Appears in the Proceedings of the International Conference on Dependable Systems and Networks
San Francisco, CA, June 22-25, 2003

Dynamic Verification of End-to-End Multiprocessor Invariants

Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University

Mark D. Hill, David A. Wood
Computer Sciences Department
University of Wisconsin—Madison

Abstract

As implementations of shared memory multiprocessors become more complicated, hardware faults will increasingly cause errors that are difficult or impossible to detect with low-level, localized mechanisms. In this paper, we argue for dynamic verification (i.e., on-the-fly checking) of end-to-end, system-wide invariants in shared memory multiprocessors. We develop two invariant checkers based on distributed signature analysis. Our coherence-level checker dynamically verifies that every cache coherence upgrade has a corresponding downgrade elsewhere in the system. Our message-level checker verifies that all nodes in an SMP observe the same total order of broadcast requests. We use full-system simulation to show that the checkers detect the targeted errors while not significantly degrading system performance.

1 Introduction

Symmetric multiprocessors (SMPs) are an important class of computers that is getting more complex in the quest for higher performance. Traditionally, an SMP consisted of multiple nodes—which each contain a processor, cache(s), and a portion of the shared memory—connected by a single shared-wire bus. The bus served as a single ordering point that facilitated the total order of cache coherence requests required by the broadcast snooping coherence protocol. Current SMPs, in order to provide the same total order of broadcast requests but with greater bandwidth, may use multiple interleaved logical buses implemented as bit-sliced pipelined broadcast trees with point-to-point links [6], such as the system illustrated in Figure 1. While snooping cache coherence transactions in traditional SMPs were atomic, now even SMPs with shared-wire buses have split transactions [14].

In addition to high performance, SMP designers also seek high availability, in part, because SMPs are often used to run commercial applications, such as database management systems and web servers. Unwilling to sacrifice performance, SMPs have only used fast, localized mechanisms to detect errors caused by physical faults. Memory, caches, buses, and links have been protected by parity or error correcting codes (ECC). SMPs crashed to avoid data corruption on some detected errors and used localized recovery for others (e.g., bus or link re-transmission).

Localized mechanisms for error detection and recovery may not be sufficient for future SMPs as (1) vendors seek to promote high availability (e.g., "five nines" or available 99.999% of the time), (2) designers add more complexity to obtain even greater performance, and (3) deep submicron fabrication becomes increasingly susceptible to faults.

Consider a modern SMP in the following scenario. Processor P1 broadcasts a RequestForExclusive cache coherence request for memory block B. Processor P2 has B in the Shared (i.e., ReadOnly) state in its cache. P1 receives data from memory, while P2 should be invalidated. If a fault causes the loss of P1's request as it passes from the interconnect into P2's network interface, P2 will retain B in Shared. Since broadcast snooping protocols do not use acknowledgments, P1 will not learn that P2 did not receive its request. If P2 continues to read block B, the system could violate its memory consistency model and thus violate correctness.

An approach to addressing such complex scenarios is to adapt Saltzer's *end-to-end argument* [13]. That argument observes that certain functions can only be implemented at higher levels (although lower levels can help). For example, the integrity of file transfer should be checked with end-to-end checksums. Link-level mechanisms can detect some errors, aid fault diagnosis, and help performance, but data may be corrupted elsewhere (e.g., in router memory) or via fault models not anticipated during design.

This paper seeks end-to-end error detection for SMP coherence protocols and interconnects via *dynamic verification* of system-wide, end-to-end invariants. Dynamic verification is the process of the hardware checking its execution on the fly. We implement our invariant checkers with *distributed signature analysis*. For each checker, each cache and memory controller in the system maintains a local signature that it updates for each event (e.g., incoming cache coherence request). Periodically, the system performs a global reduction on all of these local signatures that determines whether the invariant holds.

