[Cla93] E. M. Clarke, O. Grumberg, H. Hirashi, S. Jha, D.E. Long, K.L. McMillan, and L. A. Ness, “Verification of the Futurebus+ cache coherence protocol”, Proc. 11th Intl. Symp. on Computer. Hardware Description. Lang. and their Application, 1993

[Col92] W. W. Collier, “Reasoning about Parallel Architectures”, Prentice-Hall, Englewood Cliffs, New Jersey, 1992

[Gal92] M. Galles, E. Williams, “Performance Optimization, Implementation, and Verification of the SGI Challenge Multiprocessor”, Hot Chips Symposium, Stanford, 1993.

[Gha90] K. Gharachorloo, D. Lenoski, J.Laudon, P. Gibbons, A. Gupta, and J. Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors”, Proc. 17th Ann Int’l Symp. on Computer Architecture, ACM, pp. 15-26, 1990.

[Gha93] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessey, and M. D. Hill, “Specifying System Requirements for Memory Consistency Models”,University of Wisconsin-Madison Comp. Sci. Tech. Report #1199.

[Gor88] M. J. C. Gordon (ed), “HOL: A Proof-Generating System for Higher-Order Logic”, Kluwer SECS 35, pp. 73-128, 1988.

[Gup92] A. Gupta, “Formal Hardware Verification Methods: A Survey”, Formal Methods in System Design”, Vol. 1, 2/3, pp. 5-92, Oct. 1992.

[Hei94] Joe Heinrich, “MIPS R10000 Microprocessor User’s Manual”, MIPS Technologies, Inc., 2011 N. Shoreline, Mountain View, CA, 1994

[Kur94] R. P. Kurshan, “Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach”, Princeton University Press, 1994

[Len92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D Weber, A. Gupta, J. Hennessy, M. Horowitz, M. Lam, “The Stanford Dash Multiprocessor”, IEEE Computer, vol. 25, pp. 63-79, March 1992.

[Lon93] D. E. Long, “Model Checking, Abstraction and Compositional Verification”, Ph.D. Thesis, CMU 1993

[Low90] P. Lowenstein, D. L. Dill, “Verification of a Multiprocessor Cache Protocol using Simulation Selations and Higher-Order Logic”. Formal Methods in System Design”, Vol. 1, Num 4, Dec. 1992, pp. 355-383

[McM91] K. L. McMillan, J. Schwalbe, “Formal Verification of the Encore Gigamax cache consistency protocol.”, Int. Symposium on Shared Memory Multiprocessors, 1991.

[McM93] K. L. McMillan, “Symbolic Model Checking”, Kluwer Academic Publishers, 1993

[Seg93] C. J. Seger, R. E. Bryant, “Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories”, Tech. Report 93-8, Dept. of Computer Science, University of British Columbia, Aug. 1993.

[Tan95] A. S. Tanenbaum, “Distributed Operating Systems”, Prentice-Hall, 1995

[Yoe90] M. Yoeli, “Formal Verification of Hardware Design”, IEEE Computer Society Press, Los Alamitos, CA 1990.

error prone. It was demonstrated repeatedly, that SMV is a very powerful and efficient specification proof-reader.

Model checking with SMV has sufficient capacity to analyze complete specifications of real-life directory based cache coherency protocols, within the aggressive time schedule of a computer design project. It helped us find several problems in the protocol specification. Most of the problems would have been found in simulation, but the problems involving the interaction of processors and the I/O subsystem would only have been found at the tail end of verification, leading to a disproportional disruption in project schedules. There were several problems that would probably not have been found in simulation. Finally, one protocol problem, that would never have been found in simulation, led to loss of cache coherency, and subsequently due to the subtle symptoms, might not have been found in the test lab!

Model checking has not progressed to the state where a designer can analyze a complex protocol design with SMV at the same time that he/she is designing the protocol; the two activities together are too time consuming. This is, in part, because the usage of SMV (and other model checkers) is not a turn-key activity. In our experience, knowledge of the behavior of BDDs, and the inner workings of SMV are crucial, when creating tractable models. At the same time, formal analysis with SMV requires the use of structural, behavioral, data, and temporal abstraction, which can only be accomplished if the person designing the SMV model has a detailed understanding of the system being modelled. This leads us to conclude that, to successfully adopt formal verification/analysis in industry, requires an organizational re-alignment. The formal verification specialists, that typically have been members of a CAD/CAE group, have to become more involved in the design process, in order to understand the designs well enough, so they are able to create tractable formal verification models.

7. Open Issues & Future Research

We plan to verify that the processor together with the cache coherence protocol implements the sequentially consistent and release consistent memory models, by mapping the protocol model onto the verification framework presented in [Gha93].

Because of model size constraints we were not able to verify the protocol for larger configurations than 4 clusters, not for more than 1 cache line, nor with blocking in the interconnect.

8. Acknowledgment

The authors are indebted to Jim Laudon of **Silicon Graphics Inc.**, the designer of the cache coherency protocol. The formal verification of the protocol would not have been successful without his strong support.

