# Dynamic Verification of Cache Coherence

Jason F. Cantin, Mikko H. Lipasti, James E. Smith

Dept. of Electrical and Computer Engineering
University of Wisconsin-Madision
Madison, WI 53706

{ jcantin, lipasti, jes }@ece.wisc.edu

**Abstract**

Proposed is a method for improving the fault tolerance of cache coherent multiprocessors. By dynamically verifying coherence operations in hardware, errors caused by manufacturing faults, soft errors, and design mistakes can be detected. Analogous to the DIVA concept for single-processor systems, a simple version of the protocol functions as a checker for the aggressive implementation. An example implementation is shown, and the overhead is estimated for a small SMP system. This is work in progress, and detailed simulation results are not yet available.

## 1 Introduction

Cache coherence protocols are notoriously difficult to design and verify [23]. Though a protocol description may specify only a few states (e.g., MOESI), implementations quickly become very complicated as states are added to handle the non-atomicity of memory operations, preserve correctness, and implement protocol optimizations [30]. The complexity increases the possibility of subtle errors in the specification and/or low-level implementation. Furthermore, transient failures, caused by non-ideal operating environments and alpha particles interacting with very small devices are likely to pose major reliability problems [2]. Thus, tolerance against design errors and transient faults will be important for ensuring the reliability and scalability of cache coherent multiprocessor systems.

Recently, Rotenberg observed that the result of a complex computation may be checked for correctness more efficiently than it was first computed provided the check is delayed in time [4]. Austin proposed a novel approach for runtime verification of complex superscalar processors based on this principle [3]. Because the verification hardware is simple and centralized, its correctness can be easily verified. This verifies every computation dynamically using a form of complete induction. We refer to this process as dynamic verification (DV).

We propose using DV techniques to improve the fault tolerance of cache coherent multiprocessor systems. However, a centralized check processor approach as used for single processor systems exhibiting serial semantics [3] is probably inappropriate for distributed cache coherence hardware based on parallel multiprocessor semantics. Consequently, we propose a distributed version of DV for concurrently checking cache coherence protocols during execution. As an example, we demonstrate this concept with a symmetric multiprocessor system. In this paper, we concentrate on the error detection mechanisms. We leave the integration of DV with recovery techniques for future work.

### 1.1 Dynamic Verification

As mentioned above, a complex computation can be checked for correctness more efficiently than it can be computed in the first place, provided the check is delayed in time. The key is that a form of complete induction can be used to perform the check, yielding simple control and a high level of parallelism in the checking process. This process is illustrated in Figure 1 below.

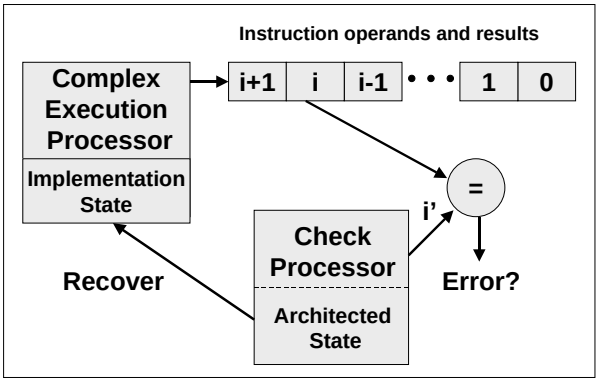


Figure 1: Dynamic Inductive Verification of a Processor Core

The primary execution processor (e.g. a superscalar implementation) ultimately produces a sequence of state changes <PC, reg, data> or <PC, mem address, data> that capture the semantics of the computation. As proposed by Austin, this sequence is held in the reorder buffer and can be passed to a check processor after any speculation has been resolved [3].

The check processor lags behind and re-executes the program. However, because of the time lag, any check processor instruction needing the result of a previous instruction can use the corresponding result produced

by the execution processor. The check processor never has to stall for values that it produces itself. After the check processor produces a result (including the next PC), it is compared with the corresponding value produced by the execution processor. In effect, instructions 0,1,...,i-1 produced by the execution processor are assumed to be correct by the check processor. The check processor uses this information to generate the result of the next instruction, i', and compares it with the result of instruction i produced by the execution processor. If the instructions 0,1,..., i-1 are correct, then instruction i is also correct, provided its result matches i'. Hence, the check process verifies the full execution by complete induction.

