Formally Verifying the Distributed Shared Memory Weak Consistency Models

Venkateswarlu Chennareddy [1], Jatindra Kumar Deka [2]
[1] *D. E. Shaw India Software Private Limited,*
Begumpet, Hyderabad - 500016, INDIA, venkateswarlu.Chennareddy@deshaw.com
[2] *CSE Department, Indian Institute of Technology Guwahati*
North Guwahati, Guwahat - 781039, INDIA, jatin@iitg.ernet.in

Abstract

A specification and verification methodology for Distributed Shared Memory (DSM) consistency models specifically weak consistency model is proposed. For this, we designed and implemented abstract DSM System. In DSM system, sequential consistency unnecessarily reduces the performance of the system because it does not allow to reorder or pipeline the memory operations. Relaxed memory consistency allows reordering of memory events and buffering or pipelining of memory accesses. So that relaxed consistency improves the performance of the DSM system. For any critical system, it is very important to develop methods that increase our confidence in the correctness of such systems. One of such methods for checking the correctness of critical system is formal verification. For verification of weak consistency models we specify the weak consistency properties and are verified on Abstract DSM System using CADP Tool box.

1. INTRODUCTION

Today, hardware and software systems are widely used in applications where failure is unacceptable. We frequently read of incidents where some failure occurred due to error in a hardware or software system. For reliable systems, it is very important to develop methods for correctness of such systems. The principal validation methods for complex systems are simulation, testing, deductive verification, and model checking [1]. Simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. In both cases, we will give certain inputs and observe corresponding outputs. Deductive verification consists of axioms and proof rules to prove the correctness of systems. The importance of deductive verification is widely recognized by computer scientists. Deductive verification is a time consuming process that can be performed only by experts who are educated in logical reasoning and have considerable experience. Consequently, use of deductive verification is rare. An advantage of deductive verification is that it can be used for reasoning about infinite state systems. Model checking is a technique for verifying finite state concurrent systems. One benefit of this restriction is that verification can be performed automatically. The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not. The procedure will always terminate with yes/no answer. Model checking consists of modeling, specification and verification steps. An exciting new research direction [2] attempts to integrate deductive verification and model checking, so that the infinite states of complex systems can be verified automatically.

As the need for more computing power demanded by new applications constantly increases, systems with multiple processors are becoming a necessity. The gap between processor and memory speed is apparently widening, and that is why the memory system organization becomes one of the most critical design decisions to be made by computer architects. According to the memory system organization, systems with multiple processors can be classified into two large groups: *shared memory systems* and *distributed memory systems*. In a shared memory system (SMS) [3] (often called a tightly-coupled multiprocessor), a single global physical memory is equally accessible to all processors. The advantage of SMS is very simple and easy to program. However, they typically suffer from increased contention in accessing the shared memory, especially in single bus topology, which limits their scalability. In addition to that, the design of the memory system tends to be more complex. A distributed memory system (often called a multicomputer) consists of a collection of autonomous processing nodes, having an independent flow of control and local memory modules. Communication between processes residing on different nodes is achieved through a message passing model, via a general interconnection network. Such a programming model imposes significant burden on the programmer, and induces considerable software overhead. On the other hand, these systems are claimed to have better scalability and cost-effectiveness.

A distributed shared memory (DSM) [4] tries to combine the best of these two approaches. A DSM system logically implements shared memory model on a physically distributed memory system. This approach hides the mechanism of communication between remote sites from the application writer, so the ease of programming and the portability typical for shared memory systems, as well as the scalability and cost-effectiveness of distributed memory systems, can be achieved with less engineering effort.

In this work, we formally verified some of the weak consistency properties of distributed shared memory model. We modeled an abstract distributed shared memory system

1-4244-0716-8/06/$20.00 ©2006 IEEE.

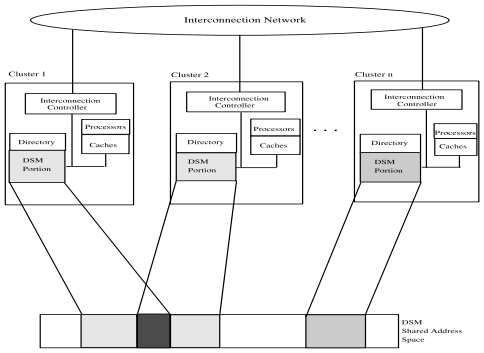

Fig. 1: Structure and Organization of a DSM system

which captures the behavior of a distributed shared memory. We specified the abstract DSM in LOTOS and verified some of the properties related to weak consistency model of DSM with the help of CADP tool box. During our experiment it is observed that CADP tool box can handle a reasonably big system for specification and verification.