9. References

[Adv93] S. V. Adve, “Designing Memory Consistency Models For Shared-Memory Multiprocessors”, Ph.D. Thesis, U of Wisconsin-Madison, 1993.

[Bry86] R. E. Bryant, “Graph Based Algorithms for Boolean Function Manipulation”, IEEE Trans. on Comp., C-35, pp. 677-681, 1986.

[Bry91] R. E. Bryant, D. L. Beatty, and C. J. Seger, “Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation”, Proc. 28th ACM/IEEE Design Automation Conf., 1991

any implementation of the protocol. These assumptions were uncovered/discovered during the development of the SMV model. These assumptions remained unchanged during subsequent protocol revisions.

The cut & paste typos are the errors introduced after a change in the protocol, and are caused by the designer using a particular message response as a template for the response to a different message, i.e. cutting, and then pasting in transactions and modifying only message types. Every time that there was a change in the protocol, there were a couple of problems caused by the fact that there were also minor changes required, in addition to the message name change. It is very efficient to use SMV to check for these types of the problems with an AG(*never an unexpected input condition to a table*) specification. In fact, once we were convinced of the utility of SMV, one of its uses, was to proof read the modified specification!

The protocol revision lint errors introduced after a change in the protocol, are the errors caused by changing a certain transaction, but not changing all of the affected/implicated transactions. Every time that there was a change in the protocol, there were also a couple of these types of problems. These are handled the same as the cut & paste typos.

The variables not reset at the conclusion of a transition problems, all occurred in I/O section of the protocol. The I/O section of the protocol has several transactions that involve two request response pairs, that use the same CRB entry. There were some cases where all the relevant status bits in the CRB were not reset correctly at the conclusion of the first request response pair.

The unanticipated input conditions are those that cause a combination of input values to a state machine that was not anticipated in the protocol specification. Most of the errors of this type were uncovered in model configurations with both processor and I/O clusters. The early detection of these problems using formal verification is a major improvement over the previous methodology. In the previous methodology these problems would not have been discovered until system simulations of the processor, memory and I/O sub-systems, at the tail end of the verification phase of the project.

There were several protocol race conditions that lead to erroneous behavior, i.e. broke protocol invariants, and led to deadlock; and one condition that led to loss of coherency. Interestingly only one of the deadlocks was uncovered by an AG EF type of specification, all the others were uncovered by AG *invariant* specifications, or an AG (*outstanding request but no messages in flight*) specification. The deadlock discovered by the AG EF specification was uncovered before we developed the method for transforming deadlock detection to a safety property.

The counter-example in the case of the loss of coherency was discovered in a 3 processor model, and consisted of 19 steps, with loss of coherency only occurring if the intervention forwarding mechanism was used in a certain step, and the outstanding protocol messages arrived in a certain order. This problem would never have been found in simulation, and because of the symptoms, i.e. loss of coherency would have been exceedingly difficult to trace on the test floor.

6. Conclusions

An unexpected advantage to using SMV is due to the size of the specification. The protocol was revised several time, and manual editing of the specification proved to be

processor has a cached copy; and if a processor has a *SHD* copy, then the directory has the bit the particular processor set in directory bit-vector. The third type of AG specification verifies the invariant cache relationships for all states in reachable state set.

At a particular time step the H, P, and I/O modules either non-deterministically issue a new request or remain idle. This means that outstanding transactions can finish without producing new requests. The final AG specification verifies that requests that have been issued are not deadlocked, by verifying that if one of the processor or I/O nodes has an outstanding read or write request, then there are always some messages in flight. The transformation of deadlock detection to a safety property is important because safety properties are checked during the reachable state space exploration, and deadlock is therefore detected faster.

The following four types of absence of deadlock properties were also verified: it is always possible for the processors at some future time to issue each type of processor request, it is always possible for the directory to transition into any of the possible directory states through some chain of events, each of the processor caches can always transition into any of the possible processor cache states through some chain of events, and each of the request buffers can always be allocated and freed through some chain of events. These properties are verified by the following specification AG EF *cond*, where *cond* is one of the above conditions.

The correctness property verification involved verifying that each processor and I/O request always eventually receive the correct response, i.e. AG(request → A(RRB/CRB allocated U expected response). For the processors request ε {*RDSH*,*RDEX*,*UPGRD*}; and for the I/O subsystems, request is a read or write request for partial and full cache line, and replacement/writeback of a cache line in the I/O cache. The correctness properties require a fairness constraint to prevent starvation caused by messages not being delivered to the different clusters. The constraint that is used for each message buffer is the following: FAIR ($\overline{\text{buffer empty}}$ & message delivered | buffer empty), i.e. if there is a message in flight, it is eventually delivered.

The atomic write property is verified by using the synchronous write specification AG(data & data-valid → AG(data-valid → data)) presented in [McM93], and invoking the equivalence between write synchronization and write atomicity theorem, presented in [Col92] (the theorem assumes that read atomicity does not have to be obeyed).

