Modelling and Model Checking a Distributed Shared Memory Consistency Protocol

Kathi Fisler[1] and Claude Girault[1 2]

[1] Department of Computer Science, Rice University,
6100 Main Street, Houston, TX 77005-1892
[2] University Paris 6, Laboratory of Computer Science LIP6,
4 Place Jussieu, 75252 Paris Cedex 05

Abstract: Distributed Shared Memory (DSM) systems provide the abstraction of a common virtual address space across a network of processors. Such systems employ a variety of protocols to maintain a consistent view of data across all local memories. Li and Hudak proposed several of the pioneering protocols for DSM [LH 89]. We have used both Petri net modelling and model checking to explore some of their protocols. Our work has detected inefficiencies, unstated assumptions, and errors in the original protocol descriptions. This paper presents Petri net models for one protocol at two layers of abstraction. For each model, we describe corresponding specifications for model checking and provide verification statistics. This combination of models and specifications gives different views of the protocol, inspiring greater confidence in the correctness of our analysis than if we had used only one approach.

Keywords: Protocol design and verification, distributed shared memory, memory consistency, model checking, high level Petri nets.

1 Introduction

Processors in a distributed environment can utilize each other's local memories. From a programmer's perspective, however, managing data across shared memory is a complicated and distracting task. Distributed Shared Memory (DSM) systems provide a viable alternative: the abstraction of a common virtual address space for a network of processors (Fig. 1). Based on access requests, DSM systems replicate or migrate data between processors, thus relieving programmers from managing data location. As memory coherence is important for the correct execution of programs, DSM protocols are strong candidates for formal analysis.

Sequential consistency is one of the strongest and most commonly used memory consistency models [AG 96]. It guarantees that all processors see all the data accesses in the same sequential order and also preserves the order of each processor’s accesses. Weaker models, such as release consistency or causal consistency, achieve better performance by exploiting synchronization within the program to increase parallelism and reduce network traffic at the price of more elaborate protocols [KCZ 92, CBZ 95].

Li and Hudak proposed several pioneering and widely quoted protocols to achieve sequential consistency in software for DSM systems [LH 89]. We have explored two of them using formal modelling and model checking. This paper presents our work on

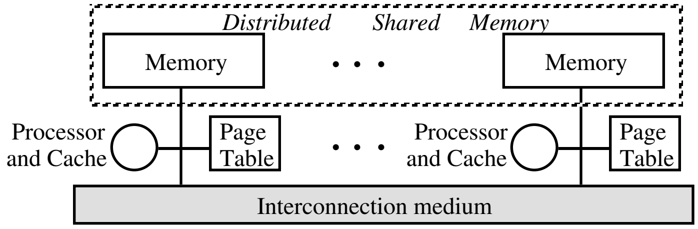

Fig. 1. Distributed Memory Architecture

one of them. Our models follow Li and Hudak's original pseudocode descriptions, adding assumptions only when they are underspecified. We present models at two levels of abstraction. The first considers only data management; the second considers both data and message management. For each level, we provide a high level Petri net description and a series of specifications suitable for model checking. Formal analysis of these identifies unspecified assumptions and errors. The paper demonstrates the relative roles of Petri nets and model checking in locating these problems. It further shows the benefits to using several model checkers (Cospan, Murφ and Spin) for the analysis, due to differences in specification language and underlying semantic models.

Other verification efforts have addressed sequential consistency and similar protocols; Pong and Dubois provide a survey [PD 97]. Our work differs in both the protocols analyzed and the combination of Petri net modelling and model checking. Lindemann and Schön use Petri net modelling for performance analysis of several DSM memory consistency protocols [LS 95]. Gopalakrishnan *et al.* applied model checking to some of Li and Hudak's DSM protocols [GKKR 94]; for the one presented here, we have verified configurations with up to eight processors while their work covered only two processors. Petri net modelling [BG 85] and verification [CGH+ 93] have also been applied independently to cache coherence protocols, which are similar to memory coherence protocols. Blondel *et al.* implemented the protocol discussed here and located several errors [BRRS 98] using distributed debugging tools. Petri net tools such as PEP [G 97], PROD [VHL 97], and Design/CPN [J 95] support model checking. We performed our verifications independently of such tools. Separately developed specifications serve as sanity checks on one another and allow us to investigate what types of errors are easily found with each type of model.

Sect. 2 presents an overview of the Li/Hudak protocol. Sect. 3 presents the Petri net description of the first, more abstract, protocol model. Sect. 4 discusses the formal analysis of this model. Sect. 5 introduces the second model as a refinement of the first and Sect. 6 discusses its formal analysis.

2 Protocol Overview

Li and Hudak's protocol specifies how processors request and receive data. Data are organized into blocks of memory called *pages*. The DSM architecture (Fig. 1) maintains a *page table* on each processor, giving the status of each page relative to the processor. The status indicates the processor's access to the page (nil, read, or write) and whether the processor owns the page. Access is nil if the page is not in local memory or if another processor has modified its local copy of the page. Several processors may have read access to a page at once; write access includes read privileges, but is exclusive. This type of protocol is called multiple-reader, single-writer.

The processor last having write access to a page is its *owner*. The owner, which can change over time, sends the page contents to any other processor that requests them. It also tracks which other processors have read access to the page in a *copy-set*. These processors must be told to *invalidate* their copies before a write to the page occurs.

When a processor wants access privileges that it does not have, it *faults* and invokes a *handler* to request the page. Requests are made via broadcasts to all other processors; accordingly, Li/Hudak call this the *Broadcast Distributed Manager* (BDM) protocol. Processors respond to requests using a *server*. A handler and a server on the same processor may try to access a page simultaneously. To ensure exclusive access, each page table entry also maintains a *lock* which is obtained through a test-and-set technique. Fig. 2 gives the pseudocode for the handlers and servers [LH 89, p. 351] and the invalidation protocol [LH 89, p. 328]. The handler and server definitions are parameterized by the page for which the fault or service request occurred. In addition, the read and write servers take a parameter indicating the processor making the request.

Read fault handler (p) Lock (PTable[p].lock); broadcast to get p for read; PTable[p].access:=read; Unlock (PTable[p].lock);	Read server (p,i) Lock (PTable[p].lock); IF PTable[p].owner=true THEN BEGIN PTable[p].copy set:=PTable[p].copy set∪{i}; PTable[p].access:=read; send p to i; END; Unlock (PTable[p].lock);
Write fault handler (p) Lock (PTable[p].lock); broadcast to get p for write; *invalidate* (p,PTable[p].copy set); PTable[p].access:=write; PTable[p].copy set:={ }; PTable[p].owner=self; Unlock (PTable[p].lock);	Write server (p,i) Lock (PTable[p].lock); IF PTable[p].owner=true THEN BEGIN send p and PTable[p].copy set to i; PTable[p].access:=nil; END; Unlock (PTable[p].lock);
Invalidate (p, copy set) FOR k in copy set DO send invalid request to proc k;	Invalidate server PTable[p].access:=nil;

Fig. 2. Broadcast Distributed Manager (K. Li and P. Hudak)

Li and Hudak state a few assumptions on message communication and system organization. They explicitly assume atomic broadcast of messages [LH 89, p. 334], which guarantees that a message broadcast by a processor arrives at all others before any other message is sent. Simple examples show that starvation and exclusivity violations may occur without this atomicity. Invalidation replies are also needed [L 88, p. 98] to avoid exclusivity violations (otherwise, a processor granted for write could perform an access before the invalidation completes, conflicting with any readers that have not yet invalidated). However, they state no assumptions on the scheduling of handlers and servers within processors. Processors may contain several threads, but only one thread per processor is capable of faulting (the others are system threads). Each such thread has its own handler and each processor has a single server.

Fig. 3 illustrates a write request by processor P_i for a page owned by processor P_j. Dashed lines indicate page locks. P_i's handler broadcasts requests to all servers (*1*). P_j's server sends a grant, the copy set, and the page contents (*2*). The other servers disregard the request (*3*). P_i's handler multicasts invalidations to all the readers in the copy set (*4*) which update to nil access and send acknowledgments (*5*). When all acknowledgments are received, P_i takes exclusive ownership and write access (*6*).