The rest of the paper is organized as follows: We present some work related to distributed shared memory, its consistency models and verification of consistency model of distributed shared memory in Section 2. Section 3 presents a brief overview of CADP tool box. In Section 4, we present the design and implementation issues of abstract DSM system. The issues related to specification and verification of weak consistency properties with the help of CADP tool box is presented in Section 5. Finally Section 6 briefly explains the conclusion of our work.

2. Related Work

A DSM system generally involves a set of nodes or clusters, connected by an interconnection network is shown in Figure 1. A cluster itself can be uniprocessor or a multiprocessor system, usually organized around a shared bus [5]. Private caches attached to processors are virtually inevitable for reducing memory latency. Each system cluster contains a physical local memory module, which maps partially or entirely to the DSM globally address space. Regardless of topology - bus, ring, mesh or local area network - a specific interconnection controller in each cluster connect it into the system. In order to reduce average access time to the shared data, we replicate some data in multiple copies that reside in different memory locations. When multiple copies of same data exist, modification of one copy makes other copies stale. So we have to invalidate or update the other copies. In order to maintain consistent data we have to follow consistency model and choosing consistency model is one of the key issue in DSM systems.

The memory consistency model [6] defines the legal ordering of memory references issued by a processor, as observed by other processor. The memory consistency models basically divided into two types, i.e., *Strong Consistency* and *Relaxed Consistency*. Different types of parallel applications inherently require various consistency models. The model's restrictiveness largely influences the system performance in executing these applications. Stronger forms of the consistency model typically increase memory access latency and bandwidth requirements, but it simplifies programming. Looser constraints in more relaxed models, which allow memory reordering, pipelining, and overlapping, consequently improve performance, at the expense of higher programmer involvement in synchronizing shared data accesses. For optimal behavior, systems with multiple consistency models adaptively applied to appropriate data types have recently emerged. Stronger memory consistency models that treat synchronization accesses as ordinary read and write operations are *sequential* and *processor consistency*. More relaxed models that distinguish between ordinary and synchronization accesses are *weak, release, lazy release*, and *entry consistency*. The *Weak Consistency* (WC) model was proposed by Dubois et al. [7]. In WC model, memory accesses are divided into ordinary shared accesses and synchronization accesses. The performance of WC models heavily depends on synchronization rate in the user code [8]. If the synchronization rate is less in user code then the performance of weak consistency is equivalent to release consistency. The disadvantage in WC Model is all synchronization accesses must be identified by the programmer or the compiler.

We have mentioned closely related work, pertaining to finite-state verification of protocols with respect to consistency. Graf [9] introduced a verification approach for sequential consistency. They gave a set of properties expressible as temporal logic formulas such that any system satisfying them is a sequential consistent memory. Then, they verified these properties on a distributed cache memory by means of verification method. Our approach is similar to Graf's approach. Rob Gerth [10] proposed a very similar approach to ours, using a lazy caching algorithm and sequential consistency.

Henzinger et al. [11] proposed an approach for verifying sequential consistency on shared-memory multiprocessor systems. They verified sequential consistency of memory systems with an arbitrary number of processors, locations and data values using a model checker. They have considered two specific memory protocols, namely the lazy caching protocol and a snoopy cache coherence protocol. Shaz Qadeer [12] proposed an approach for verifying sequential consistency on shared memory multiprocessor systems by model checking. They presented a model checking algorithm to verify sequential consistency on systems for a finite number of processors, memory locations and an arbitrary number of data values. Condon et al. [13] proposed a verification approach based on logical clocks for verification of sequential consistency.

Recently, P. Chatterjee et al. [14] proposed an approach for specification and verification framework for developing weak shared memory consistency protocols. They applied the proposed method to four snoopy-bus protocols for implementing aspects of the Alpha and Itanium memory models. Ghughal et

al. [15] proposed an approach for verification of weak shared memory consistency models. They constructed an architectural testing programs similar to those constructed by Collier suited for weaker memory models. Their work was mainly focused on architectural tests for weaker memory models and the new abstraction methods to construct test automata for weaker memory models. P. Chatterjee et al. [16] proposed a formal approach to verify protocol implementation models against weak shared memory models through refinement checking supported by a model checker. They verified four different alpha and Itanium memory model implementation against their respective specifications. They used it to check for the existence of a refinement mapping between an implementation model and an abstract model.