5. Protocol Problems

We expected at the outset that formal verification methods would add value by uncovering race condition, deadlock, and livelock problems. But formal verification offered several unforeseen benefits, and the type of problems that we found are more diverse than we expected. The problems can be divided into the following seven different groups: hidden/implicit assumptions in the specification, cut & paste typos, protocol revision lint, variables not reset at the conclusion of a transition, unanticipated state machine input conditions, protocol race conditions leading to a violation of a protocol invariant, and race conditions leading to protocol deadlock. We didn't find any unanticipated livelock problems, i.e. no livelock conditions that aren't caused by an infinite loop of NACK response messages.

The hidden/implicit assumptions of the protocol specification are those parts of the protocol that are not explicitly in the protocol specification, but are assumed to exist in

cache coherence property is an example of a protocol safety property, and it is verified using a CTL specification of the form AG(*cond*), where *cond* is a boolean predicate (no temporal operators) that expresses the relation between the state of cache lines in different processors. The absence of deadlock is verified with a specification of the form AG(EF *cond*), where *cond* is each of the possible processor, and directory states, and an allocated or free RRB, WRB, or CRB entry. The correctness properties are verified using a specification of the form AG(request → A(buffer entry allocated U response received)).

The version of SMV that we used, verifies the safety properties while the state space is being explored, and checking the correctness properties is more time consuming than checking the deadlock properties. It is therefore most efficient to verify the safety properties first, and to proceed within Table 1, from top to bottom, and from left to right.

	safety properties	deadlock	correctness	atomic writes
1H+2P	ν	ν	ν	ν
1H+1P+1I/O	ν	ν	ν	–
1H+2I/O	ν	ν	ν	–
1H+3P	ν	ν	ν	ν
1H+2P+1I/O	ν	ν	ν	–
1H+3I/O	ν	ν	ν	–

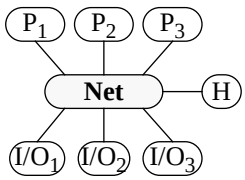

Table 3: configurations that were verified

The following four types of safety properties were verified: expected state machine input conditions, protocol message invariants, cache coherence invariants, and a special case of deadlock.

The specification of the protocol only contains the valid input conditions to each of the different state machines. A state machine input error function is derived from each state machine specification; the error function returns false for the valid input conditions, but true otherwise. The first type of AG specification verifies that an invalid input condition never occurs in the set of reached states.

It is an invariant of the protocol that each cluster, for a given cache line, can at any time only have at most one outstanding request to a particular cache line, that there can be at most one invalidate/intervention targeting a particular cluster, that the number of invalidate acknowledges can never exceed the number of processors in the model, and that there can only be one directory cache line ownership transfer message in flight at any time. These invariants are used to minimize the amount of buffering in the interconnect: there is only one request message buffer per cluster, and one ivn/inv buffer per cluster. If there is more than one request from the same cluster, or more than one ivn/inv targeting the same cluster in flight at the same time, the respective buffer will overflow. The second type of AG specification verifies that the interconnect buffers never overflow; that the message invariants are true for each state in the reachable state set.

The cache coherency of a cache coherency protocol is verified with the invariant relations between the state of a cache line in the different processors; and the invariant relations between the state in the processors and the directory. An example of these properties is the following: if one of the processors has a *CEX* or *DEX* copy, then no other

tends to be strongly correlated with the state of the sender of that message (and sometimes with the receiver). For this reason, allocating an arbitrary unused buffer to a new message will be very inefficient, since the message sender and message buffer may be arbitrarily far apart in the BDD variable order. Instead, it is much more effective for the user to define a function that assigns messages to buffers based on their content.

In making this assignment, there are two important considerations. First, messages must be assigned to buffers in such a way that buffers holding information correlated with the state of a particular P or I/O cluster can be located near that cluster in the BDD variable order. To some extend, the need to store messages near their source or destination has to be traded off against the need to have as few buffers as possible. Second, we have to ensure that there is always a buffer available for a given message. This can be done based on known invariants of the protocol, and verifying as a safety property that no message is ever blocked. As an example, a request message and the corresponding response message typically cannot be in flight at the same time, therefore they may be assigned to the same message buffer.

Finally, when using BDD-based model checking, it is important to minimize the amount of global communication (as opposed to local or nearest-neighbor communication) that occurs in one step of the model. For this reason, we used an *interleaving* model, in which only one (non-deterministically chosen) message is delivered at each step. This can result in up to three new messages being generated. Since messages are typically buffered near the sender, however, the typical amount of *global* communication is just one message in a given step.

For the protocol model, the H cluster has one input message buffer; and the P and I/O clusters require 3 message buffers each, see Figure 3. The H input buffer is used to store a directory ownership transfer message. The P and I/O clusters use one buffer to store requests originating from the particular cluster, and destined for the H module, and also use the same buffer for response, and backed-off intervention messages. The second buffer is used for response, and backed-off invalidate messages; and the third buffer for intervention requests. Three message buffers are required. This can be seen for example by adding P3 to the model shown in Figure 2. Then after H receives *XFER*, but before P2 receives *ESPEC*, or *EACK*, it is possible that H receives an *RDEX* from P3, that causes an *IRDEX* to be sent to P2.