As one concrete example, we develop a cache coherence-level checker that ensures that all coherence upgrades (i.e., *increases* in access permissions to a memory block) have corresponding downgrades (i.e., *decreases* in permissions)

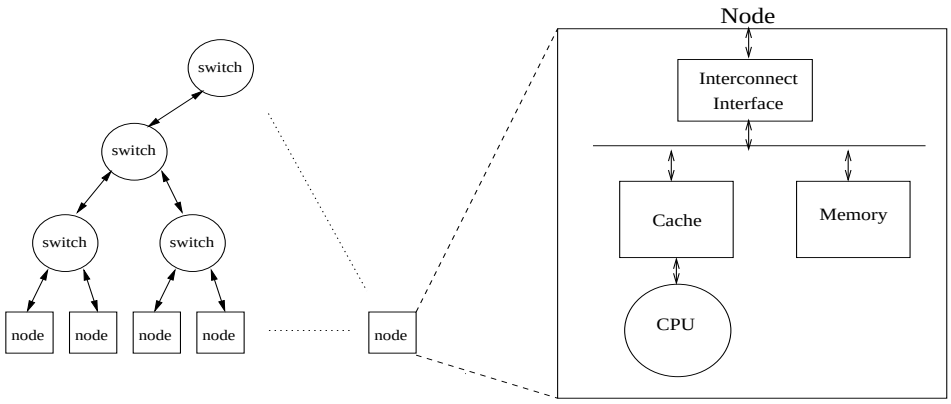

FIGURE 1. Example of modern SMP with broadcast tree interconnect

elsewhere in the system. Simplifying the details for now, each cache and memory controller adds or subtracts the address of the incoming broadcast request to its local signature, depending if the request triggers a coherence upgrade or downgrade at the controller. Periodically, components send their signatures to a centralized controller that checks to ensure that the global sum of all signatures equals zero. This checker would detect the previous error in which P2 did not receive a broadcast request that would have downgraded it.

Of course, end-to-end error detection in SMPs is more valuable if one can recover from errors instead of just crashing. Recovery can be achieved with a *backward error recovery (BER)* mechanism that maintains a safe recovery point that can be restored when a signature check fails. To be acceptable for modern SMPs, however, the BER mechanism should add only modest hardware and performance overheads, despite having to provide a recovery point that is hundreds of cycles behind the active execution. Distributed signature analysis takes that long to verify execution, since even just sending a signature to the system controller takes more than 100 cycles today (and more in the future).

While many BER schemes exist and could be used in conjunction with dynamic verification, we choose SafetyNet [16], a recently developed all-hardware BER mechanism that is well-suited to enabling recovery after end-to-end error detection. SafetyNet (reviewed in Section 3) tolerates long error detection latency by allowing execution to continue even as error detection (e.g., distributed signature checking) is done is the background. This approach hides error detection latency in the common case of error-free execution.

In the rest of this paper, we first explain further why the complexity of modern SMPs motivates end-to-end error detection (Section 2) and review efficient hardware checkpoint/recovery with SafetyNet (Section 3), before making three contributions to improve multiprocessor availability.

- We show, in general, how to apply distributed signature analysis to dynamically verify system-wide invariants in SMPs (Section 4).
- We develop a signature analysis scheme for verifying that every cache coherence upgrade has corresponding downgrades elsewhere in the SMP (Section 5).
- We develop a signature analysis scheme for verifying that all SMP nodes receive the same broadcast coherence requests in the same order (Section 6).

In Section 7, we evaluate our invariant checkers with full-system simulation and commercial workloads. In Section 8, we discuss related research before concluding in Section 9. This work concisely presents work developed in Chapter 4 of Sorin's Ph.D. thesis [15].

2 Motivation for End-To-End Invariant Checking in Modern SMPs

A modern SMP, such as the one we discuss in this paper, consists of a large number of interacting finite state machines (FSMs). A fault in any of these FSMs or in the communication between them can manifest itself as an error. A modern SMP has a complex cache coherence protocol (Section 2.1) and interconnection network (Section 2.2), and this complexity exacerbates the problem of detecting when an error occurs. To address the problem of error detection in such a complicated system, we propose the use of distributed signature analysis schemes to dynamically verify end-to-end system-wide invariants. These system-wide invariants (Section 2.3) are high-level properties of the system that are independent of low-level implementation details.