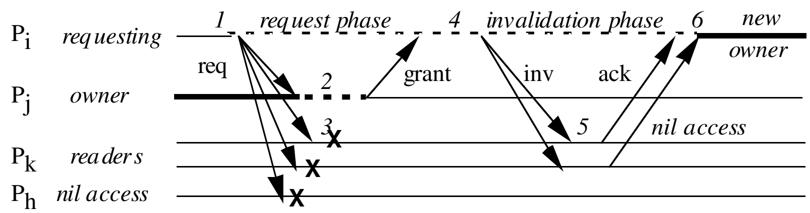

Fig. 3. Protocol Request/Grant Phase and Invalidate/Acknowledgment Phase

The above description suggests three expected safety properties of the BDM protocol:

- **One Owner**: if a processor owns page *p* then no other processor also owns *p*;
- **Exclusive Write**: if a processor has write access to page *p*, then no other processor has either read or write access to *p*;
- **Copy Set Adequacy**: if processor *i* has read access to page *p* but does not own *p*, then *i* is in the copy set for *p* on the processor that owns *p*.

In addition, no processor should deadlock while waiting for a desired access:

- **Request Completion**: if a processor faults for page *p,* it eventually obtains the desired access privileges to *p*.

Request completion is a liveness property. Our analyses explored the above properties, but not sequential consistency. Sequential consistency is irrelevant to the first model because the property relies on the message passing details which only appear in the second model. Sect. 6 discusses sequential consistency in the context of that model.

3. Petri Net Model Without Message Management

Our first model manages locks and page table information as indicated in the pseudocode, but replaces message passing with synchronous rendezvous. Fig. 4 gives a manually developed, colored Petri net [JR 91, J 95] specification of the model. Sect. 3.1 summarizes the notations and describes the places and their initial markings. Sect. 3.2 correlates the places and transitions to the pseudocode in Fig. 3.

3.1 Notations, Initializations, and Places

Let **S** denote the set of processors, **P** the set of pages, and **X** the set of threads on a processor. Elements of these sets are denoted *i*, *p*, and *x* respectively. The model uses several types of tokens, organized around key components of the protocol.

Processors and Threads: A token <i,x> denotes a thread *x* on processor *i*. Place ***TR*** *(Thread Ready)* models the set of threads ready to attempt an access. Initially, for each processor *i* it contains the set of tokens denoted by the formal sum $\Sigma_{x \in X}$ <i,x>. Over all the processors, this yields the double sum $\Sigma_{i \in S} \Sigma_{x \in X}$ <i,x>.

Requests and Access Rights: A token <i,x,r,p> models a request by thread *x* on processor *i* for access *r* (either *W* for write or *R* for Read) to page *p*. Place ***TW*** *(Thread Waiting)* contains tokens modelling the pending requests. It is initially empty.

Page Table: Place ***PT*** *(Page Table)* contains tokens corresponding to the page table entries on all the processors. The page table entry for page *p* on processor *i* is modelled by a token <i,p,li,ai,oi,si> where:

- *li* indicates whether the page is locked on processor *i* (*F* for False, *T* for True);
- *ai* is the access to page *p* of processor *i* (*N* for Nil, *R* for Read, *W* for Write);
- *oi* indicates whether processor *i* is the owner of page *p* (*F* for False, *T* for True);
- *si* is the copy set (the set of processors that have a valid copy of *p*).

We do not model the page contents as they are irrelevant to the control. Li and Hudak do not discuss page table initialization. Any initialization satisfying the desired properties seems reasonable. The Petri net model assumes that each page *p* is owned by some processor *j(p)*. Processor *j(p)* has read access to *p* and all other processors have nil access to *p*. This allows the copy sets to be empty. The initial marking of *PT* therefore consists of the token <j(p),p,R,T,∅> for *j(p)* and the formal sum $\Sigma_{i\in S\text{-}j(p)}$ <i,p,N,F,∅> for all the other processors. The whole set of pages yields the double sum $\Sigma_{p\in P}$ (<j(p),p,R,T,∅> + $\Sigma_{i\in S\text{-}j(p)}$ <i,p,N,F,∅>).

3.2 Transitions

The colored Petri net of Fig. 4 has three parts. The transitions corresponding to data access are towards the left. The transitions for read faults and serves are in the middle. Towards the right are the transitions for write faults, serves, and invalidations.

Data Access Part: Any thread <i,x> available to attempt an access enables transition ***trq*** *(thread request)*. Upon firing, *trq* non-deterministically chooses a page *p* and a desired access *r*, placing a token <i,x,p,r> in *TW* to indicate the pending request. If the page table token for *i* and *p* in *PT* shows the desired access and the page is unlocked, the request enables transition ***tah*** *(thread access hit)*. When *tah* fires, it returns token <i,x> to ready status in place *TR*. If the page is locked, the request token waits at place *TW*. If it corresponds to a fault, the read management or write management parts handle it, as described below.

Read Management: A read fault is enabled at transition ***hrl*** *(handler read lock)* by a request <i,x,r,p> in place *TW* and an unlocked page table token <i,p,F,ai,oi,si> in place *PT* with nil access. After *hrl* fires, the updated page table token shows a locked page and the thread waits for a grant at place ***HRP*** *(Handler Read Prepare)*. Processor *j* can serve the request (and enable transition ***srlg*** *(server read lock and grant)*) if its page table token for *p* is unlocked and shows ownership. After *srlg* fires, the page table token for *p* on *j* is locked. Tokens also move to places ***HRE*** *(Handler Read End)* and ***SRM*** *(Server Read Management)* to enable page table updates for processors *i* and *j*.

Transition ***sru*** *(server read unlock)* updates the page table to finish a read serve. The page table token becomes <j,p,F,R,oj,sj+i> to unlock the page, set the access to read, and add *i* to the copy set. If the page was in write mode, *sj* now includes only *i*. Transition ***hru*** *(handler read unlock)* updates the page table at the end of a read fault. Moving token <i,x> to place *TR* models that the access has been performed. The page table token becomes <i,p,F,R,oi,si> because page *p*, still owned by processor *j*, is unlocked in read mode on processor *i*. Transitions *sru* and *hru* fire asychronously.

Write Management: Write management is similar to read management but includes an invalidation phase. Write faults are enabled at transition ***hwl*** *(handler write lock)*. Token <i,x,p> in place ***HWP*** *(Handler Write Prepare)* models waiting for the grant.

Transition ***swlg*** *(server write lock and grant)* fires when processor *j* can serve a write request for page *p*. The page table token for *p* on *j* is locked and returns to place *PT*. The token in place ***SWM*** *(Server Write Management)* enables transition ***swu*** *(server write unlock)*, which updates the server's page table entry. Token <i,x,p,sj> in place ***HWI*** *(Handler Write Invalidate)* starts the invalidation phase.

Tokens <i,x,p,s> in place *HWI* drive transition ***his*** *(handler invalidate server)*; *s* denotes the set of processors yet to invalidate *p*. As the processors *k* in *s* invalidate and acknowledge asynchronously, they are removed from *s*. Transition ***hwu*** *(handler write unlock)* may fire when its guard *[s=∅]* indicates completion of the invalidation phase. The new page table token has processor *i* as a write-owner with an empty copy set.

