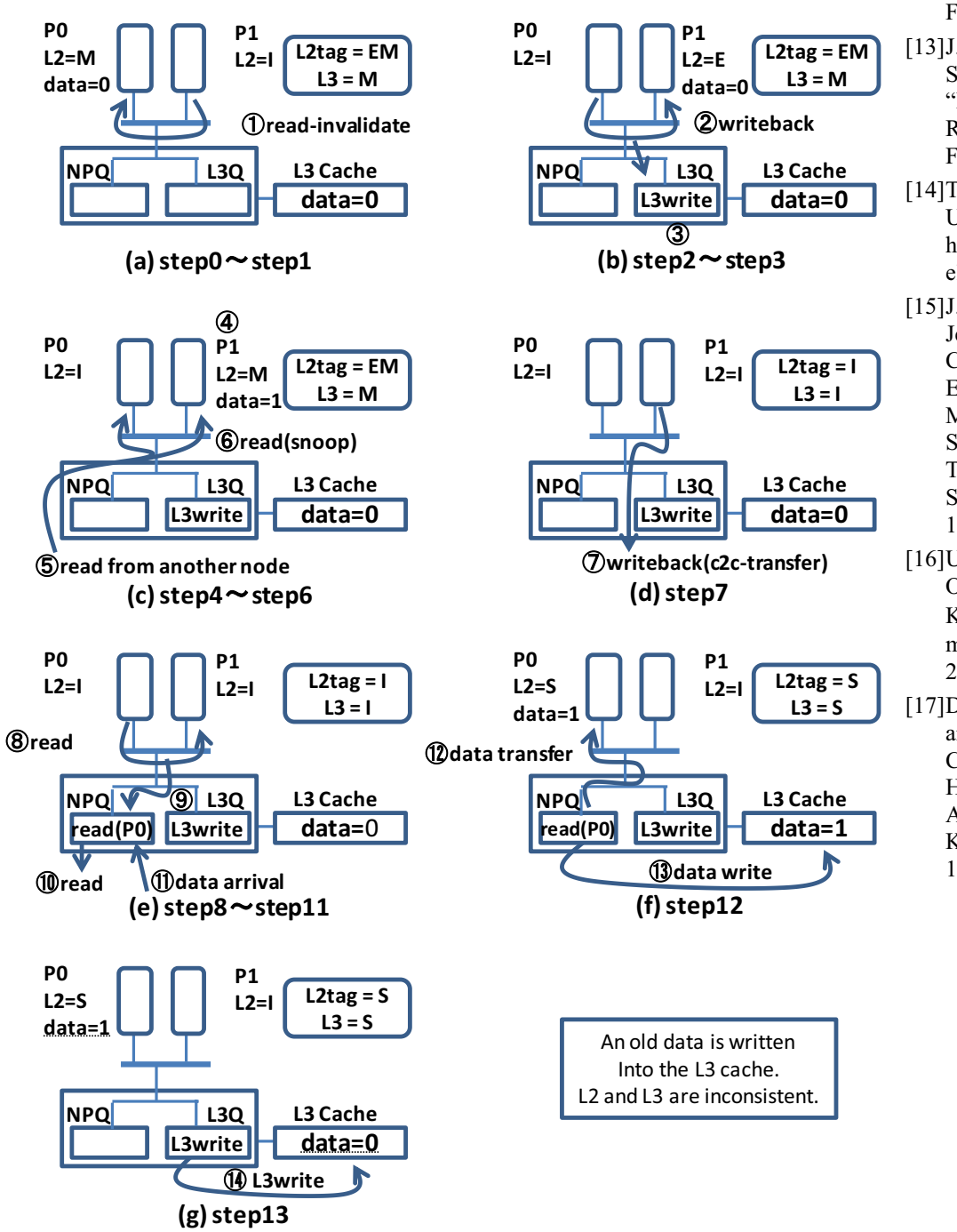

Figure 3 Consistency Bug Example