2.1 Cache Coherence Protocol

We base the memory system design on a hardware-only, broadcast snooping cache coherence protocol. The protocol has four stable states for blocks of memory [7]: Modified (M), Owned (O), Shared (S), and Invalid (I). A processor can broadcast a request on the interconnection network to

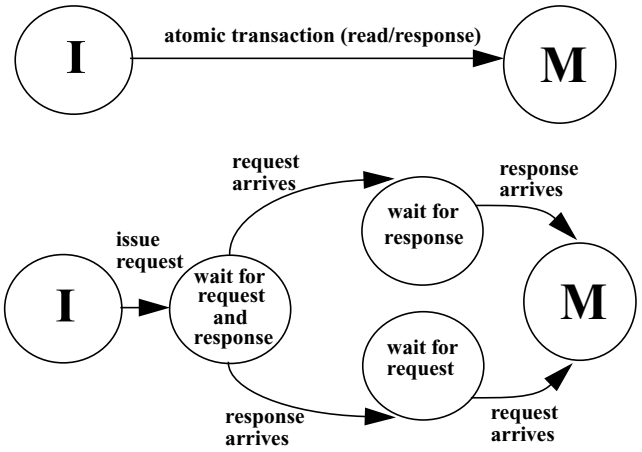

FIGURE 2. Comparing an atomic coherence transaction with a split transaction

upgrade (RequestForShared or RequestForExclusive) or downgrade (WritebackExclusive) its coherence permissions for a block. Processors can silently evict Shared blocks, so there is no WritebackShared request. When the owner of a block (a processor in Modified or Owned or the home memory if no processor owns the block) observes a request for upgraded permissions, it responds with the data. When a downgrading owner processor observes its own WritebackExclusive (since requests are broadcast to all nodes, including the requestor), it sends the data to the home memory, which is now the owner.

The cache controllers and memory controllers are FSMs with some number of states and events. While traditional protocols had few, if any, transient states, modern protocols such as ours have numerous transient states. Moreover, they have more events, since split transactions have more message exchanges than simple atomic transactions. In Figure 2, we illustrate the difference in complexity between an atomic transaction, such as might occur in a simple bus-based system, and a split transaction in our protocol. States are circles and transitions are arcs labeled with the events that cause them. The greater number of transient states and events reflect the additional concurrency and asynchrony in our system. While the top path through the split transaction corresponds to a typical scenario in a bus-based split transaction (since there can be a delay between issuing a request and when it wins the bus and arrives), the bottom path is non-intuitive and we will explain how it can happen in Section 2.2.

Coherence controller complexity increases the importance of end-to-end error detection. With additional states and events, the probability of a fault causing an error increases. Instead of adding low-level, localized error-detection mechanisms to accommodate additional error models, we propose checking end-to-end invariants that are largely implementation-independent.[1] For example, the coherence checker that we develop in Section 5 dynamically verifies that every coherence upgrade has a corresponding downgrade. No matter how complicated a coherence protocol might be, this invariant must still hold.

2.2 Interconnection Network

Unlike traditional bus-based SMPs, we interconnect the nodes in our system with a broadcast tree. Designers have turned away from shared-wire buses because of the electrical complications involved in scaling their bandwidths to accommodate more demanding processors [6]. Like a bus, a broadcast tree can provide the total order of broadcast coherence requests that snooping coherence protocols require, but it enables the use of point-to-point signalling instead of shared wires. A processor unicasts a coherence request up to the root of the tree, which is the point at which the interconnect inserts requests into the total order, and then the root broadcasts the request down the tree. Data response messages also travel on the physical links of the broadcast tree, but they do not need to be broadcast.

Using a broadcast tree instead of a bus can violates an invariant that has traditionally simplified system design. Broadcast requests may no longer be synchronous, i.e., a request does not have to arrive at each node at the same time. Differing levels of contention within the tree, due to flow control at the leaves which backs up the tree, could cause a request to arrive at one node before it arrives at another node. Moreover, as a result, the data response to a request can arrive at the requestor before its own request (as shown in the bottom path of Figure 2). Because of the asynchrony, processor P1 can broadcast a request that arrives at the owner, P2, long before it arrives back at P1. In the meanwhile, P2 has already sent the response to P1 and it arrives first.