3. CADP Tools Overview

CADP [17] (Construction and Analysis of Distributed Processes) is a popular toolbox for the design of communication protocols and distributed systems. CADP is developed by the VASY team at INRIA, France. LOTOS (Language Of Temporal Order Specification) is a specification language that has been specifically developed for the formal description of the OSI (Open Systems Interconnection) architecture, although it is applicable to distributed, concurrent systems in general. We will write high level protocol description in LOTOS. The CADP tool box contains various closely interconnected tools: *CAESAR* is a compiler that translates the behavioral part of a LOTOS specification into Labelled Transition System (LTS). *CAESAR.ADT* compiler translates the data part of LOTOS specifications into libraries of C types and functions. *ALDEBARAN* is a tool to convert LOTOS program to LTS in *aut format*. *XTL* (eXecutable Temporal Language), a Functional-like programming language designed to allow an easy, compact implementation of various temporal logic operators. *EUCALYPTUS* is a graphical user interface written in Tcl/Tk that integrates CADP. *SVL* (Script Verification Language), is a scripting language that targets at simplifying and automating the verification of LOTOS programs. *EVALUATOR* is an on-the-fly model checker for regular alternation-free mu-calculus formulas on Labeled Transition Systems.

4. Design and Implementation of Abstract DSM System

The architecture of Abstract DSM System M_{dsm} is depicted in Figure 2. M_{dsm} consists of a DSM address space and n processors, each processor associated with local DSM portion. Each local Memory M_i contains a part of DSM memory and has two queues associated with it: *out-queue* Out_i in which P_i's write requests are buffered and *in-queue* In_i in which pending local DSM updates are stored. The arrows indicate the information flow from out-queue to in-queue and DSM.

The data structures include DSM Address Space and n pairs of unbounded FIFO queues, In_i and Out_i. The entries in these queues are either (data, address) or (data, address, $*$), where $*$ stands for either 0 or 1. Here 0 indicates that the entry is written by some other processor and the updation done by the processor itself is denoted by 1. We define the following operations to be performed on these queues:

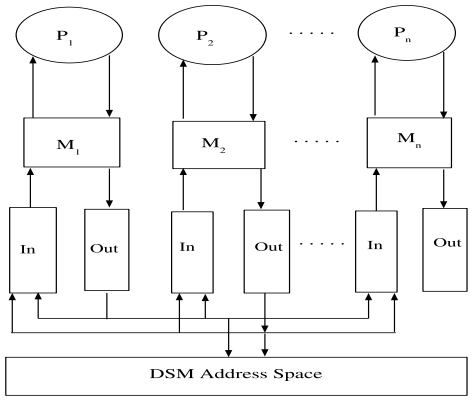

Fig. 2: Architecture of Abstract DSM System M_{dsm}

- *append(queue, item)* adds item as the last entry in queue.
- *First(queue)* returns the first entry in queue.
- *tail(queue)* returns the result of removing First(queue) from queue.
- { } denotes the empty queue.
- *queue[i]* denotes the i^{th} element of queue where queue[0]=First(queue).

The initial state of M_{dsm} are those states in which all queues are empty.

In our program formalism, the abstract DSM system can be described as a set of processes of the form $P_1 \ ||| \ P_2 \ ||| \ P_3 \ |||\cdots\cdots||| \ P_n$ where each process P_i is defined as follows.

Process Name: P_i
Variables:

Input:	a: address, d: datum
Local:	M_i: memory of address $\times$ datum (local memory)
Out_i:	buffer of address $\times$ $datum$, i: index
Shared:	In_i: buffer of address $\times$ datum $\times$ boolean, i: index
DSM:	memory of address $\times$ datum (Distributed Shared Memory)

Transitions:

$init$:	$\forall a \in address \wedge empty(Out_i) \wedge empty(In_i)$ $\wedge holds(M_i, (a, null)) \wedge holds(DSM, (a, null))$
$write_i$(a, d):	append(Out_i, (a, d), $Out_i^{'}$)
$read_i$(a, d):	empty(Out_i) $\wedge$ notBoolOne(In_i) $\wedge$ holds(M_i, (a, d))
mw_i(a, d):	First(Out_i,(a, d)) $\wedge$ tail(Out_i ,(a, d), $Out_i^{'}$) $\wedge$ update(DSM, (a, d), $DSM^{'}$) $\wedge$ $\forall$k$\in$index $\cdot$ append(In_i, ((a, d), i=k), $In_i^{'}$)
mr_i(a, d):	holds(M_i, (a, null)) $\wedge$ holds(DSM, (a , d)) $\wedge$ $\neg$isin(In_i, (a, d)) $\wedge$ append(In_i,((a,d),0), $In_i^{'}$)
du_i(a, d):	First(In_i,(a,d)) $\wedge$ tail(In_i,(a ,d),$In_i^{'}$) $\wedge$

	update(M_i,(a,d),$M_i^{'}$)
dl_i(a):	clear(M_i,a,$M_i^{'}$)
sync:	empty(Out_i) $\wedge$ empty(In_i)

Explanation for the functions used above is given here:

- *update(M_i,(a,d),$M_i^{'}$)*: updates at address location a in local memory M_i with datum d. We indicate the local memory after update as $M_i^{'}$.
- *notBoolOne(In_i)*: if In_i queue contains an entry of the form ($*$, $*$, 1) then it returns $false$, otherwise it returns $true$.
- *holds(M_i,(a,d))*: determines at address location a contains datum d in local memory M_i, if an entry available then it returns $true$, otherwise it returns $false$.
- *isin(In_i,(a,d))*: returns $true$ value if an entry (a,d) is available somewhere in queue In_i.
- *clear(M_i,a,$M_i^{'}$)*: sets datum to $null$ at address a in local memory M_i.

Process P_i wants to perform a write operation then add an entry (a, d) to Out_i queue. Process P_i wants to perform a read operation then Out_i queue must be empty and In_i queue doesn't have an entry of the form ($*$, $*$, 1) and local memory M_i contains an entry (a, d), and then read operation proceeds further. For memory write operation of process P_i, removes an entry from Out_i queue and updates its value in DSM global address space and then add an entry in all remaining processes In queues with an entry (a, d, 0) and add an entry (a, d, 1) to In queue of process P_i. Process P_i wants to perform a memory read operation, DSM global memory has an entry (a, d) and local memory M_i doesn't have value at address location a and process P_i In queue doesn't have an entry (a, d) then add an entry (a, d, 0) to In queue of process P_i. To perform process P_i's local memory update operation, removes an entry from In queue of Process P_i and update its local memory M_i. For local memory invalidate operation, we just clear the entry (a, d) in local memory M_i. For synchronization operation we have to complete all operations that are there in both In_i and Out_i queues. If we want to perform $sync$ operation In_i and Out_i queues must be empty. LOTOS also provides built in synchronization operation to perform synchronization between processes. Just, we have to mention where we want synchronization between processes. For example, we want synchronization between Process P_i and process P_j at $write$ operation, then we mention this as P_i |[write]| P_j.

5. Specification and Verification of Weak Consistency Properties

To verify weak consistency properties of DSM system, we need to specify weak consistency properties in temporal logic. This is one way of verification of properties. Another way of verification of properties is to identify the states involved in the properties that we want to verify, and then hide all the states in abstract DSM LTS except those states required in that property. After that, apply strong reduction on abstract DSM system. Then, we describe that property in states and transitions. Compare *Observational Equivalence* of abstract DSM system and property written in *aldebaran* format. SVL provides these features, i.e., comparison of observational equivalence and strong reduction. If these two systems are observationally equivalent then it terminates with $TRUE$, otherwise property is not satisfied some where in the system and terminates with $FALSE$. Third way of verifying the properties is to write property in temporal logic either in XTL or mu-calculus form. Then, SVL provides facility to verify that property written in temporal logic directly in LOTOS program. We verified weak consistency properties in several ways.