4. Verification Goals

For a protocol model with 1 cache line, with a 1-bit data value, and multiple P and I/O clusters, our goal is to verify the following properties: there's no deadlock in the protocol; all the different processor requests, always receive the correct response; there is never unsolicited response; the protocol guarantees cache coherence between the different processor and I/O caches; and the protocol implements atomic write-backs *WB*.

Table 3 shows the different types of configurations, and properties that were verified. It was not practical, because of run times, to verify larger configurations than 4 clusters. The 1H+3P configuration requires 1G bytes of memory, 67 hours to explore the reachable state space (100MIPS92, 150MHz Challenge machine with 1280M of memory), and each fixed-point iteration takes 20-40 minutes.

The properties that were verified can be classified as safety properties, absence of deadlock properties, correctness properties, and memory consistency properties. The

ifying data consistency properties, and for configurations with I/O modules, each of the H and P message buffers contain an extra bit, due to the larger number of messages.

The H cluster receives requests from the P and I/O clusters, and also receives directory ownership transfer responses. The P requests consist of the following: read shared *RDSH*, *RDEX*, *UPGRD*, and *DEX* cache line writeback *WB*. The I/O requests include read and write requests for partial and full cache lines; and a replacement/writeback of a cache line in the I/O cache. The H model decides non-deterministically to use the inter-

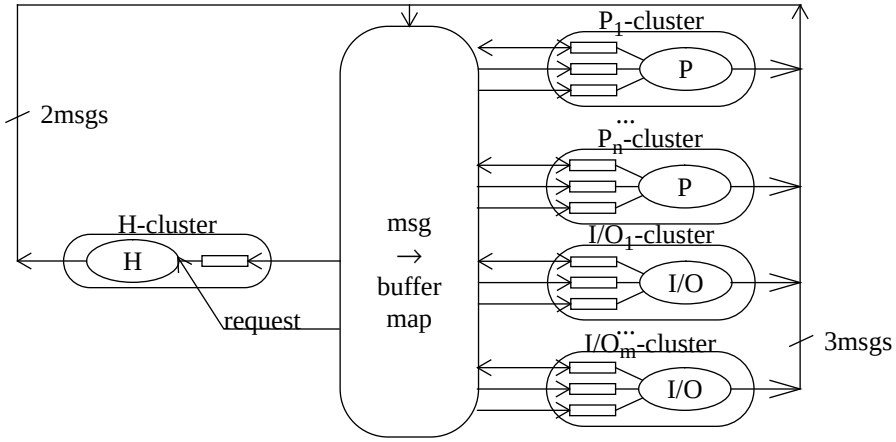

Figure 3: Directory Protocol Model

vention/invalidate back-off mechanism. A P and I/O cluster in addition to issuing requests, forward backed-off ivn/inv requests; and respond to request acknowledge, data responses, and ivn/inv messages.

The structure of the interconnection network model is very important in making property verification tractable using SMV. It is essential not to introduce unnecessary detail in the interconnect model when verifying the protocol – first so as not to add unnecessary complexity to the verification, and second, so that the protocol is verified under the most general possible conditions in which it is intended to operate. For these reasons, the network is modeled simply as a collection of message buffers. The decision as to which messages (if any) to deliver at any given time is non-deterministic. Thus, the time from sending a message to its delivery is completely arbitrary. This is acceptable, since the protocol does not rely on any ordering properties of the network. If no message is delivered at a given time, a P or I/O cluster is non-deterministically chosen to issue a request; it either selects one of the possible requests or stays idle. This ensures that the interconnect can drain all messages in flight to their destination without producing new messages.

There are other issues in modeling the network that relate specifically to the fact that SMV uses Binary Decision Diagram (BDD) [Bry86] based model checking. This has several implications for the model structure.

First, to represent the reachable state set efficiently using a BDD, we need to order the state components of the model such that, if two state components are related, then they are close to each other in the order. In this case, the contents of a given message

input, and next state values. The format line consists of a delimiter (#), the input variable names, a delimiter marking the end of the inputs (>), followed by the output variable names. Table 1 shows a section of the directory memory state machine specification. The table fragment shown in the figure is simplified in that the rows only produce one output message, there can be at most two; and transitions shown don't involve issuing of invalidates/interventions (inv/ivn) and the possible forwarding of the issuing of the inv/ivn. The actual directory table has nine more output variables then shown in Table 1. The format line in the Table specifies the following three input variables: input message *IMsg*, current directory state *DirSt*, and if the bit in the bit-vector is set for the requesting cluster *MyBit*. The format line specifies the following six output variables: the new state of the directory *NewSt*, the new value of the bit in the bit-vector *NewV*, the response message *OMsg1*, the source of that message *OSrc1*, the destination of that message *ODst1*, and the memory operation *MemOp*. The *only* operation in the *MyBit* column specifies that only the requesting cluster has a cached copy. The *set* operation in the *NewV* column specifies that the bit-vector for the requesting cluster is set, in addition to any that might have been set. The *force* operation specifies that the ID of the requesting cluster is stored in the bit-vector; the bit-vector is used to store a cluster ID except in the case where the directory state is *SHRD*.