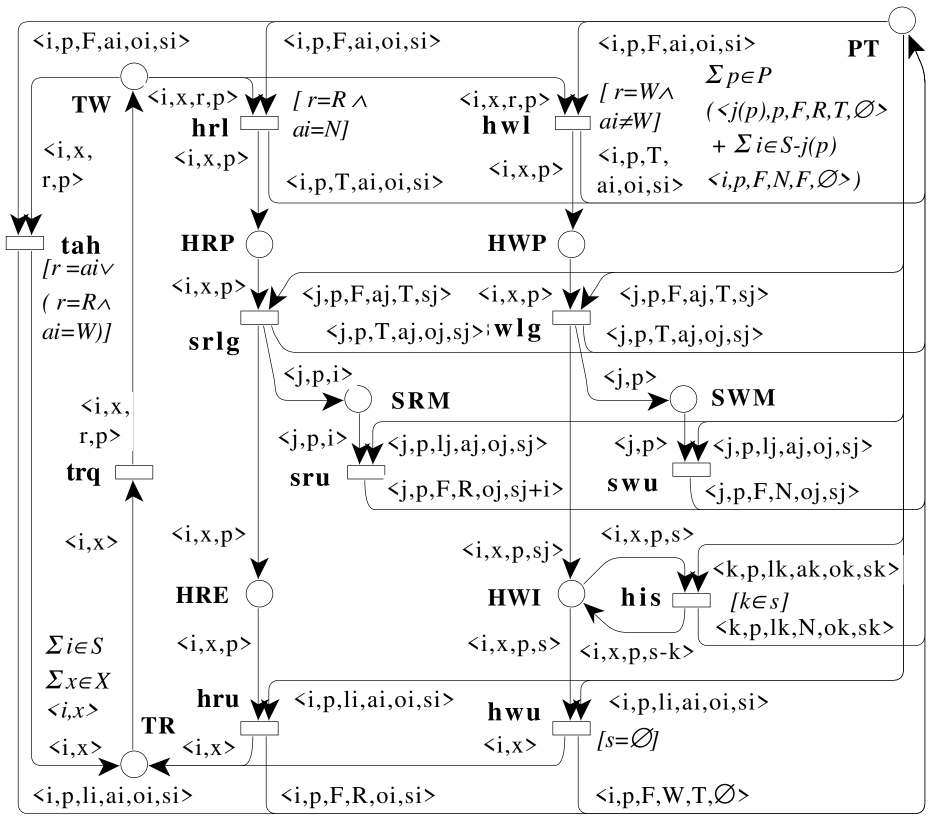

Fig. 4. Original Specification of the First Model

4. Verification of the First Model

This section presents the Petri net analysis and model checking performed on the first model. These analyses located several errors, for which we propose corrections. The discussion refers to *configurations* which indicate the number of processors, threads per processor, and pages in the model being analyzed.

4.1 Petri Net Analysis

P-semi flows allow easy checking of some basic structural invariants and safety properties. They are more general than the safety properties in model checking because they are structural properties of the Petri net, independent of particular configurations. Automatic construction of semi-flows is exponential for non colored Petri nets and still open for general predicate/transition nets [CHP 93, J 95]. If the invariants are known, however, checking is linear in the number of transitions of the net. This section presents flow equations capturing desired invariants for the first model.

Let $(P_1 + ... + P_k)\backslash\langle f=x,\ g=y\rangle$ denote the total number of tokens in places $P_1,...,P_k$ having the same colors x and y for the fields f and g, respectively, regardless of the other token fields. The first two invariant equations denote that (1) threads are never created or destroyed, and (2) there is exactly one page table token per page per processor.

$$\forall i\ \forall x\ \ (TR + TW + HRP + HRE + HWP + HWI)\backslash\langle proc=i,\ thread=x\rangle = 1 \qquad (\mathbf{1})$$

$$\forall i\ \ \forall p\ \ PT\backslash\langle proc=i,\ page=p\rangle = 1 \qquad (\mathbf{2})$$

By equations (3) and (4), each page on each processor is either unlocked or locked but being managed by exactly one handler or server.

$$\begin{gathered}\forall i\ \ \forall p\ \ \ PT\backslash\langle proc=i,\ page=p,\ locked=F\rangle\ + \\ (HRP + HRE + HWP + HWI + SRM + SWM)\backslash\langle proc=i,\ page=p\rangle = 1\end{gathered} \qquad (\mathbf{3})$$

$$\begin{gathered}\forall i\ \ \forall p\ \ \ PT\backslash\langle proc=i,\ page=p,\ locked=T\rangle = \\ (HRP + HRE + HWP + HWI + SRM + SWM)\backslash\langle proc=i,\ page=p\rangle\end{gathered} \qquad (\mathbf{4})$$

The next two equations express page ownership invariants. Each unlocked page should have exactly one owner (the One Owner property of Sect. 2.2). For locked pages, there are transient states during which ownership transfers between processors. Equation (5) says that each page either has an owner or is being transferred to a new owner. In the latter case, tokens at place *HWI* indicate that invalidations are in progress for *p*, while tokens at *SRM* indicate that a processor is about to relinquish ownership.

Equation (6) ensures that all but one processor denies ownership of *p*. Recall that **S** is defined to be the set of all processors. The second summand requires page table tokens for *p* with *lock=T* to correspond to handlers waiting for grants, handlers finishing read requests, or serves about to complete. Handlers finishing write requests are excluded because they result in ownership changes.

$$\forall p\ PT\backslash\langle page=p,lock=F,owner=T\rangle + SRM\backslash\langle page=p\rangle + HWI\backslash\langle page=p\rangle = 1 \qquad (\mathbf{5})$$

$$\begin{gathered}\forall p\ PT\backslash\langle page=p,lock=F,owner=F\rangle \\ + (HRP + HRE + HWP + SWM)\backslash\langle page=p\rangle = |\,\mathbf{S}\,| - 1\end{gathered} \qquad (\mathbf{6})$$

T-semi-flows allow checking of desired stationary sequences. Simple sequences take advantage of symmetries and return to the original symbolic marking under a suitable permutation of node colors [CDFH 97]. Some elementary sequences for the model are:
(1) the hit cycle: *(trq, tah)**,
(2) the transfer of write ownership : *(trq, hwl, swlg, swu, hwu)**,
(3) the ping-pong between a write owner *i* and a reader *j* in which *i* gives *j* read access, then takes write access and invalidates *j*.

Manually analyzing the equations and the sequences uncovers some problems in the BDM description. All reported problems reflect underspecifications, omissions, or errors in Li and Hudak's descriptions, not in our models. We suspect most, if not all, of these problems have been corrected by those implementing this protocol (including Li and Hudak). However, corrections to the protocol have never been published. Our analysis therefore reflects problems in the only available protocol description.

Error 1. Multiple processors can claim ownership for the same page because the write servers fail to relinquish ownership when sending grants. From the semi-flow perspective, transitions *swl*g, *swu*, and *hwu* cannot satisfy equations (5) and (6). The corrected model changes transition *swu*, which now puts token <j,p,N,F,∅> (instead of <j,p,N,oj,sj>) in place *PT*; this indicates *j*'s loss of ownership and resets the copy set.

Error 2. Deadlock occurs if a processor requests write access to a page that it owns with read access. Checking sequence 3 unveiled this error. Transition *swlg*, which needs a processor *j* to serve a request from processor *i*, cannot fire with *j=i* because the token <j,p,lj,aj,oj,sj> in place *PT* has lj=T. The handler for *i* has locked the page, waiting for a grant. The server on *i*, meanwhile, waits for the lock to free in order to service the request. Deadlock results. The pseudocode description therefore needs a special case in the write handler for self-requests. Correspondingly, we add transition ***hwol*** *(handler write owner lock)* to the Petri net (Fig. 5). Firing on tokens from *TW* and *PT* indicating self-ownership, *hwol* locks the page and initiates invalidation by putting a token in place *HWI*. We also add guard oi=F to transition *hwl* to distinguish it from the new transition *hwol.*

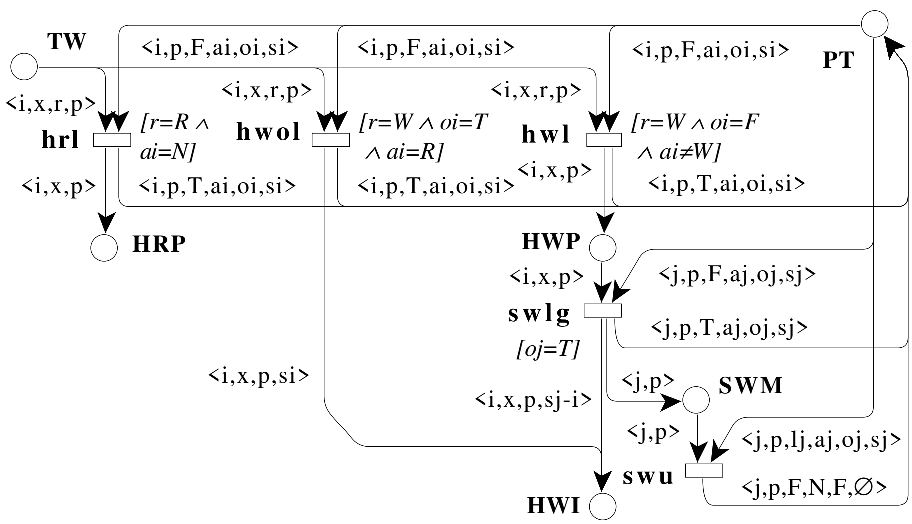

Fig. 5. Corrections for the First Model

Fig. 5 shows the corrections for both errors. It also removes an inefficiency related to invalidations. Consider a processor taking write access to a page for which it had read access. The processor must be in the copy set received from the previous owner. By the pseudocode, the processor therefore sends an (unnecessary) invalidation to itself. Removing the requesting processor from the copy set when the grant is managed avoids the useless invalidation. Fig. 5 adjusts transition *swlg*, putting token <i,x,p,sj-i> (instead of <i,x,p,sj>) in *HWI*. The corrected model satisfies all the desired equations.