In [3] hardware mechanisms are proposed to allow the primary execution processor to recover correct state and continue execution after an error occurs. Referring again to Figure 1, a copy of architected state maintained by the check processor can be used to restore and restart the execution processor after an error is detected. If instruction i and i' differ, the correct check processor value i' is used, and execution resumes with instruction i+1. Consequently, forward progress is made, despite the detected error. The check processor is assumed to be correct by virtue of its simpler, easy to verify design, and the use of fault avoidance in its implementation [3].

As articulated in [3], the benefits of the DIVA approach are the following.

- It detects hardware faults; assuming faults in the execution processor and the checker are not correlated.
- It detects design errors in the execution processor, assuming that the checker is correct. The check processor is very simple, so verification should be straightforward.
- It can recover from faults, provided the check processor maintains architected state and makes at least one instruction of forward progress when a failure is detected.
- The level of design verification for the execution core can be relaxed because the checker can be depended upon for both detection and recovery of design errors. This could allow a microprocessor to ship sooner, potentially containing elusive design errors in the execution processor.
- Going further, the technique may be used to enhance performance by intentionally letting the checker take care of rare corner cases. This simplifies the design of the execution hardware and potentially allows it to go faster for common cases.

### 1.2 Cache Coherence

Modern multiprocessor systems are typically constructed from commodity processors with on-chip caches or cache hierarchies. Despite the replication of data in caches, it must appear to the programmer that there is one coherent memory. Cache-coherence protocols are used to efficiently maintain this illusion.

Figure 2 shows a simple example of a coherence protocol, MSI, where the states for a cache line are "Modified", "Shared", and "Invalid". When the data is not present in the cache, its state is *Invalid* (I). When a read-only copy of the line is present, the state is *Shared* (S), indicating that copies may exist elsewhere. When a single, modifiable copy is present the state is *Modified* (M), indicating that this is the most up-to-date copy of the data.

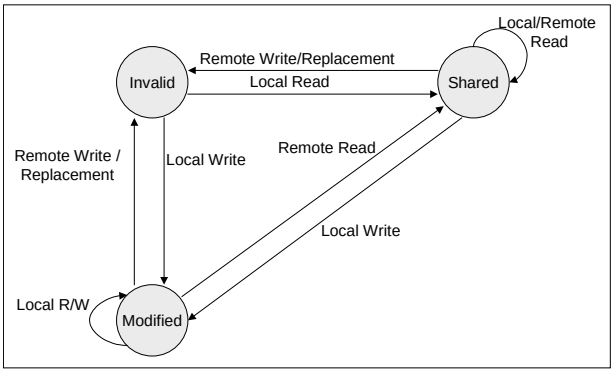


Figure 2: State Diagram of MSI Protocol (Adapted from version shown in [30])

Note that not all combinations of cache states are allowed at once. For example, two processors with a modifiable copy of a cache line may erroneous state combinations shown in Fig. 3.

| Processor A State | Processor B State | | |
|---|---|---|---|
| | I | S | M |
| I | Ok | Ok | Ok |
| S | Ok | Ok | Error |
| M | Ok | Error | Error |

Figure 3: Allowed State Combinations for MSI

This is a very simple cache coherence protocol by current standards. Real implementations require many additional states to handle non-atomic memory operations and speculation [30]. For example, just accounting for pending write-backs complicates implementation of the protocol considerably (Figure 4).

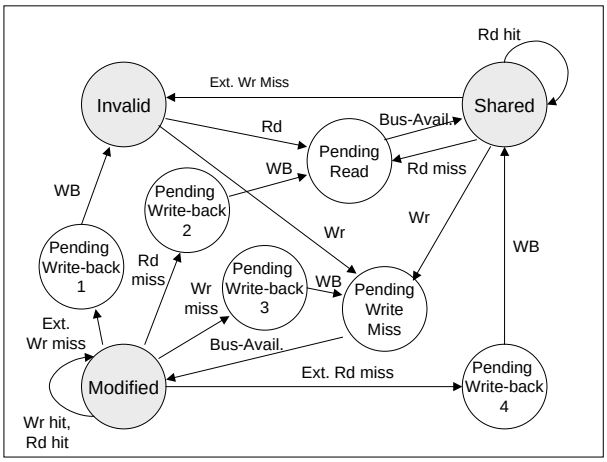


Figure 4: MSI Protocol with Transient States (Adapted from [30])

Before proceeding further, we define some useful terminology. We refer to the states used in Figure 3, as *stable* states. These states are defined in high-level descriptions of the protocol, and used to reason about interactions between processors and memory. Given atomic memory operations, the *stable* states are sufficient to correctly realize the protocol. The additional states added to implement the protocol with real hardware are referred to as *transient* states, following the convention in [30]. We refer to the state machine composed of stable states as the *simple protocol*, and the combined state machine (stable and transient states) as the *implementation protocol*. Finally, the current state of the implementation protocol is the *implementation state*, and the current state in the simple protocol is referred to as the *architected state*.