The first row of the table specifies that a *RDSH* request to the *SHRD* directory state, results in the bit of the requester being set in the bit-vector, a read being issued to the memory, and a *SRPLY* being sent back to the requester. The second row specifies that a *RDEX* request to the *UOWN* state results in an exclusive reply complete *ERPC*; complete because there aren't any interventions or invalidates issued. The third row specifies that

	request tbl. (rows/cols)	response tbl. (rows/cols)	#message buffers	#state bits
H		138/19	1	9+2#P+logN
P	17/13	19/11	3	33+6×logN
		109/29		
I/O	15/13	263/34	3	42+6×logN
	19/13			

Table 2: H, P, and I/O model characteristics

a writeback request *WB* to the *EXCL* state, where the requester has the *EXCL* copy, results in a transition to the *UOWN* state, a writeback exclusive acknowledge *WBEAK* to the requester, and the cache line being written to memory.

There are 7 tables that specify the protocol, with a total of 580 different input conditions, and 2950 different output variable value *changes*. The SMV model of the protocol consists of 6000 lines of SMV code: 5000 lines of protocol transitions, and 1000 lines modelling the clusters, and the interconnection network, between the clusters.

The model consists of the following three types of clusters: directory cluster H, processor cluster P, and an I/O cluster. The H, P, and I/O models consist of translated versions of the protocol tables (to SMV format), and logic to respond to messages, and in the case of the P and I/O models, logic to generate the possible types of requests. The characteristics of each model is summarized in Table 2, and the directory protocol model in Figure 3. The N in the table, is the sum of the number of H, P, and I/O modules. The protocol model contains 1 cache line. A data bit is added to each model when ver-

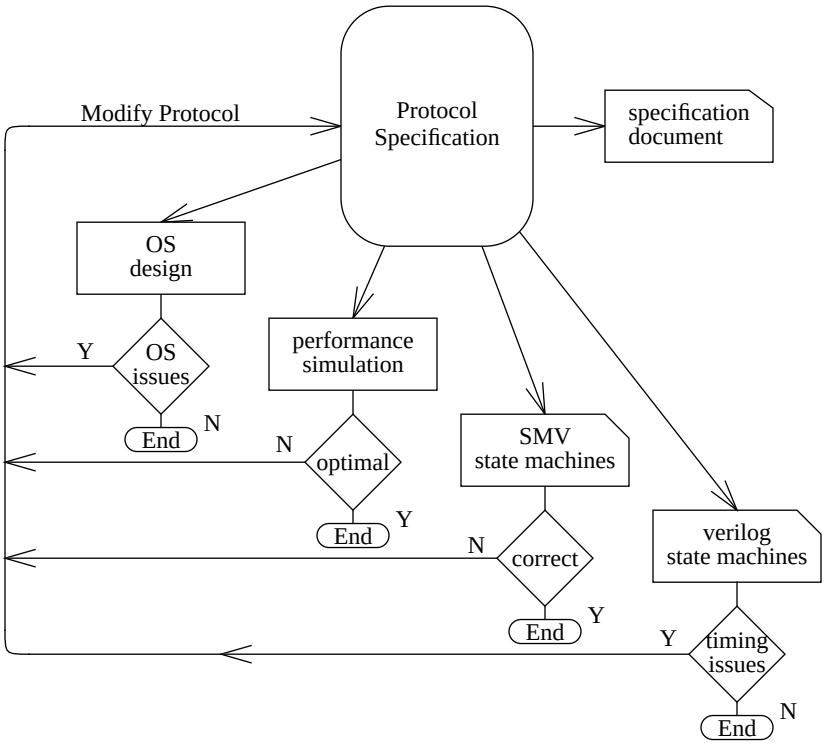

Figure 5: design workflow

performance simulations uncovered several missing transitions, and deadlock conditions. The formal verification of the protocol was started at the tail end of the performance evaluation, and in turn the RTL simulation at the tail end of the formal verification.

After performance evaluation trade-off simulations were completed, the protocol went through several revisions. The three primary driving forces behind the changes were the following: Operating System (OS) requirements, RTL synthesis timing issues, and protocol problems discovered by formal verification.

Having one source for the protocol specification has several important benefits. The first is that that the different tools are always working with the same version of the protocol. Another is that it is possible to regression verify with the formal verification tool after each revision to the protocol. Finally, once formal verification finds a problem in the protocol, and regression verifies the proposed modification, the revised version of the protocol is immediately available to the RTL simulator.

#IMsg	*DirSt*	*MyBit*	>	*NewSt*	*NewV*	*OMsg1*	*OSrc1*	*ODst1*	*MemOp*
RDSH	*SHRD*	any	>	nop	*set*	*SRPLY*	src	src	RD
RDEX	*UOWN*	any	>	*EXCL*	*force*	*ERPC*	src	src	RD
WB	*EXCL*	*only*	>	*UOWN*	clear	*WBEAK*	src	src	WR

Table 1: state machine input format

The protocol specification tables consist of a format line, followed by rows of

ing writeback to the same cache line, and a register that stores the difference between the number of expected invalidate messages and the number of invalidate acknowledge messages. The WRB has the following fields: a flag to indicate if the writeback was the target of an intervention, and a flag to indicate that there was a read request from the same processor targeting the cache line being written back to memory.