Interconnection network complexity can lead to errors that systems cannot currently detect. For example, a multi-bit fault that corrupted a broadcast request (traveling down the tree) in a non-root switch could cause that request to not be delivered to a subset of the nodes. Also, a transient fault in the arbitration logic of a switch could cause a newer message to bypass an older message that had been buffered due to flow control and thus re-order the arrival of broadcast coherence requests at a node. Once again, instead of adding low-level error detection to target these new error models, we would prefer an end-to-end checker. The message-level checker that we develop in Section 6 tests the end-to-end invariant that all nodes observe the same total order of broadcast requests; thus, it detects all message corruptions, drops, and re-orderings across the system.

1. We cannot necessarily discard the low-level detectors, since the system will need some of them for fault diagnosis. However, we can take them off the critical path, since fault diagnosis occurs only after error detection.

2.3 End-To-End Invariants to Check

While we could dynamically verify a number of system-wide invariants, we present here two specific checkers that are motivated by the previous two subsections. Both invariants are independent of the implementation and we could thus also check them to dynamically verify SMPs which satisfy the same properties with simpler implementations.

- In Section 5, we present a technique for dynamically verifying that every cache coherence upgrade has a corresponding coherence downgrade at another node (or nodes) in the system.
- In Section 6, we present a technique for dynamically verifying that every node in the system observes the same total order of cache coherence requests.

3 Hardware Backward Error Recovery with SafetyNet

While checker failure could trigger a system crash, higher availability can be achieved by invoking a backward error recovery (BER) scheme. We choose the recently-developed SafetyNet BER mechanism [16], an all-hardware scheme that can hide the long error detection latencies incurred by end-to-end invariant checking. SafetyNet periodically checkpoints the system state, to allow the system to recover its state to a consistent previous checkpoint. If the checkers detect an error, SafetyNet recovers the state to the *recovery point*, the old checkpoint most recently validated error-free. Checkpoints between the recovery point and the active system state are pending validation. A system-wide checkpoint includes the state of the processor registers, memory values, and coherence permissions. SafetyNet handles system I/O with the standard solution of buffering inputs and delaying outputs until validation [9]. We provide an overview of its operation, but we refer interested readers to Sorin et al. [16] for implementation specifics and more detailed evaluation.

Checkpointing Via Logging. Logically, SafetyNet checkpoints contain a complete copy of the system's architectural state. SafetyNet explicitly checkpoints registers and incrementally checkpoints memory state by logging previous values and coherence permissions. Conceptually, processors and memory controllers log every change to the memory/coherence state (i.e., save the *old* copy of the block) whenever they might have to undo an action (i.e., a store or a transfer of ownership). To reduce storage and bandwidth requirements, SafetyNet only logs the block on the first such action per checkpoint interval. By using coarse checkpoint intervals, SafetyNet significantly reduces logging overhead.

Creating Consistent Checkpoints. All of the components (cache and memory controllers) coordinate their local checkpoints, so that the collection of local checkpoints represents a consistent global recovery point. Coordinated system-wide checkpointing avoids both cascading rollbacks [8] and an output commit problem [9] for inter-node communication. SafetyNet coordinates checkpoints across the system in *logical time* to avoid a potentially costly exchange of synchronization messages. To ensure that checkpoints reflect consistent system states, the logical time base must ensure that all components can independently determine the checkpoint interval in which any coherence transaction occurs (not just its request). One basis of logical time in an SMP is for each component to create a checkpoint after every T_c broadcast coherence requests (i.e., logical cycles). All nodes can agree when a transaction occurred, since a transaction *appears* in retrospect (i.e., after it has completed) to have occurred atomically in logical time, at the time of the request. Thus all nodes can agree that the T_c^{th} transaction happened before the checkpoint and the T_c+1^{th} transaction happened after it. A lost coherence request is not a problem because, even though this error can affect a node's current logical time, it does not affect any node's view of the recovery point (i.e., a checkpoint from before the error occurred). Thus, when the checkers detect the error (the lost request), all nodes will still recover to the same recovery point.