## 2 Dynamic Verification of Cache Coherence

DV can be used for cache coherence. Unlike the centralized DIVA checker paradigm, a mechanism for dynamically verifying cache coherence should be logically distributed. Figure 5 shows a conceptual view. In the uniprocessor case, complex hardware does the computation initially, using some combination of implementation state (ROB, prediction tables, etc.) and architected state. The check processor maintains only architected state, and verifies the computations. For cache coherence, the implementation protocol initially computes the state, sends requests, and services external requests. Completed transitions between stable states are passed in program order to checker circuits that verify the state based on the simple protocol. The checker circuits maintain the architected state, and communicate with each other via check messages to ensure coherence between nodes.

DV can detect subtle implementation mistakes, manufacturing faults, and transient faults in the control circuitry. Furthermore, if the checker is co-designed with the implementation protocol, it can be used to flush out specification errors early in the design phase.

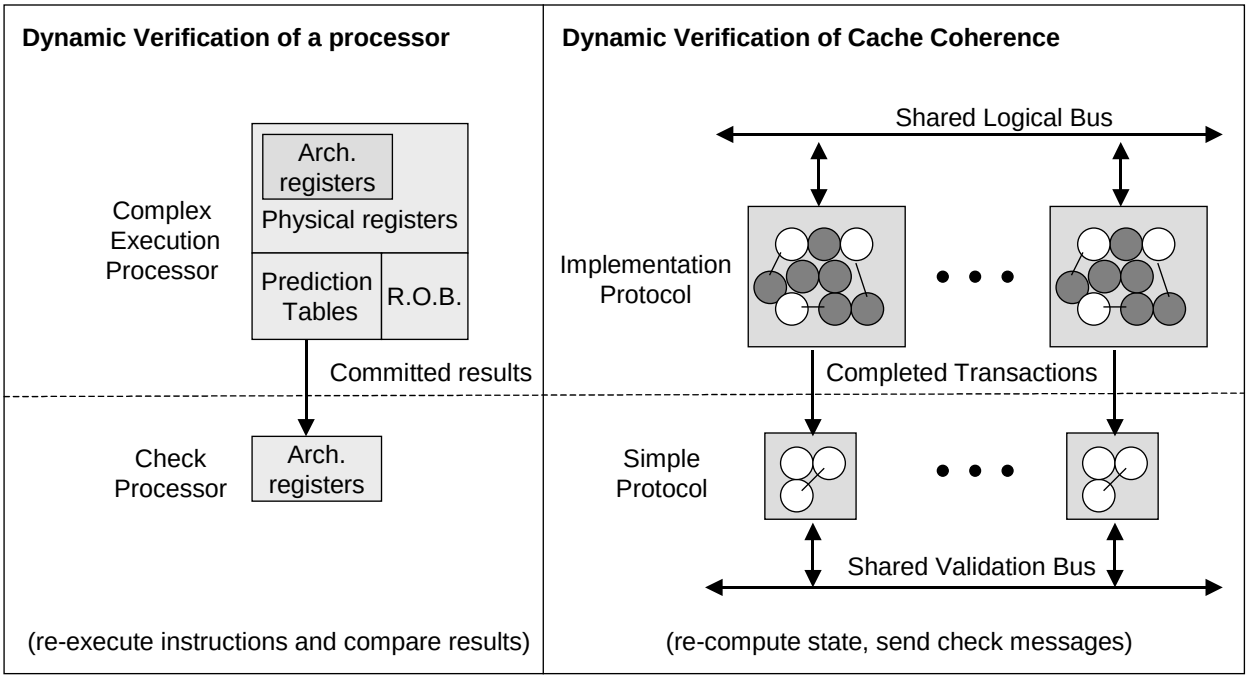


Figure 5: Conceptual View of Dynamic Verification for Cache Coherence

With this approach, the checker is implemented in a software model as part of the design verification effort, and used to check a model of the implementation protocol in simulations.

### 2.1 Symmetric Multiprocessor (SMP) Example

To incorporate dynamic verification of coherence into a multiprocessor, a special checker circuit is placed in each node to verify the protocol actions locally (See Figure 6). The checker implements a simplified version of the coherence protocol logic (i.e., no transient states), and maintains its own copy of the tags. A watchdog timer is included to check for omission faults [4].