2.2 I/O Protocol

The I/O section of the protocol implements transfers from an I/O device into the memory address space of some program; and also the transfer from memory to I/O devices. The I/O section is interfaced to an I/O cache which it uses for partial writes. Partial I/O writes are performed by first obtaining an exclusive copy and then modifying the exclusive copy in the I/O cache. I/O can also do uncached block and partial reads, and writes. The I/O can be an owner of an exclusive cache line, but can never have a shared copy. There's a coherent request buffer CRB for each outstanding request. The CRB has fields that are a subset of the RRB fields.

As an example, an I/O cache line write *WINV*, that writes a cache line into memory, is shown in Figure 4. Initially the cache line in processor P1, the directory H, and the I/O_1 cluster have a *CEX*, *EXCL* in P1, and *INV* state, respectively. Then H receives a *WINV*, and responds to I/O_1 with a *WINV* completion *WSPEC* message and issues an intervention remove *IRMVE* message to P1; or responds to I/O_1 with a back-off intervention remove *BRMVE* message, that in turn issues the *IRMVE* message. The processor *IRMVE* response, consists of a remove acknowledge *RACK* message to I/O_1, and a cached copy purged (*PURGE*) transfer message to H.

In all the cache coherency protocol consists of 58 different types of coherence messages: 32 of which ensure consistency between processors, and 26 that are used when moving data from I/O devices into coherent space, or coherent space to I/O devices.

3. Design Workflow & SMV Protocol Model

The protocol specification consists of a collection of multiple input, multiple output state machine tables, that determine the response to incoming messages in terms of state changes and outgoing messages. The tables serve as input to a performance simulator, SMV, verilog RTL state machine generation, and to a text processor specification document generator.

An overview of the design workflow is shown in Figure 5. During the early phases of the project, protocol design alternatives were evaluated, for the processor portion of the protocol, using a trace driven performance simulator. This simulator was turned into an effective verification tool by monitoring, that no processor in the model is hung. The

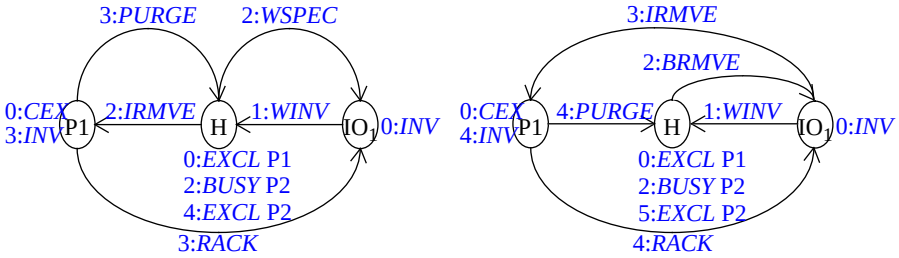

Figure 4: protocol example, *WINV* to an *EXCL* state

exclusive transfer of ownership message (*XFER*), that P2 is the exclusive owner of the cache line. Note that the *ESPEC*, and *EACK* (or *BRDEX* in the back-off case) messages could arrive at P2 in either order. Also the *XFER* might still be in flight when another request arrives at H. Another protocol example, an upgrade (*UPGRD*) to a shared (*SHRD*) state is shown in Figure 3. Initially the cache line in processor P1, the directory H, and processor P2 have a shared *SHD*, *SHRD* in P1, and *INV* state, respectively. Then H receives an *UPGRD* request from P2, responds to P2 with an upgrade acknowledge *UPACK* message, and either issues an invalidate message to P1; or a back-off invalidate *BINV* message to P2, that in turn issues the *INVAL* message. In general the *INVAL* is issued to all clusters that have their bit set in the directory bit-vector. The *UPACK* message contains the number of invalidate acknowledge (*IVACK*) messages that will be received. In the case of back-off, the *BINV* message contains a copy of the bit-vector.

2.1 Directory Cache Scheme

A directory memory entry can be in several states including the following: unowned by any cluster *UOWN*, *SHRD*, *EXCL*, and busy intervening or invalidating (*BUSY*). If the state is *SHRD*, the directory stores a bit-vector with a bit set for clusters that have a *SHD* copy. If instead the state is *EXCL* or *BUSY*, it stores the ID of the owning cluster.

A processor cache-line can have one of four states: *INV*, *SHD*, *CEX*, and dirty exclusive *DEX*. A processor executes instructions speculatively, can have several pending read requests, but the instructions complete in program order. The processor appears to execute cached memory with strong ordering, i.e. it appears to the programmer that the memory operations are executed in exactly the same order as the corresponding instructions were executed in the program. Strong ordering in the processor depends on the implementation of the coherence protocol in the memory system. It is possible to design a memory system that does not return data in strongly ordered fashion, allowing weak consistency models to be employed. In this case the processor uses a fence instruction to control sequencing of memory operations. The fence instruction guarantees that the entire system has completed all previous memory instructions, before any subsequent instructions accessing memory can become visible to the programmer.