4.2 Model Checking

Model checkers vary in numerous dimensions such as semantic models (synchronous or asynchronous), specification languages (hardware oriented or process oriented), property language (temporal logic or automata) and system representation (symbolic or explicit). Different combinations are suited to different problems. Often, the problem to be verified is better suited to a particular semantic model or specification language. The choice between symbolic and explicit system representation is based on both the size of the design and the extent to which variables in the design are mutually dependent. Symbolic representations, such as Binary Decision Diagrams (BDDs), are very compact for many designs. Designs with large degrees of inter-variable dependencies, however, are sometimes more succinct when expressed explicitly as directed graphs.

Most model checkers view systems as sets of simultaneously executing, finite state machines. The full system model is taken to be the cross-product of these machines. As the number of components in a design grows, the number of states in the full model increases exponentially. This situation, known as *state-explosion*, explains the practical upper bound on the sizes of designs amenable to verification. Current model checkers can handle designs with roughly 10^{70} states, sometimes more depending upon the structure of the design in question.

We explored the first model using two model checkers: Cospan [HHK 96] and Murφ [ND 96]. These tools have rather different features and support complementary techniques, as described in the following sections.

Verification with Cospan: We chose the Cospan model checker from Lucent Technologies/Bell Laboratories for several reasons. First, the underlying semantic model is inherently synchronous. This fits our abstraction of messages to synchronous rendezvous between processors. Second, its input language is designed for verification, not synthesis (unlike specification languages such as Verilog, as used in the VIS model checker). The actual connections between the handlers, servers, and page tables may therefore be left implicit, which is consistent with the level of abstraction in the protocol description. Next, Cospan supports symbolic model checking using BDDs, which could be useful for verifying configurations with several processors. Finally, it supports powerful techniques not available in other model checkers, such as the ability to prove whether one machine is a property-preserving refinement of another. Cospan uses LTL properties, though the LTL/CTL distinction is irrelevant for our purposes.

Our specification contains finite-state machines for representing page table entries, threads (with fault handlers), and processors (with request servers). All protocol actions, such as broadcasts, request responses, and page table updates, are performed atomically. While this is a considerable abstraction over the protocol's behavior in practice, it allows us to quickly locate any simple control errors in the model.

Each page table entry machine contains four state variables: *lock_status* (which has values locked or unlocked), *access_right* (read, write, or nil), *owner* (boolean), and *copy_set* (an array of booleans, one for each processor). Each variable is updated according to the protocol description (Fig. 2). Initially, processor zero owns all pages with read access and all other processors have nil access; this is consistent with the initial marking of the Petri net model.

Each thread machine contains three state variables: *status* (one of running, faulted, broadcasting, or invalidating), *faultpage* (any page number), and *faulttype* (read or write). The thread also contains a non-deterministic boolean variable called *fault*. When *fault* is true and the thread status is *running*, the status changes to *faulted* and a particular faultpage and faulttype are selected non-deterministically; these values hold until the fault has been serviced. The model does not consider access hits because the faultpage and faulttype are chosen so as to generate faults only.

Each processor machine maintains a variables corresponding to its server: *serving* (a boolean indicating whether it is serving a request), *serving_to* (a processor number), *serving_type* (read or write), and *serving_page* (a page number). The latter three variables hold their values while *serving* is true and are used for coordination between faulting threads (handlers) and the processors serving their requests.

We do not model messages explicitly. Instead, the machines monitor each other's variables to detect messages. For example, if a thread's status is *broadcasting*, the other processors act as if they had received a request from that thread. As several threads may be broadcasting at once, the servers must uniformly choose one "message" to handle. Given the constructs in Cospan's input language, the only reasonable choice is for each server to see the one broadcast by the lowest numbered thread on the lowest numbered processor (each thread and processor has a unique id).

This modelling decision affects the verification of the liveness property. A thread on processor two may have its broadcast repeatedly ignored if threads on processors zero and one broadcast their own requests at certain times. This problem does not affect processors zero and one. Rather than add a complicated (and potentially expensive) fairness condition, we instead verify the liveness property relative to processors zero and one only. By construction, processors one and two are symmetric. A fair message scheduling protocol should therefore support the liveness property for processor two if it supports it for processor one. The second model (Sect. 5) adds a fair scheduling protocol, so we defer general analysis of the liveness property to that model.

Table 1. Safety and Liveness Checks with Original Information Structures

	one owner				request completion			
u,p,t	reached	BDD nodes	MBytes	CPUsec	reached	BDD nodes	MBytes	CPUsec
2,1,1	1 700	12 165	0.0	16	1 920	11 566	0.0	24
2,1,2	67 957	26 256	2.2	139	105 898	24 650	2.2	74
2,2,1	89 030	31 183	3.2	285	95 901	98 608	13.8	1 033
2,2,2	$2*10^7$	155 076	40.5	3 905				

Errors 1 and 2 (Sect. 4.1) surface almost immediately using a two processor, one thread, two page configuration. The Cospan specification was corrected similarly to the Petri net model. Table 1 shows the verification statistics obtained for various configurations of the corrected model using Cospan version 8.22 on a Silicon Graphics IP19 with 1Gb of memory. As the figures are similar for all the safety properties, we show only those for One Owner. The *u,p,t* column indicates the numbers of units (processors), pages, and threads in the configuration. The *reached* column indicates how many states are actually reachable from the initial state. Model checkers explore only the reachable states (for u,p,t=2,2,2 only $2*10^7$ states of $5*10^{15}$ are reachable). The *BDD nodes* column gives the size of the data structures used in the verification. The *Mbytes* and *CPU sec* columns give memory and time usage statistics.

The table presents only those configurations that were verifiable within the available memory (usually about 500 Mb). Larger configurations face several potential problems with regards to memory usage. First, their state spaces grow rapidly, requiring much more information to be stored. Second, the BDD data structure used for symbolic model checking behaves unpredictably with respect to memory usage. A BDD's size can vary dramatically depending on the order in which it considers design variables. Furthermore, the intermediate computations required in model checking can result in exponential blow-up in the size of the intermediate BDDs. For large designs, this often renders symbolic model checking intractable. From experience, the memory usage leaps in Table 1 are not unusual, nor is the inability to verify larger configurations. The different algorithms needed to verify liveness versus safety properties account for differences in resource usage for properties of the same configuration.

Two optimizations to the Cospan model reduce the number of states and enable us to verify larger configurations. First, a processor only uses the copy sets of pages that it owns. Since a page has at most one owner at a time, we abstract to only one global copy set per page. The copy sets contribute ProcessorCount^2 $\times$ PageCount state bits (ProcessorCount bits for each of ProcessorCount $\times$ PageCount page table entries) in the original specification. Global copy sets reduce this by a factor of ProcessorCount. The statistics in Table 2 show significant savings in the reachable states and resource usage for the larger configurations under the new model. We verified the safety properties simultaneously. We attribute the slight increases in memory usage and time for the safety properties under the smallest configuration to the differences in BDD variable orderings since the numbers of BDD nodes are similar.