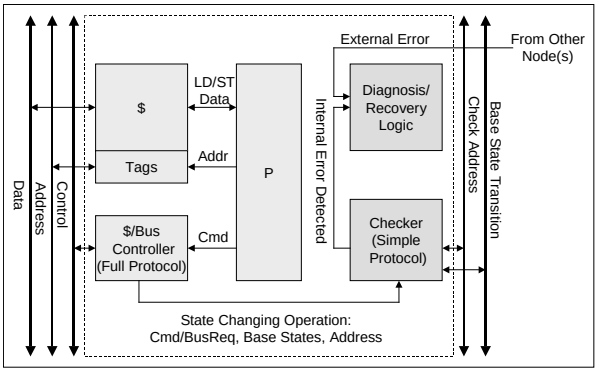


Figure 6: Processing Node with Additional Hardware for Dynamic Verification

A second logical network is used to check the protocol globally. See Figure 7 for an example based on shared buses. We call this network the *validation network* to distinguish it from the main interconnection network. This network is used to broadcast final states that have to check for illegal combinations of states between nodes (e.g., two caches with exclusive copies). For the SMP case, this is just a second bus for addresses and final states.

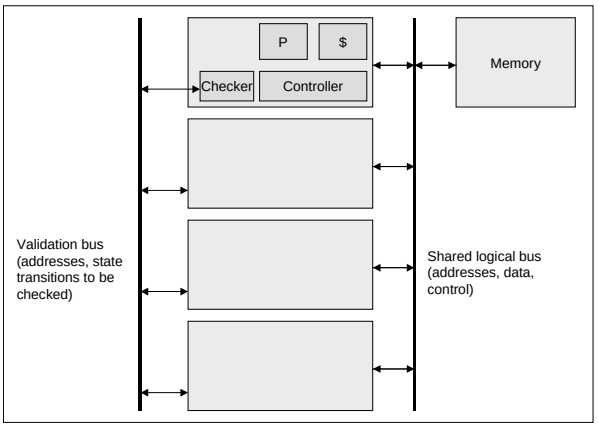


Figure 7: Bus-based Symmetric Multiprocessor with Support for Dynamic Verification

### 2.2 SMP Coherence Checker Operation

Following a network transaction in the implementation protocol, the address, command, initial and final stable states are sent to the local checkers. Each checker re-computes the final state of the cache line and compares it to the implementation protocol's result. If the final states do not match, an error has occurred in the implementation.

The checker also performs a tag-lookup with the address to get the architected state of the cache line. The architected state stored in the checker must match the initial state reported by the implementation protocol. Disagreement signals an error. See Figure 8 for a simplified checker datapath.

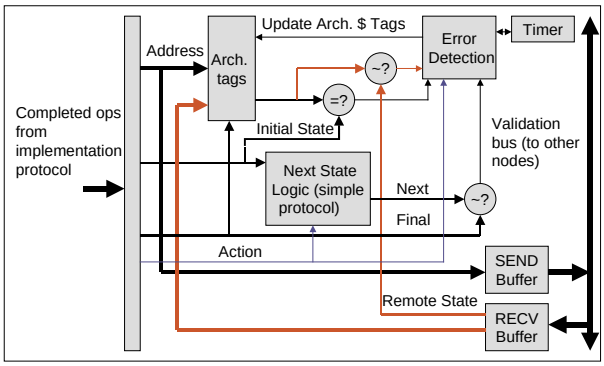


Figure 8: Coherence Checker Logic (Simplified).

Once the transaction has been verified locally, the cache states must be checked globally. The node that initiated the transaction (via a request) broadcasts the final state and address of the cache block over the validation network. The other nodes snoop the network and determine if the broadcasted cache state conflicts with the state of a cached copy they hold. For example, if a node acquires a modifiable copy of a memory block, it sends a check message indicating that it has the block in "M" state. The other checkers must determine if they have any shared or modified copies still in their caches. If an illegal combination of cache states is detected for an address (Figure 3), an error is signaled.

Once the node that provided data for the transaction (the receiver) sees the check message, its checker knows that the transaction has completed. The transaction may then be retired and removed from any queues. Note: we do not allow further updates to the architected state of the receiving node until a corresponding check message is received. Depending on whether or not recoverability is desired, the initiator can retire the transaction after sending the check, or wait to make sure that no errors are signaled.