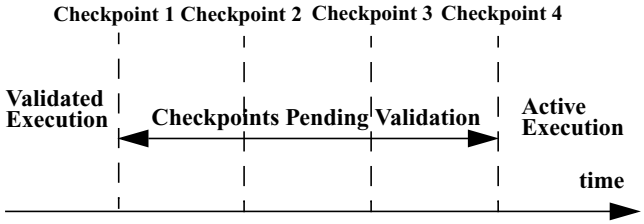

FIGURE 3. Pipelined checkpoint validation

Validating Checkpoints off the Critical Path. Checkpoint validation is the process of determining that a checkpoint is error-free (e.g., using the checkers developed later) and can be made the new recovery point. The key to enabling long-latency error detection is that SafetyNet can pipeline validation and perform it in the background. Figure 3 illustrates an example in which the active execution can proceed unimpeded while the system works to validate Checkpoints 2-4.

Recovering to a Consistent Global State. If the checkers detect an error, SafetyNet restores the globally consistent recovery point. Recovery requires that the processors restore their register checkpoints and that the caches and memories unroll their local logs to recover the system to the consistent global state at the pre-error recovery point. The system discards all state associated with transactions in progress at the time of recovery, since this state is (by definition) unvalidated state that occurs logically after the recovery point. After recovery, the system reconfigures, if necessary, and resumes execution.

4 End-To-End Checking with Signature Analysis

In this section, we describe in general how to use *distributed signature analysis* to detect violations of end-to-end system invariants due to errors. Signature analysis takes a large amount of input data—in this case, system states and events—and produces a small output, called a *signature*, that almost-uniquely characterizes the large amount of input data. In our schemes, in which events are incoming coherence requests, compression is necessary, since coherence requests are eight bytes each and can occur every network cycle. The idea of signature analysis has existed for a long time, and it is widely used in built-in self-test (BIST) [1]. However, to the best of our knowledge, this research is the first to use distributed signature analysis to dynamically verify end-to-end invariants of shared memory multiprocessors.

All of the N components in a distributed signature analysis scheme maintain a local signature, $S(i,k)$, where i is the identity of the component and k is the number of events (of interest) observed thus far at component i. The local signature is updated for every event, and we denote the k^{th} event at component i by $E(i,k)$. When unambiguous, we will denote an event simply as E, for clarity of notation. Components update their signatures according to an update function U that takes two parameters: $S(i,k-1)$ and $E(i,k)$. Thus, $S(i,k) = U[S(i,k-1), E(i,k)]$. We assume that components process events in order of occurrence. To check for errors, a centralized checker periodically performs a global reduction of the local signatures.[2] The checking function, C, periodically (e.g., after every T_c events) takes all of the local signatures from the N components as its variables and produces a boolean result of the form $C[S(0,nT_c), S(1,nT_c), \ldots, S(N-1,nT_c)] = \{true, false\}$, where *true* denotes that the checker detected an error. We illustrate the general situation in Figure 4.

Designing a distributed signature analysis scheme entails choosing the two functions, U and C. We want to choose the function U so that the same signature has an arbitrarily low probability of characterizing two different histories (i.e., sequences of events), a phenomenon known as *aliasing*. We say that aliasing occurs if:

$$(S(i,k) = S(j,k)) \wedge \exists (m \le k) | E(i,m) \ne E(j,m) \qquad \text{(EQ 1)}$$

With perfect anti-aliasing in U, C will detect all violations of the invariant. However, it will not detect a violation if one of three aliasing situations arises.

- *Finite Resource Aliasing*: Engineering restrictions limit signatures to a finite number of bits, say b, and b bits can only represent 2^b sequences of events. Many typical signature analysis functions convolve the input stream with a pseudo-random number generator so as to reduce aliasing to an arbitrarily small probability. In hardware, designers often implement pseudo-random number generation with a linear feedback shift register (LFSR) [10]. We do not address this issue further in this paper, since it is simply an engineering trade-off, and we can make this type of aliasing arbitrarily small at the cost of additional hardware.
- *Signature Analysis Fault Aliasing*: Aliasing could occur because of a fault in the signature analysis hardware itself. We can protect this hardware from faults with redundancy (e.g., TMR), since it is a small fraction of total hardware, or we can consider its faults an unlikely double-fault scenario (since a signature hardware fault may mask a coherence error but not create one).
- *Inherent Aliasing*: Aliasing could occur because the chosen update function, U, inherently suffers from aliasing. For example, if an update function adds the address of the k^{th} incoming message to $S(i,k-1)$ and the address is zero, then the occurrence of this event (the incoming message) is indistinguishable from the case in which it did not occur. As such, aliasing could occur even with infinite hardware resources for implementing signature analysis. We address this form of aliasing in our examples of signature analysis, since it is a fundamental property of the schemes and not an implementation artifact.