Table 2. Safety and Liveness Checks with Global Copy set

	one owner & one writer & copy set				request completion			
u,p,t	reached	BDD nodes	MBytes	CPUsec	reached	BDD nodes	MBytes	CPUsec
2,1,1	937	10 424	0.0	24	1 065	10 458	0.0	23
2,1,2	38 629	23 969	1.1	67	62 626	23 299	2.2	76
2,2,1	27 271	48 586	2.2	131	29 404	55 923	10.6	404
2,2,2	$7*10^6$	504 724	31.0	3 361				
3,2,1	10^7	3764 591	233.0	28 765				

Our second optimization concerns the processor machine variables *serving_to*, *serving_page*, and *serving_type*. Once a server chooses to serve a request from a particular thread, the information kept by these variables can be obtained by looking at the *faultpage* and *faulttype* variables on the thread. The processor number is also accessible through the thread by the structure of the specification. The variables on the server therefore appear to be redundant. We alter the specification slightly, adding a boolean variable *being_served* to each thread machine and removing the *serving_to*, *serving_page*, and *serving_type* variables. Information sharing between thread and server machines is now simple. A server can determine which page it is serving by finding the thread that is being served for a faultpage owned by the server. Since the server can only process one request at a time, only one such thread can exist. Eliminating the server variables makes the specification substantially smaller and allows us to explore larger configurations, as shown in Table 3.

Although this information sharing deviates from real implementations of servers and handlers, such reductions are often necessary to make model checking tractable. This is where the nature of a model checker's specification language becomes important. If the language requires explicit connections between components and the variables they examine (as does a hardware description language like Verilog), making such reductions can require extensive changes to the model. If such connections are left implicit, the variables may be examined without further modification.

Table 3. Safety and Liveness Checks with Global Copy set and Reduced Information

	one owner & one writer & copy set				request completion			
u,p,t	reached	BDD nodes	MBytes	CPUsec	reached	BDD nodes	MBytes	CPUsec
2,1,1	161	13 804	1.1	20	191	14 379	1.1	22
2,1,2	6 103	27 056	2.2	60	15 919	28 089	2.2	75
2,1,3	189 701	52 396	3.5	87	10^6	56 458	3.5	69
2,2,1	3 350	31 183	1.1	51	3 900	31 811	2.1	82
2,2,2	583 969	155 076	9.2	713	10^6	227 582	44.0	1 511
2,2,3	$7*10^7$	288 356	29.8	2 857	$5*10^8$	862 028	181.0	9 759
3,1,1	3 056	30 467	1.1	31	3820	32 819	1.1	46
3,1,2	801 026	60 675	3.7	170	$3*10^6$	80 578	11.0	408
3,1,3	10^8	109 752	8.4	414	$2*10^9$	160 977	23.1	951
3,2,1	368 742	238 346	15.2	1 305	442 999	308 507	25.7	1 956
4,1,1	50 756	72 823	4.5	216	68 423	86 603	6.7	246
4,1,2	$8*10^7$	174 513	13.7	872	$5*10^8$	241 705	44.3	2 712
4,1,3	$8*10^{10}$	231 366	20.2	1 564	$3*10^{12}$	599 200	98.1	7 396
5,1,1	745 296	166 850	20.6	902	10^6	207 370	40.5	1 231

Our Cospan efforts discovered a potential deadlock when multiple threads can fault on a processor. Specifically, a thread *t* can fault for a page *p* that its processor comes to own with *t*'s desired access because another thread on the same processor also faulted for *p*. In this case, *t* should return to normal operation. Li and Hudak do not handle this situation because they considered only one faulting thread per processor. However, supporting multiple faulting threads is straightforward. Handlers waiting for the lock to begin broadcasting need only monitor the page table and return to ready status if their desired access is obtained. The results in Table 3 are for a model using this revision; all properties verify for the configurations shown.

This deadlock cannot occur in the Petri net model because a thread *t* waits in place *TW* for its desired access, at which point it enables transition *tah*. This illustrates the advantage to multiple modelling efforts: different modelling decisions elicit different implementation requirements. In this case, threads in the Petri net model resume when their desired accesses appear in the page table. Threads in the Cospan model wait for direct responses to their requests. Either design could arise in practice.

Verification with Murφ: The processors in this protocol are highly symmetric. Given two processors, neither of which has access to a given page, the resulting system runs are similar regardless of which one faults. A model checker need only explore one run from each equivalence class under symmetry. Cospan does not support symmetry reductions; the Murφ verification tool supports user-supplied symmetries [ND 96].

The Murφ specification uses global copy sets and reduced server variables. It differs from the last Cospan model in a few small details due to differences in the tools and their languages. To enable symmetry reduction, Murφ requires the processor initially owning the pages to be chosen non-deterministically; this is more general than in the Cospan model. Also, unlike the Cospan version, the Murφ version is not restricted to responding to the lowest numbered request. Table 4 shows statistics for verifying the safety properties in Murφ (Murφ does not currently support liveness properties under symmetry) under Murφ version 3.0 on an UltraSparc running SunOS 5.1. We do not report memory usage because Murφ does not give accurate memory figures. Instead, it uses as much memory as the user allows. These runs were each allowed 100Mb, though the smaller runs needed as little as 8Mb. General comparisons between the Murφ and Cospan results are not meaningful here because Murφ uses an asynchronous (interleaving) semantic model and explicit-state, rather than symbolic, model checking. These statistics do, however, demonstrate the benefits to exploiting symmetry.

Table 4. Safety Checks with Murφ

	With symmetry		Without symm.			With symmetry		Without symm.	
u,p,t	Reached	CPUsec	Reached	CPUsec	u,p,t	Reached	CPUsec	Reached	CPUsec
2,1,1	66	0.3	131	0.3	4,1,1	2 176	2.5	43 785	42.6
2,1,2	1 423	1.5	5 645	6.0	5,1,1	8 272	15.0		
2,2,1	774	0.8	3 055	4.0	6,1,1	26 479	73.6		
2,2,2	74 832	301.2			7,1,1	74 445	335.1		
3,1,1	451	0.6	2 560	1.3	8,1,1	189 087	1 176.6		
3,1,2	81 878	312.1							
3,2,1	27 356	49.5							

5. Petri Net Model with Message Management

Our second model extends the corrected first model with message management. At the I/O level, queues store messages that cannot be processed immediately either because the handler or server is busy with another task, or because the request concerns a page which is currently locked in the local page table. Several types of messages arise in the BDM protocol: requests, invalidations, grants, and acknowledgments (to invalidations). The first two activate servers, while the latter two awake waiting handler threads. Li and Hudak neither discuss thread scheduling nor specify whether messages are placed on a single or separate queues.

Our Petri net and model checking specifications use slightly different organizational assumptions on queues. The Petri net model uses a distinct queue for each message type, with one queue of each type per page per processor. The model checking specification uses one queue for requests, one for invalidations and another for grants and acknowledgments to make verification more tractable. This optimization is acceptable because a processor can wait either for a grant or for acknowledgments at any given time. The size of the request and acknowledgment queues is bounded by the number of processors. Since a handler may only await one grant at a time, the grant queue needs only one slot. Similarly, the invalidation queue has one slot since a given page can be invalidated by at most one handler at a time.

The Petri net model has six parts: data access, read handler, write handler, read server, write server, and invalidation server. Fig. 6 shows the data access part and the handlers; Fig. 7 contains the server parts. The two figures share the request queue ***REQ*** and the places ***GR*** *(grants),* ***INV*** *(invalidations)*, ***ACK*** *(acknowledgments)* and of course the page table place *PT*. Each designator represents a complete array of separate places or queues indexed by the processor numbers.

Data Access Part: This part is similar to that in the first model, except faulted requests are directed into queue *REQ* by transition ***tam*** *(thread access miss)*. The second field of the request token <i,i,x,r,p> indicates that processor *i* must manage it.