## 3 Evaluation of Coherence Checker Implementation

Design of the coherence checker is in progress, however we can reason about its effectiveness and performance. Ideally, a checker implementation should have full fault coverage. By full coverage, we mean complete detection of faults that have propagated to the point of being visible to the coherence checker (e.g., a stable state transition that violates coherence is made by the implementation protocol). For example, a design error may cause the omission of an invalidate message in the implementation protocol, however this will not be detected by our scheme until it results in an improper stable state transition.

In addition, the coherence checking hardware should not slow down the system by introducing too much overhead, signaling too many false positives, or lagging too far behind the implementation for efficient recovery.

### 3.1 Coherence Checker Coverage and Specificity

Symbolic model checking is a powerful technique for verifying finite state machines and protocols [21, 22, 23]. It has been successfully used to verify cache coherence protocols [21, 22]. Given a model of a system and a set of logical properties, a program can determine if the modeled system satisfies the properties in all cases. We can use symbolic model checking techniques to verify the correctness of the simple protocol, determine the checker implementation's coverage, and determine if the checker implementation ever incorrectly signals an error (a false positive).

Determining if the simple protocol is correct is straightforward. A model of the simplified state machine can be written in a language such as SMV (top part of Figure 9). A set of necessary conditions for maintaining coherence is then specified formally in CTL. The model checking software determines if the model always meets the conditions, or produces a counter-example. Without the complexity of the full implementation, the state space will be relatively small and easily searched by the model checking software.

To determine if the coherence checker implementation detects errors, we write a detailed model for the checker implementation (middle part of Figure 9). Next, we formally define (in CTL) what errors the coherence checker implementation should detect. The checker implementation achieves full coverage if, for every defined error condition, the coherence checker signals an error.

To determine specificity, we can also use model checking (bottom part of Figure 9). We can combine the two models mentioned above such that the simple protocol is checked by the implementation of the coherence checker. Since the simple protocol has been proven correct, the coherence checker implementation should never detect an error in this configuration. Any errors signaled by the coherence checker in this configuration are considered false positives.

This is analogous to proving the correctness of a DIVA checker in the single processor case, however the burden of defining the necessary conditions for correctness is placed on the designer. This is true in general for verifying concurrent systems with model checking (a design may still be incorrect, but satisfy the designers specifications). Further, model checking can only tell us if the checker implementation will detect errors specified by the designer, such as disagreement between the implementation protocol and the simple protocol. Finally, the definition of false-positives relies on the correctness of the simple protocol.

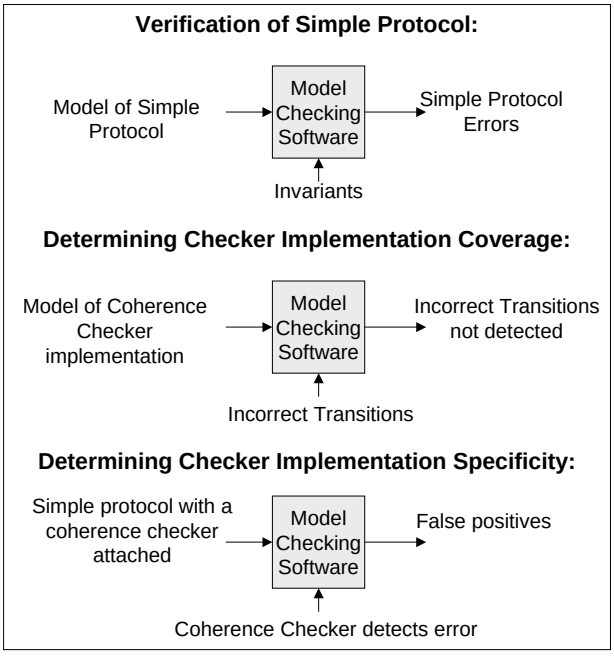


Figure 9: Determining Checker Coverage and Specificity

### 3.2 Coherence Checker Overhead

Conceptually, the coherence checker implementation proposed should not become a bottleneck for the base system, or increase system cost excessively. The absence of transient states keeps the checker logic simple and fast. Further, the second bus proposed for the SMP configuration produces no more transactions than the main address bus, so duplicating the address

portion of the bus may be sufficient to support the extra messages in that case. However, if a second physical network is infeasible, the main network may be used to send extra messages (with low priority). If this is the case, bandwidth overhead is incurred by the extra messages used for DV.