For weak consistency model of distributed shared memory, we have specified the following properties and verified for the abstract model of our distributed shared memory model:

Property P1: Whenever process P_i writes some value then process P_j wants to read the same value then Process P_j has to get the latest value written by process P_i. We will say this as in every process $write_i(a,\ d)$ has occurred, then $read_j(a,\ d)$ has to wait until (a, d) available, where index i indicate the process P_i performing the event, where a is address of the memory element and d is data element. Formal specification of this property is:

(P1) $\forall$(a,d) $\in$ address $\times$ data, $\forall$i $\in$ index
init$\Rightarrow$**AG**[*after(write(a,d))* $\Rightarrow$ (¬*enable(read(a,d))* **U** *avail(a,d)*)]

Property P2: Whenever process P_i has been written some value then local memory has to update it. We will say this property as whenever process P_i performed $write_i(a,\ d)$ operation then local memory updates $du_i(a,\ d)$ has to occur in the future states. Formal specification of this property is:

(P2) $\forall$(a,d) $\in$ address $\times$ data, $\forall$i $\in$ index
init$\Rightarrow$**AG**[*after(write(a,d))* $\Rightarrow$ **AF**(*du(a,d)*)]

Property P3: Third property of weak consistency is before an ordinary $READ$ or $WRITE$ access is allowed to perform with respect to any other processor; all previous synchronization accesses must be performed. Whenever we want to access ordinary $READ$ or $WRITE$ access all previous synchronization accesses must be completed. Synchronization accesses must be identified by the programmer or compiler in weak consistency. We need to ensure that data must be consistent at those synchronization accesses. Formal specification of this property is:

(P3) $\forall$(a,d) $\in$ address $\times$ data, $\forall$i $\in$ index
init$\Rightarrow$**AG**[*before(read(a,d)* $\vee$ *write(a,d))* $\Rightarrow$ **A**(*avail(prev(sync))*)]

Property P4: Before synchronization access is allowed to perform with respect to any other processor; all previous ordinary $READ$ and $WRITE$ accesses must be performed. Formal specification of this property is:

(P4) $\forall$(a,d) $\in$ address $\times$ data, $\forall$i $\in$ index
init$\Rightarrow$**AG**[*before(sync)* $\Rightarrow$
avail(prev(read(a,d) $\wedge$ *write(a,d)))*]

We verified these properties in three ways. In first method, Properties are written in *mu-calculus* form. *Caesar* compiler convert LOTOS program of abstract DSM system into LTS System. *Evaluator* is an on-the-fly model checker for regular alternation-free *mu-calculas* formulas on LTS. With the help of *Evaluator* model checker, we verified the properties in LTS. In second method, *Caesar* compiler translates the abstract DSM system described in LOTOS to LTS in BCG format. Then, hide all the states except those states which involve the operations required for that property in the LTS of the DSM. After that, we applied strong reduction on LTS System. We describe the properties in *Aldebaran* format *(.aut)*. *Aldebaran* form involves states and transitions, and then we compare the observational equivalence of these two systems. In third method, properties are specified in mu-calculus and verified on DSM in LOTOS format. *SVL* provides the facility to verify property written in mu-calculus directly on LOTOS program. We wrote script file for entire process of verification for weak consistency properties of abstract DSM system in *.svl* form.

We have modeled the abstract DSM system in LOTOS and used the CADP tool set to verify some of the properties on the abstract model of DSM. During our experiment we have modeled the DSM with different number of processors involved in the system. We look for the handling capabilities of the CADP tool set. We have gone up to 40 number of processors in the DSM system, which is a reasonably good number in distributed environment. The outcome of the experiment is tabulated in the Table 1. The first column of the table shows the number of processors in the distributed system. The second and third columns indicate the number of states involve in the system. We have performed the experiment in two different ways. First one is related to the number of states in the system which is given in column two and the other one is performed with strong reduction on the system. the number of states in the system after application of strong reduction is presented in column three. There is a considerable reduction of states after application of strong reduction. Similarly columns four and five show the number of transitions in the system before and after application of strong reduction respectively. Column six and seven indicate the memory requirement to store the abstract model of DSM before and after application of strong reduction respectively. It is observed that CADP tool set can handle DSM system with reasonably good number of processors involved in the distributed system.