Handler Read Part: Transition ***hrl*** *(handler read lock)* is the same as in the first model except the input token <i,i,x,r,p> comes from queue *REQ* instead of place *TW*. Transition ***hag*** *(handler already granted)* supports multiple faulting threads; if the requested access has already been obtained, the thread token returns to place *TW*. Firing transition ***hrb*** *(handler read broadcast)* broadcasts read requests from handler *i*. It puts a set of tokens <i,j,p,R> in queue *REQ* for each *j* other than *i*. The token moved from place ***HRP*** *(Handler Read Prepare)* to place ***HRW*** *(Handler Read Wait)* models waiting for a grant. Grants appear as tokens <j,i,p,g> from place *GR*, where *g* indicates the type of grant. When the grant arrives, transition ***hrg*** *(handler read grant)* fires, moving token <i,x,p> from place *HRW* to place *HRE* to update the page table. Finally, transition ***hru*** *(handler read unlock)* is the same as in the first model.

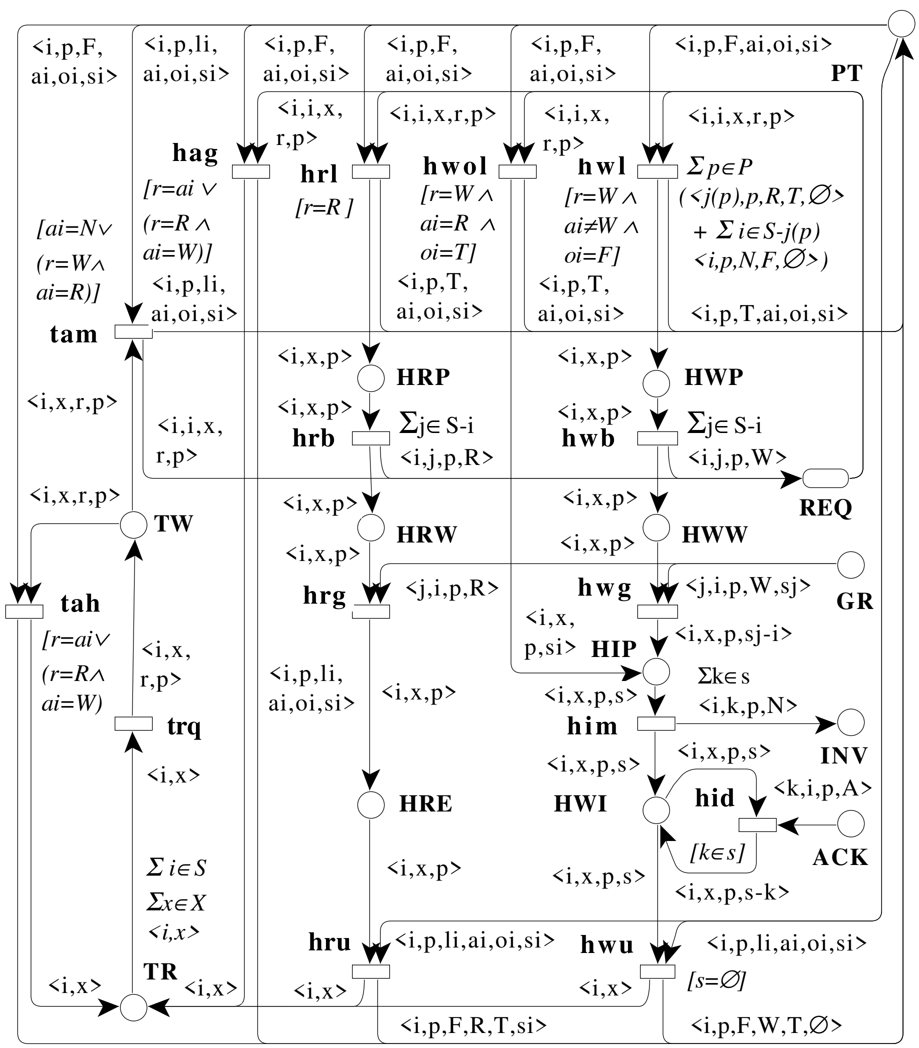

Fig. 6. Communication Refinement. Thread and Handler Parts of the Second Model

Handler Write Part: Transition ***hwl*** *(handler write lock)* is similar to *hrl*. Transition ***hwol*** *(handler write owner lock)* manages write requests when the faulting processor owns the page for read. For other write faults, firing transition ***hwb*** *(handler write broadcast)* broadcasts write requests to other processors through queue *REQ*. The token moved from place ***HWP*** *(Handler Write Prepare)* to place ***HWW*** *(Handler Write Wait)* awaits a grant token <j,i,p,W,sj> from place *GR*; the *sj* field denotes the sent copy set. When this token arrives, transition ***hwg*** *(handler write grant)* fires, moving token <i,x,p,sj-i> to place ***HIP*** *(Handler Invalidation Prepare)* to enable the invalidation phase.

Transition ***him*** *(handler invalidate multicast)* multicasts invalidation messages by putting one token <i,k,p,N> in place *INV* for each *k* in the copy set. Token <i,x,p,s> moves from place *HIP* to place ***HWI*** *(Handler Write Invalidation)* to await the acknowledgments. Transition ***hid*** *(handler invalidate done)* receives acknowledgments from place *ACK* as tokens <k,i,p,A> for some $k \in s$ where *s* appears within a token <i,x,p,s> in place *HWI.* The acknowledging processor *k* is removed from *s*. After invalidation, transition ***hwu*** *(handler write unlock)* is as in the first model.

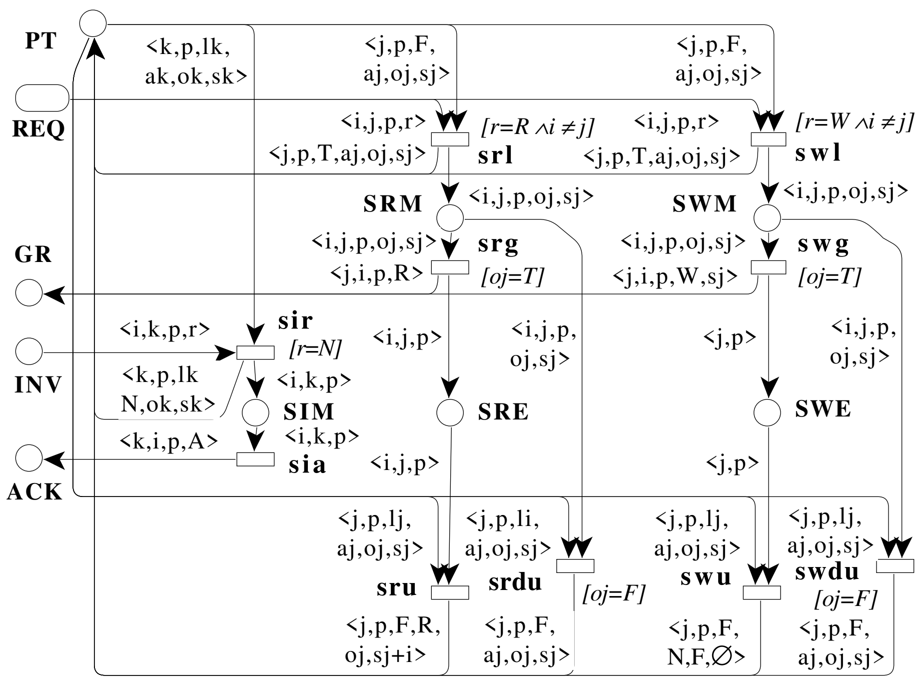

Fig. 7. Communication Refinement. Server Part of Second Model

Server Read Part: Transition ***srl*** *(server read lock)* locks a page to analyze a read request from another processor *[i≠j]* by putting a token into place ***SRM*** *(Server Read Management)*. If the processor does not own the page, transition ***srdu*** *(server read discard unlock)* fires and unlocks the page table entry. If the processor does own the page, transition ***srg*** *(server read grant)* fires, putting a grant token in place *GR* and a token <j,p,i> in place ***SRE*** *(Server Read End)* to update the copy set. Transition ***sru*** *(server read unlock)* updates the page table following a read serve.