To estimate overhead, we collected data from a 4-processor SMP system with 1MB 4-way set associative caches and 64-byte lines. For simplicity, an MSI protocol is used, though the results do not change significantly for a MOESI protocol. From this data, we can determine (relatively) how often the checker must verify a coherence operation. Figure 10 shows (for five benchmarks) that between 0.54% and 7.17% of memory references (loads and stores) result in a change of architected state for a cache line. We infer from this that the checking of local state transitions is infrequent and need not be fast.

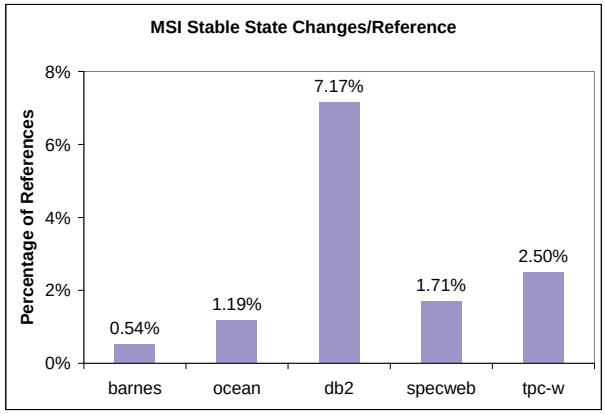


Figure 10: Percentage of Memory References that Result in Stable State Transitions

The estimated overhead incurred by extra messages is calculated for several strategies below (See Figure 11). For each of these strategies, the transactions they check is shown in Figure 12.

The first approach is to have the node's checker send a message each time that a modifiable copy of a cache line is loaded. This ensures that no other shared or modified copies exist. This simple approach requires 31% or fewer extra messages for the benchmarks simulated, but cannot detect certain types of errors. Cache lines brought into a node in the shared state are not checked to see if modified copies still exist, and replacements are not checked.

A second approach is to have check messages sent for all transactions resulting from a cache miss. This checks that data is brought into a node in the correct state, but does not check upgrades (S->M). This approach requires 45% or fewer additional messages.

Third, check messages may be issued for all bus read or upgrade operations. This approach is a combination of the first two, and has improved coverage. Each time a block is brought into a cache or upgraded, a check message is sent by the initiator of the transaction. However, this is even more costly than the previous methods (as much as 64%).

Finally, a check message may be sent for all bus transactions. In addition to all cases checked by the third approach, writebacks resulting from replacements are also checked. In effect, the second bus proposed for checking purposes mirrors the main bus, ensuring that all transactions were completed correctly. This incurs 100% address bandwidth overhead (same overhead as replicating the main address bus, but with the ability to catch design errors).

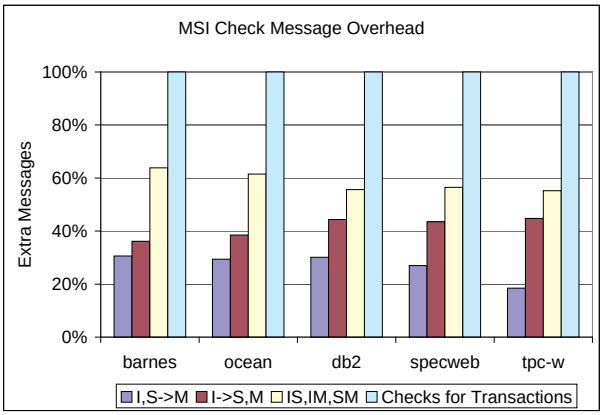


Figure 11: Calculated Message Overhead for Checker Implementation Strategies

| Initiator Trans. | Remote Node State | | | Bus Trans. | Check Message Coverage | | |
|---|---|---|---|---|---|---|---|
| | I | S | M | | Scheme 1 | Scheme 2 | Scheme 3 |
| I->S | Ok | Ok | Error | Yes | Checked | | Checked |
| I->M | Ok | Error | Error | Yes | Checked | Checked | Checked |
| S->M | Ok | Error | -- | Yes | | Checked | Checked |
| S->I (Inv) | Ok | Ok | -- | Yes | Checked | Checked | Checked |
| S->I (Repl) | Ok | Ok | -- | No | | | |
| M->I (Inv) | Ok | -- | -- | Yes | Checked | Checked | Checked |
| M->I (Repl) | Ok | -- | -- | Yes | | | Checked |
| M->S | Ok | -- | -- | Yes | Checked | | Checked |

Figure 12: Coverage Provided By Check Messages

None of the strategies mentioned checks the transition from *Shared* to *Invalid* that occurs when a cache line is replaced. This is a silent transition that does not involve updating memory, or sending data to other nodes. The data is discarded, and we rely on the checker internal to the node to make sure that the state transition takes place.