The cluster coordinator has a read request buffer RRB entry for each outstanding processor read request, and a write request buffer WRB entry for each outstanding write-back. Each RRB entry has several fields including the following: a field to indicate if the particular read request is the target of an intervention or invalidation request, a flag to indicate if the data has been given to the processor, a flag to indicate that the read request acknowledge has been received, a flag to indicate that the read request targeted a pend-

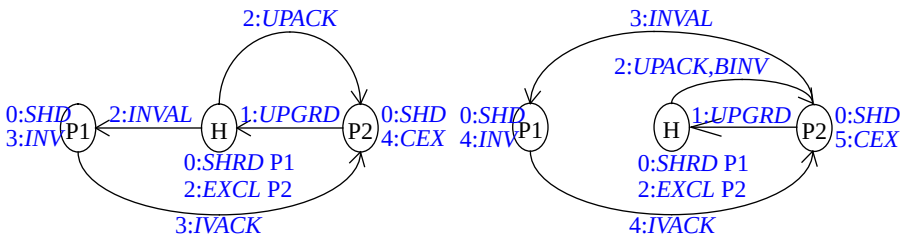

Figure 3: protocol example, *UPGRD* to a *SHRD* state

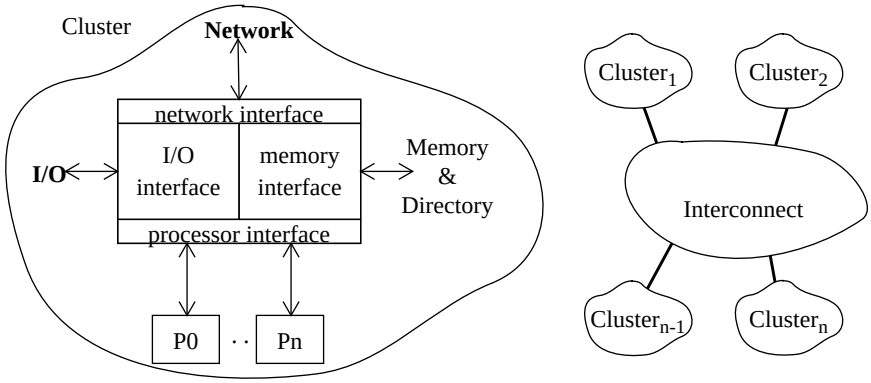

Figure 1: Architecture of a Distributed Shared Memory machine

that at any time, different processor and I/O caches contain coherent values for the same memory location, and that the order of writes to different locations, as seen from the programmer, are consistent with the memory consistency model. The coherence protocol that we verify is invalidation based, and together with the processor [Hei94], supports both the sequentially consistent, and release consistent memory models [Adv93,Gha90,Gha93]. The directory memory stores information about the state of a particular cache line. The protocol is non-blocking, i.e. never buffers requests at the directory memory. If the memory does not have ownership, the directory state is modified to signal that the request was handled and the request is forwarded as an intervention or invalidate to the processor that does have ownership. An invalidate (inv) is forwarded if a processor has a shared copy, otherwise there's an intervention (ivn). In order to be independent of a specific network topology, the protocol does not rely on network ordering.

A protocol example, a read exclusive (*RDEX*) to an exclusive (*EXCL)* state is shown in Figure 2. The numbers before the message, or state, gives the order of receipt. Initially the cache line in processor P1, the directory H, and processor P2 have a clean

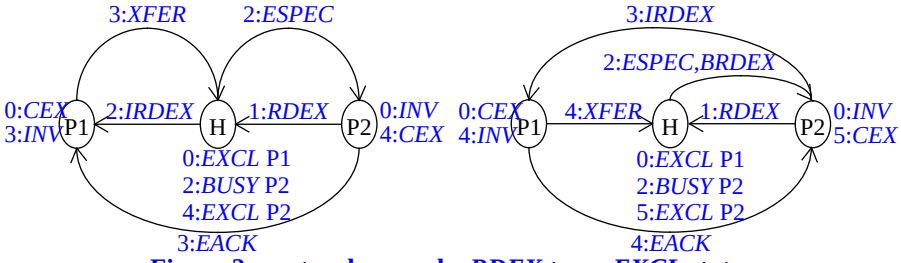

Figure 2: protocol example, *RDEX* to an *EXCL* state

exclusive (*CEX*), *EXCL* in P1, and invalid (*INV*) state, respectively. Then H receives an *RDEX* request from P2, responds to P2 with an exclusive speculative reply *ESPEC*, and either issues an exclusive intervention *IRDEX* to P1, or if H lacks resources, forwards the issuing of the intervention to P2 using a back-off exclusive intervention *BRDEX* message. The *BRDEX* message contains the cluster ID of P1. When the *IRDEX* reaches P1, it replies directly to P2 with an exclusive acknowledge (*EACK*), and informs H, using an

properties. In contrast to formal verification: conventional simulation methodology can be viewed as verifying that a system model conforms to the diagnostic test suite, i.e the specification of the design consists of the diagnostic tests. The inherent problem with this approach is the writing of diagnostics that sufficiently cover all the possible behaviors of the system. The Everest/Challenge system[Gal92], for example, is testimony to the fact that conventional simulation techniques can be very powerful. But it is also clear that parts of that system, i.e. protocols that govern the interaction of the processor, memory, and I/O sub-systems have so many possible cases of interactions, that it is very difficult to verify them with conventional simulation.