Server Write Part: This is similar to the server read part. Transition ***swl*** *(server write lock)* puts a request token into place ***SWM*** *(Server Write Management)* for analysis. Transition ***swdu*** *(server write discard unlock)* fires if the processor does not own the page. If it does, transition ***swg*** *(server write grant)* sends a write grant and puts a token in place ***SWE*** *(Server Write End).* This token enables transition ***swu*** *(server write unlock)*, which completes the serve.

Server Invalidation Part: Transition ***sir*** *(server invalidate reception)* models the atomic updating of the entry for page *p* by server *k* when it receives an invalidation request in the place *INV*. Following the pseudocode (Fig. 2), it does not lock the page or check the condition $ak=R$. After firing, it replaces the page table token by <k,p,lk,N,ok,sk> and puts token <i,k,p> into place ***SIM*** *(Server Invalidation Management)* to enable the acknowledgment. Transition ***sia*** *(server invalidate ack)* sends the acknowledgment by putting a token in place *ACK*.

6. Verification of the Second Model

For verifying the model with message queues, we chose the Spin model checker [H 91] instead of either Cospan or Murφ. Spin provides message queues as a built-in data structure, and is therefore very well-suited to protocols employing message passing. In Cospan or Murφ, we would model a message queue manually using an array of state machines. Each machine would model one position in the queue, and would update its value based on those in adjacent machines whenever elements were added to or deleted from the queue. Spin handles these operations automatically.

Like Murφ, Spin uses explicit-state model checking and an interleaving semantics. However, Spin supports partial-order reductions instead of symmetry. Partial order reductions detect when two transitions can be taken in either order without affecting the truth or falsehood of properties; the model checker explores only one order. They are incomparable to symmetry reductions, but each can yield savings in practice.

Attempting to verify the configuration with two processors, one thread per processor, and one page uncovered a write-exclusivity error. In the following figures, as in earlier figures, thick bars denote ownership and dashed lines indicate page locks.

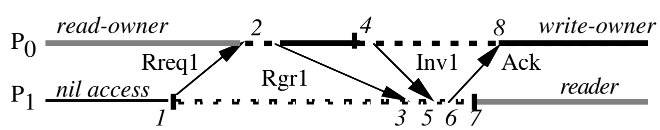

Fig. 8. Dummy Invalidation with Loss of Read-Write Exclusivity

Error 3. A processor may take read access while another processor has write access due to sequencing problems between grants and invalidations. Consider Fig. 8:

- P1 locks and broadcasts read request Rreq1 (*1*).
- P0, the owner, locks the page and returns grant Rgr1 (*2*). P0 unlocks.
- P0 locks to take write access. Since it is the owner, it sends invalidation Inv1 (*4*).
- P1 has both an invalidation (*3*) and a grant (*5*). It looks at the invalidation first and sends an acknowledgment Ack (*6*). Its access does not change.
- P0 receives the acknowledgment and takes write access (*8*).
- P1 sees its grant, takes read access, and unlocks (*7*).

Although this scenario appears to violate assumptions on message ordering, it does not. All message broadcasts were atomic and all access requests were seen in the same order on all processors. The problem arises because P1 had messages waiting in two queues simultaneously. This situation is not unrealistic because processors may run at different rates from one another and from the I/O subsystem. Li and Hudak do not state how to arbitrate such conflicts. The pseudocode in Fig. 2, however, suggests that invalidations should receive immediate attention (invalidations need precedence over requests to avoid deadlock). This example proves such a policy is insufficient.

We tried a modified Spin model, in which grants take priority over invalidations. The safety properties verified for the two processor, one thread, one page configuration (3006 states) in .5 CPU seconds and 4.8 megabytes of memory using Spin version 3.0.0 on an UltraSparc running SunOS 5.1. The number of reachable states for this configuration is higher than for Cospan due to the complexity added by the message queues. Unfortunately, the one owner property fails in a three processor configuration with this modification, so grant priority is also insufficient.

The original Spin model witnesses a separate message-based error, this one affecting the liveness property. We detected Error 4 during manual inspection of the Petri net model given in Fig. 6. As a sanity check, we ran Spin in simulation mode to reproduce the error in the three processors, one thread, one page configuration.

Error 4. Fig. 9 shows a case in which a page becomes permanently unowned by all processors because a requesting processor receives a redundant grant.

- P0 owns the page in write mode. P1 locks and broadcasts a write request Wreq1 (*1*).
- P2 locks for broadcast (*2*) but first receives Wreq1 (*3*) and then broadcasts Wreq2 (*5*).
- P0 receives Wreq1 (*4*) and locks. It queues Wreq2 (*6*) and sends grant Wgr1 to P1 (*8*).
- P0 is no longer the owner, so it discards Wreq2 (X).
- P1 queues Wreq2 (*7*). It receives grant Wgr1 and becomes the owner.
- P1 sends grant Wgr2 to P2 (*9*). P2 receives Wgr2 (*10*) and becomes the owner.
- P2 responds to the queued request Wreq1 by sending a grant Wgr1 to P1.
- P1 receives an unexpected grant (*11*), resulting in no owner.

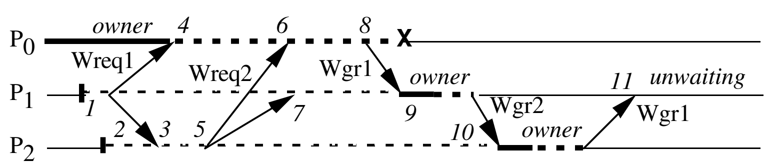

Fig. 9. Redundant Grant with Ownership Loss

Error 4 affects the liveness property because the protocol relies on owners to respond to requests. It provides another example in which message sequencing is insufficient. Clearly, all processors received all messages in the same order, and in the order in which they were sent. However, P2 received Wreq1 after locking to broadcast Wreq2, while P1 received Wreq2 after locking to broadcast Wreq1. Some conditions on locking as well as message sequencing appear necessary.

Unfortunately, we are not aware of any discussions of this protocol that provide a general classification of the necessary message system requirements. Li's IVY prototype for the BDM protocol implements message passing by an I/O sub-system using an Apollo token ring [L 86, L 88]. A token ring contains a slot that circulates to all processors in some fixed order. A processor can send a message if the slot it empty when it passes by. The message circulates to all processors, then back to the original sender, which removes the message from the ring.

Using a token ring could avoid Error 4 in the context of the BDM protocol. As a message passed by in the ring, each processor would either process it or place it in the appropriate queue. Consider processor P2. If it broadcasts a write request on a token ring, it expects to receive that request in its own queue after it has been delivered to all the other processors. Any request P2 sees before seeing its own must have been sent before P2 sent its request. P2 therefore assumes that some prior owner served such a request, and accordingly discards it. This would avoid Error 4.

Using a token ring should satisfy sequential consistency. The token ring guarantees that all processors receive messages from a given processor in the order in which they were sent. It should also guarantee a sequential ordering on all messages sent to all processors. These are the requirements of sequential consistency as stated in Sect. 1. It remains to show in future work, however, that a token ring satisfies the desired properties for the BDM protocol, particularly the liveness property. We have developed a new Petri net model that uses a token ring, but are still analyzing it.

Weaker restrictions than those imposed by a token ring seem plausible for the BDM protocol. Broadcast atomicity is an overly strong assumption for general networks in which mechanisms (such as vector clocks) can be used to insure causal ordering. Furthermore, given the exclusive ownership requirement, classical mechanisms for distributed mutual exclusion may be useful. A system based on time stamps [RA 81], for example, would require all processors to reply to a broadcast and send release messages when leaving their critical sections. Algorithms based on token passing [SK 85] would manage vectors of request counts on all processors and pass these with the grant tokens. Each of these approaches entails more complex management than is suggested in the original protocol description. Refining the necessary assumptions on the I/O system for the BDM protocol requires further experimentation.

7. Conclusion

Modelling and verification have complementary advantages. Modelling develops understanding of a design, while verification increases confidence that it satisfies certain properties. Each is useful for locating errors and uncovering unstated assumptions. This paper illustrates this synergy using one of Li and Hudak's protocols for sequential consistency in a DSM system. We have constructed models of the protocol at two levels of abstraction and in several specification languages. The first level considered page table management; the second also considered the message passing mechanisms.