## 4 Related Work

Testing-based approaches are conventionally used for detecting both design errors and fabrication defects [6]. For design verification, the most commonly used technique is to develop a set of test vectors and use them to drive logic simulation [7, 8, 9]. The designer or a verification engineer may devise such test vectors. Parts of this process may be automated, but its effectiveness depends on the insight and skill of the engineers involved. This method is extremely time-consuming, and typically missed some bugs.

For hardware failures, test patterns are generated to exercise the hardware, often using a fault model such as the stuck-at model [6]. The process can be largely automated [10, 11], and test coverage can be quantitatively estimated. The test patterns are then applied to verify correctness -- up to the level of test coverage. In the field, this technique works better for permanent faults than for transient ones, as the fault must be present at the time the test is applied. Also, this method may require some downtime when the test is applied.

Rotenberg proposed a multithreaded processor that implements a form of time redundancy where a computation thread is re-executed later in time and the results of the two thread executions are compared [4]. His approach focused on transient hardware faults in the multithreaded processor's datapath, and also built on previous approaches using time-shifted redundant execution [12, 13]. He referred to this new technique as Active-stream / Redundant-stream Simultaneous Multithreading (AR-SMT).

Reinhardt and Mukherjee further explored the use of multithreading for transient fault detection in [5]. They introduced an important abstraction for simultaneous and redundantly multithreaded (SRT) processors, identified some key implementation challenges, and suggested some microarchitectural solutions.

As described earlier, Austin proposed dynamic checking with a separate check processor for the second computation [3].
Conventional forms of dynamic checking have been proposed and implemented for many years. Probably the oldest is replication with comparison checking as protection against hardware failures [14, 15, 16]. This method can be effective against both permanent and transient hardware errors, but it does not catch design errors. Furthermore, it is likely to be more expensive than the dynamic inductive checking method, because the check processor is a complete replica and is not simplified. Many systems have used replication for failure protection. The IBM G5 [17] is a recent version where both processors are on the same chip.

For detecting design errors, formal methods [19, 20] provide an alternative to conventional simulation-based testing. Formal methods typically use an architecture specification and an implementation specification, and then show the two are equivalent. This equivalence is essentially proven for all possible computations, either via model checking [21, 22, 23], theorem proving [24], or a combination [25, 26]. As high-performance implementations of coherence protocols become more complex, the computational complexity of formal methods becomes an issue.

## 5 Future Work

In future work, we will refine our implementation with data obtained from model checking and detailed timing simulations. Detailed simulations will determine the actual overhead of the check processor implementation for commercial workloads.

Though a bus-based SMP system with a simple protocol was used here to illustrate the concepts, more complex protocols (such as MOESI) and scalable directory-based coherence schemes will be explored. We intend to develop a framework for checker design, verification, and performance evaluation to facilitate the process of incorporating DV into parallel systems.

Finally, we intend to combine DV with hardware and software recovery techniques. Once an error has been detected and diagnosed, it may be possible to restart from a checkpoint or use some form of forward error recovery.

## 6 Conclusions

A method for improving the fault tolerance of cache coherent shared memory multiprocessors has been proposed. This method is focused on the cache coherence hardware, and is motivated by the observation that a complex computation can be checked more easily than it was originally computed [4]. With dynamic verification, errors caused by manufacturing faults, soft errors, and design mistakes can be detected. As an example, this concept was illustrated in more detail for a symmetric shared-memory multiprocessor with an MSI protocol. Though preliminary, some estimates were made as to the overhead incurred by the check process.

## 6 Acknowledgements

We thank Ashutosh Dhodapkar, Timothy Heil, Tejas Karkhanis, and Sebastien Nussbaum for their comments on drafts of this paper. This work is supported in part by NSF grant CCR-0083126 and by IBM. Jason Cantin is supported by a Wisconsin Distinguished Graduate Fellowship provided by Peter Schneider.

## 7 References

[1] SIA Roadmap. Semiconductor Industry Association, 1999.

[2] T. May and M. Woods. Alpha-Particle-Induced Soft Errors in Dynamic Memories. *IEEE Transactions on Electronic Devices*, 26(2), 1979.

[3] T. Austin. Diva: A Reliable Substrate for Deep-Submicron Processor Design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, Los Alamitos, November 30-December 2 1999. IEEE Computer Society.