We chose SMV [McM93] to formally verify the protocol specification. The strongest reason, from an industry perspective, is that SMV has been successfully used to verify the specifications of other cache coherency protocols [Cla93,Lon93,McM91]. Another reason for choosing SMV, is that it can be easily integrated with the existing design workflow within the company. Finally, the third reason for selecting SMV is that source code is available in case there are problems with the tool.

We did evaluate several other approaches vis-a-vis SMV: the VOSS [Bry91,Seg93] finite state machine trajectory analysis tool, the COSPAN [Kur94] finite automaton ω-language containment tool, and the HOL [Gor88] higher-order logic proof assistant.

The VOSS tool was not chosen because it is designed to verify implementations of interacting state machines, and as such it lacks the proper behavioral and temporal abstraction capabilities that are necessary to verify protocol specifications. The tool does not accept non-deterministic state machines, which are essential when verifying abstract models. Also the tool restricts temporal specifications to the always-in-next-state AX operator, but does not have the CTL eventually operators (e.g always-eventually AF, always-condition1-until-condition2 AU, and exists-eventually EF), that are necessary to verify non-deterministic models.

The COSPAN tool is built on powerful theory that uses a property specific refinement capability to counter computational complexity, and the state explosion problem. We did not pursue the use of COSPAN because we could not find any protocol verification case studies in the literature comparable to the ones cited for SMV.

Finally the HOL approach was not considered because it is manually intensive, and has mostly been successful in reasoning about data paths; exception [Low90]. Whereas the protocol specification is control logic dominated, and contains practically no data paths.

The rest of the paper is organized as follows. We first give a brief description of the protocol, then in turn describe the verification environment, the formal verification methodology, the types of design problems uncovered, and finally describe, as seen from our perspective, the open issues and future research.

2. Distributed Shared Memory Machines

Directory based distributed shared memory machines (DSM) [Len92, Tan95], see Figure 1, consist of clusters of one or more processors, physical memory, directory memory, a cluster controller, I/O devices, and interconnect between the different clusters. A programmer of a DSM class machine is presented with a single, linear, virtual address space that is shared among all the processes running on clusters, that have distributed physical memory. The cache coherency protocol is the set of rules that ensure

Using Formal Verification/Analysis Methods on the Critical Path in System Design: A Case Study

Ásgeir Th. Eiríksson[1] and Ken L. McMillan[2]

[1] Silicon Graphics Inc., Mountain View, CA
asgeir@sgi.com
[2] Cadence Berkeley Labs, Berkeley, CA
mcmillan@cadence.com

Abstract. We present a case study of the use of formal verification methods in a computer system design project. The SMV model checker was integrated into the project design flow, and used to verify a specification of a cache coherency protocol for a distributed shared memory machine. Both the processor and I/O portions of the protocol description were verified, within the strict time schedule of the overall project.
We consider the following to be the three main benefits to using the SMV model checker: it's an effective proof reader of large specifications, which facilitates faster design changes; it allows the verification of the interaction of the processors and I/O early in the design phase; and most importantly it uncovered several protocol specification problems. One problem it uncovered, would never have been found in simulation, and because of its subtle symptoms, loss of coherency, might not have been found on the test floor.

1. Introduction

The paper presents the results of integrating the use of formal verification methods with conventional computer system design methodology. The type of problem seen in the test lab, e.g. state machine deadlock, motivated us to evaluate the use of formal verification methods in the design of computer systems [Gup92,Yoe90]. A problem seen in the test lab typically involves the interaction of many state machines, and is only observable after some unusual chain of events. The size of the state space for these interacting state machines is far too large to make a thorough verification feasible with conventional simulation methods. We also observe that future generations of machines; because of their increased complexity, will exacerbate the verification problem.

The pilot formal verification project involved the verification of a cache coherency protocol in a directory based, distributed shared memory, machine [Len92, Tan95]. The evaluation of available formal verification tools and methodologies, and the design of the machine, was started at the same time. Then the formal verification of the protocol with the chosen tool, once the protocol specification was stable. To be considered successful, the pilot project had to demonstrate a quantum leap increased verification value due to formal verification methods, within the strict time schedule of the project. Specifically, the chosen tool and methodology had to find problems in the protocol specification, before RTL coding commenced.

A system model is formally verified by showing, with mathematical techniques, that it conforms to the specified properties. As an example, the properties we might wish to verify for a protocol specification are the absence of deadlock, and that a processor request, always receives the expected response. Formal verification amounts to exhaustively, for all possible cases, verifying that a particular model satisfies the specified