For the first version, we have presented a compact Petri net model (9 places and 11 transitions) and verified both a Cospan model (340 lines) and a Murφ model (443 lines). These efforts uncovered problems in Li and Hudak's protocol description:

- Two processors may own the same page after a write request.
- A processor deadlocks when requesting write access to a page it owns for read only.
- Two threads on the same processor faulting for the same page can cause deadlock.

The Petri net model also indicated an inefficiency with respect to invalidation requests.

For the second version, we presented an additional Petri net model (19 places and 24 transitions) and discussed verification using a Spin model (262 lines). These efforts show the need for still-unspecified assumptions regarding the message architecture. Underspecification of assumptions was the largest hindrance to analyzing these protocols. General, implementation independent assumptions would be useful.

Analyzing the Petri nets manually locates simple invariant violations. Model checking is better for verifying liveness properties and exploring possible corrections to the original models. The combination of Petri net analysis and model checking is useful because the representations yield models with slightly different semantics. We are therefore more confident that our corrected models and assumptions are sufficient, rather than lucky by-products of modelling decisions in a particular notation.

We have performed similar analyses for Li and Hudak's *Dynamic Distributed Manager* (DDM) protocol, which attempts to reduce the amount of message passing in the BDM protocol. Each processor stores a *probable owner* for each page, to which it sends requests for the page. Servers forward requests for pages they do not own to their stored probable owners. Correctness relies on the chains of probable owners eventually reaching actual owners. We have found no errors in the protocol description, but have identified similar unstated assumptions on message passing that imply correctness.

We plan to explore additional consistency models for DSM systems. In particular, release consistency protocols [AG 96, CBZ 95] and multiple writer protocols [K 92] are distant descendants of the Li/Hudak protocols, sharing many of the same features. We have already begun similar work on the release consistency protocols used in the Munin DSM system [CBZ 91]. Munin also uses probable owners but supports multiple writers with an elaborate page update protocol.

We have also considered writing models for a theorem prover like PVS. This would allow us to explore the protocols over all configurations. More interestingly, however, we could abstract the commonalties in the protocols into libraries and lemmas, hopefully simplifying verification of later extensions to these protocols.

Acknowledgments: The authors thank Willy Zwaenepoel for fruitful comments on DSM, the anonymous reviewers for constructive remarks, and Bob Kurshan and Bell Laboratories for providing access to the latest version of Cospan.

References

[AG 96] Adve, S. V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. In IEEE Computer, 29, 12 (Dec. 1996) 66-76

[BG 85] Baer, J-L., Girault, C.: A Petri net model for a solution to the cache coherence problem. In Proc. First International Conference on Supercomputing Systems, St Petersburg, Florida (1985) 680-689

[BD 91] Barroso, L.A., Dubois, M.: Cache Coherence on a Slotted Ring. In IEEE Trans. on Computers, 44, 7, Berlin, Germany (Sept. 1991)

[BRRS 98] Blondel, X., Rosenfeld, L., Ruget, F., Singhoff, F.: Expériences de Mise au Point d'un Algorithme Réparti. To appear in Technique et Science Informatiques (1998)

[CBZ 91] Carter, J., Bennett, J., Zwaenepoel, W.: Implementation and Performance of Munin. In Proc. 13th ACM Symposium on Operating System Principles (1991) 152-164

[CBZ 95] Carter, J., Bennett, J., Zwaenepoel, W.: Techniques for Reducing Consistency-related Communication in Distributed Shared-memory Systems. ACM Trans. Comput. Syst. 13, 3 (Aug. 1995) 205-243

[CGH 86] Chatelain, C., Girault, C., Haddad, S.: Specification and Properties of a Cache Coherence Protocol Model. In Proc. 7th European Workshop on Application and Theory of Petri nets, Oxford England (June 1986), Advances in Petri nets 87, LNCS., Vol. 266, Springer-Verlag (1987) 1-20

[CDFH 97] Chiola, G., Dutheillet, C., Francheschinis, G., Haddad, S.: A Symbolic Reachability Graph for Coloured Petri Nets, Theoretical Computer Science, Vol. 176 (1997) 39-65

[CGH+ 93] Clarke, E., Grumberg, O., Hiraishi, H., Jha, S., Long, D., McMillan, K., Ness, L.: Verification of the Futurebus+ Cache Coherency Protocol. In Proc. Conf. on Computer Hardware Description Languages and their Application. North-Holland (1993)

[CHP 93] Couvreur, J.M., Haddad, S., Peyre, J.F.: Generative Families of Positive Invariants in Coloured Nets Sub-Classes. In Advances in Petri nets 87, LNCS., Vol. 674, Springer-Verlag (1993) 51-70

[GKKR 94] Golpalakrishnan, G., Khandekar, D., Kuramkote, R., Nalamasu, M.: Case Studies in Symbolic Model Checking: Verification of an Arbiter and DSM Protocols. Tech. Report UUCS-94-009, Dept. of Computer Science, Univ. of Utah (March 1994)

[G 97] Grahlmann, B.: The PEP Tool. In Proc. 9th Int'l. Conf. on Computer-Aided Verification, Haifa, Israel, LNCS., Vol. 1254, Springer-Verlag (1997) 440-443

[HHK 96] Hardin, R. H., Har'El, Z., Kurshan, R.P.: COSPAN. In Proc. Int'l. Conf. on Computer-Aided Verification, LNCS Vol., 1102, Springer-Verlag (1996) 423-427

[H 91] Holzmann, G.: Design and Validation of Computer Protocols, Prentice Hall (1991)

[JR 91] Jensen, K. Rozenberg, G. (eds.): High Level Petri Nets, Theory and Applications, Springer-Verlag (1991)

[J 95] Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 1, 2 and 3, Springer-Verlag (1995)

[J 96] Jensen, K.: Condensed State Spaces for Symmetrical Coloured Petri Nets, Formal Methods in System Design, Vol. 9, 7 (1996) 7-40

[K 92] Keheler, P. : The Relative Importance of Concurrent Writers and Weak Consistency Models. In Proc. 16th Int'l Conf. on Distributed Computing Systems, IEEE Computer Society Press (May 1992) 91-98

[KCZ 92] Keheler, P., Cox, A.L., Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory. In Proc. 19th Annual Int'l Symposium on Computer Architecture (May 1992) 13-21

[L 86] Li, K.: Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD dissertation, Dept. of Computer Science, Yale University, New Haven, Conn., Tech Rep YALEU-RR-492 (May 1992)

[L 88] Li, K.: IVY: A Shared Virtual Memory System for Parallel Computing. In Proc. 1988 Int'l Conf. on Parallel Processing, Vol. II (1988) 94-101

[LH 89] Li, K. Hudak, P: Memory Coherence in Shared Virtual Memory Systems. ACM Trans. on Computer Systems, Vol. 7, 4 (Nov. 1989) 321-359

[LS 95] Lindemann, C., Schön, F.: Modeling Relaxed Memory Consistency Protocols. In Proc. 8th Int'l Conf. on Modelling Techniques and Tools for Computer Systems Performance Evaluation, Heidelberg, Germany (Sept 1995)

[ND 96] Ip, C. N., Dill, D. :Better Verification Through Symmetry. In Formal Methods in System Design, Vol. 9, 1/2 (1996) 41-76

[PD 97] Pong, F. Dubois, M.: Verification Techniques for Cache Coherence Protocols. ACM Computing Surveys, Vol. 29, 1, (March 1997) 82-126

[RA 81] Ricart, G. Agrawala, A.: An Optimal Algorithm for Mutual Exclusion. Comm. ACM, Vol. 24, 1, (Jan. 1981) 9-17

[SK 85] Suzuki, I. Kasami, T.: A Distributed Mutual Exclusion Algorithm. ACM Trans. on Computer Systems, Vol. 3, 4 (Nov 1985) 344-349

[VHL 97] Varpaaniemi, K., Heljanko, K., Lilius, J.: Prod 3.2 - An Advanced Tool for Efficient Reachability Analysis, In Proc. 9th Int'l. Conf. on Computer-Aided Verification, Haifa, Israel, LNCS, Vol. 1254, Springer-Verlag (1997) 472-475