[4] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *Proceedings of the 29th Int. Symposium on Fault Tolerant Computing*, pages 84-91, June 1999.

[5] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *Proceedings of The 27th International Symposium on Computer Architecture*, pages 25-36, June 2000

[6] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, New York, 1992.

[7] T. Sasaki, A. Yamada, and T. Aoyama. Hierarchical Design Verification for Large Digital Systems. In Proc. 18th *Design Automation Conference*, pages 105-112, June 1981.

[8] M. Monachino. Design Verification System for Large-scale LSI Designs. In *Proc. 19th Design Automation Conference*, pages 83--90, June 1982.

[9] A. Aharon. Test Program Generation for Functional Verification of PowerPC Processors in IBM. In *Proc. 32nd Design Automation Conference*, pages 279-285, June 1995.

[10] J. P. Roth, W. G. Bouricius, and P. R. Schneider. Programmed Algorithms to Compute Tests and Detect and Distinguish Between Failures in Logic Circuits. *IEEE Transactions on Electronic Computers*, EC-16(10):567-579, October 1967.

[11] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, C-30(3):215-222, March 1981.

[12] J. Patel and L. Fung. Concurrent Error Detection in ALUs by Recomputing with Shifted Operands. *IEEE Transactions on Computers*, C-31(7):589-595, July 1982.

[13] G. Sohi, M. Franklin, and K. Saluja. A Study of Time-Redundant Fault Tolerance Techniques in High-Performance Pipelined Computers. In *Proceedings of 19th Fault-Tolerant Computing Symp*., pages 436-443, June 1989.

[14] S. Webber. The Stratus Architecture. In Siewiorek and Swarz, editors, *Reliable Computer Systems: Design and Evaluation*. Digital Press, Bedford, MA, 1992.

[15] O. Serlin. Fault-Tolerant Systems in Commercial Applications. *IEEE Computer*, pages 19-30, Aug. 1984.

[16] D. Johnson. The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems. *IEEE Computer*, pages 40-48, Aug. 1984.

[17] L. Spainhower and T. A. Gregg. IBM s/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6):863, 1999.

[18] S. Mitra, N. Saxena, and E. McCluskey. A Design Diversity Metric and Reliability Analysis for Redundant Systems. *Proceedings of the 1999 International Test Conference*, pages 662-671, Jan. 1999.

[19] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626-643, Dec. 1996.

[20] E. Clarke and R. Kurshian. Computer-aided Verification. *IEEE Spectrum*, 33(6):61-67, 1996.

[21] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and A. L. Ness. Verification of the Futurebus+ Cache Coherence protocol. In *Proceedings CHDL*, 1993.

[22] E. Clarke, O. Grumberg, and D. Peled, Model Checking. MIT Press, Cambridge, MA. 1999.

[23] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. *International Conference on Computer Design, VLSI in Computers and Processors*, pages 522-525, Los Alamitos, Ca., USA, October 1992. IEEE Computer Society Press.

[24] A. Kuehlman, A. Srinivasan, and D. LaPotin. Verify-A Formal Program for Custom CMOS Circuits. *IBM Journal of Research and Development*, 39(1/2):149-165, 1995.

[25] R. Kurshan and L. Lamport. Verification of a Multiplier, 64 Bits and Beyond. *Proceedings of the 5th International Conference on Computer--Aided Verification*, Lecture notes in Computer Science, pages 166-179. Springer Verlag, 1993.

[26] S. Rajan, N. Shankar, and M. Srivas. An Integration of Model Checking with Automated Proof Checking. In *Proceedings of the 7th International Conference on Computer-Aided Verification*, pages 135-146, 1995.

[27] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of Correctness of a Processor with Reorder Buffer Using the Completion Functions Approach. In *Proceedings of the 11th International Conference on Computer-Aided Verification*, volume 1633 of *Lecture notes on Computer Science*, pages 135-146. Springer Verlag, July 1999.

[28] J. Swada and W. Hunt. Processor Verification with Precise Exceptions and Speculative Execution. In *Proceedings of the 10th International Conference on Computer-Aided Verification*, volume 1427 of Lecture notes on Computer Science, pages 135-146. Springer Verlag, 1998.

[29] S. Park and D. L. Dill. Protocol Verification by Aggregation of Distributed Transactions. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 300-310, New Brunswick, NJ, USA, Jul/Aug 1996. Springer Verlag.

[30] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufmann Publishers Inc., San Francisco, California. 1996. Pages 665, E3.