TABLE 1: RESULTS WITH INCREASED NO. OF PROCESSORS

No. of Processors	States		Transitions		Size(in KB)	
	Before	After	Before	After	Before	After
3	25	12	110	24	3.1	2.5
4	41	22	212	53	4.0	2.7
5	76	44	403	228	4.8	3.1
10	236	159	1366	828	11.2	8.4
20	662	493	4281	3314	21.3	14.8
30	1142	865	7549	6094	31.4	22.1
40	2086	1704	13867	11981	40.8	29.7

6. CONCLUSION

We have designed and implemented abstract Distributed Shared Memory (DSM) system. In DSM system, data consistency is one of the key issues. We have to maintain consistent data when multiple processors are accessing the shared memory. Consistency models ensure that data are consistent and for correctness of consistency models, verification is important. We have modeled an abstract DSM system in CADP tool set and verified some of the properties related to weak consistency model of distributed share memory. In our experiment we have observed that CADP tool set is a powerful modeling environment for specification and verification of distributed and concurrent system. It can handle reasonably large system. Our model can be extended to verify the release consistency model of distributed shared memory. Programmer has to identify the all synchronization operations in weak consistency. These synchronization operations further divided to $ACQUIRE$ and $RELEASE$ operations in release consistency. In release consistency, it gives some more relaxation of memory reordering and pipelining such that it will perform much better than weak consistency.

REFERENCES

[1] E.M.Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, The MIT Press, 2nd Edition, 2000.

[2] S.Rajan, N.Shankar, and M.K.Srivas, An Integration of model checking with automated proof checking, *Proc. of the 7th International Conference on Comp uter Aided Verification*, LNCS, Vol. 939, pp. 84-97, 1995.

[3] M.J. Elynn, *Computer Architecture. Pipelined and Parallel P rocessor Design*, ISBN 0-86720-204-1, Jones and Bartlett Publishers, 1995.

[4] V. Lo, Operating Systems Enhancements for Distributed Shared Memory , *Advances in Computers*, Vol. 39, pp. 191-237, 1994.

[5] Protic, Tomasevic, and Milutinovic, Distributed Shared Memory: Concepts and Systems, *IEEE Parallel and Distributed Technology*, Vol. 4, pp. 63-79, 1996 .

[6] Sarita V. Adve and K. Gharachorloo, Shared Memory Consistency Models: A Tutorial, *IEEE Computer Society Press*, Vol. 29, pp. 66-76, 1996.

[7] M. Dubois, C. Scheurich, and F. Briggs, Memory access buffering i n multiprocessors, *Proc. of the 13th Annual International Symposium on Computer Architecture*, pp. 434-442, 1986.

[8] Yong-Kim Chong and Kai Hwang, Performance Analysis of Four Memory Consistency Models for Multithreaded Multiprocessors, *IEEE Transaction on Parallel and Distributed Systems*, Vol. 6, pp. 1085-1099, 1995.

[9] Susanne Graf, Characterization of a Sequentially Consistent Memor y and Verification of a Cache Memory by Abstraction, *Distributed Computing Journa l*, Vol. 12, pp. 75-90, 1999.

[10] Rob Gerth, Sequential consistency and the lazy caching algorithm, *Distributed computing journal*, Vol. 12, pp. 57-59, 1999.

[11] T. Henzinger, S. Qadeer, and S. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems, *Proc. of the 11th International Conference on Computer Aided Veri*Œ*cation*, LNCS, Vol. 1633, pp. 301-315, 1999.

[12] S. Qadeer, Verifying sequential consistency on shared-memory multiprocessors by Model Checking, *IEEE Transactions on Parallel and Distributed Systems*, Vol.14, pp. 730-741, 2003.

[13] A. Condon and Alan J. Hu, Automatable veriŒcation of sequential consistency, *ACM Symposium on Parallel Algorithms and Architectures*, pp. 113-121, 2001.

[14] P. Chatterjee and G. Gopalakrishnan, A SpeciŒcation and VeriŒcation Framework for Developing Weak Shared Memory Consistency Protocols, *Proc. of the 4th International Conference on Formal Methods in Computer-Aided Design*, LNCS, Vol. 2517, pp. 292-309, 2002.

[15] R.P. Ghughal and G. Gopalakrishnan, VeriŒcation Methods for Weaker ShareD Memory Consistency Models, *Proc. of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, LNCS, Vol. 1800, pp. 985-992, 2000.

[16] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan, Shared Memory Consistency Protocol VeriŒcation Against Weak Memory Models: ReŒnement via Model Checking, *Proc. of the 14th International Conference on Computer Aided Veri*Œ*cation*, LNCS, Vol. 2404, pp. 123-136, 2002.

[17] http://www.inrialpes.fr/vasy/cadp/