While aliasing can cause false negatives (i.e., we can fail to detect an error), it cannot cause false positives (i.e., we will not "detect" an error that did not occur).

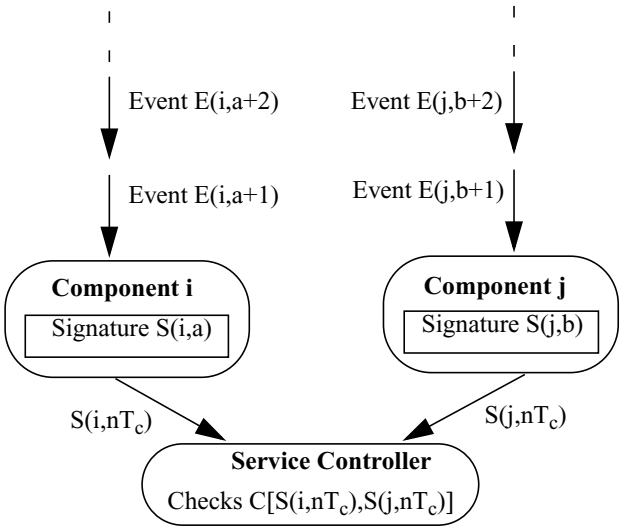

FIGURE 4. Generalized distributed signature analysis example. Components maintain signatures that they update for each event. Periodically, all components send their signatures to the service controller which then performs a reduction check on them to detect potential violations of invariants.

2. The system performs the check periodically in *logical* time, in order to obtain a consistent system view. As explained later, the system performs the check after every T_c coherence requests (i.e., every T_c logical cycles).

5 Checking Coherence-Level Invariants

In this section, we develop a scheme for testing an end-to-end invariant of the cache coherence protocol. The coherence invariant we choose to test is that every upgrade of coherence permissions at a controller (cache or memory) has an appropriate downgrade at one or more other controllers. As explained in Section 4, we test the invariant by feeding signatures, $S(i,nT_c)$, computed at each component i, into a global check function, $C[S(0,nT_c), S(1,nT_c), ..., S(N-1,nT_c)]$, where N is the number of cache and memory controllers (and thus N=2P). At a high level and assuming for now that every upgrade has only one corresponding downgrade, if an upgrade corresponds to an addition of a constant and a downgrade corresponds to a subtraction of the same constant, the global reduction should sum to zero at the end of every checkpoint interval.

The check function, C_{CL}, verifies that the global sum of all local signatures equals zero:

$$C_{CL}[S(0,nT_c),S(1,nT_c),\ldots,S(N-1,nT_c)] = \begin{pmatrix} true, \text{ if } \sum_i S(i,nT_c) \neq 0 \\ false, \text{ otherwise} \end{pmatrix} \quad \text{(EQ 2)}$$

The update function, U_{CL}, should be commutative since this invariant does not consider ordering of upgrades and downgrades. We just care that the system meets the coherence invariants within the checkpoint interval, but there are no ordering requirements within the interval itself. The update function, U_{CL}, is a function of the address of the coherence request, *Addr(E)*, i.e., *Addr(E)* is the constant that is added/subtracted for each upgrade/downgrade event.[3] An upgrade adds *Addr(E)* to *S(i,k-1)*, and a downgrade subtracts *Addr(E)* from *S(i,k-1)*. For RequestForShared and WritebackExclusive coherence requests, the update process is simple, since there is one upgrader and one downgrader. For a RequestForShared, the owner who satisfies the request, which could be memory, is the downgrader. However, since a single RequestForExclusive upgrade can cause multiple downgrades, we would need to multiply the added address by the number of controllers that should downgrade.

The challenge in developing this checker is that the upgrader does not necessarily know how many downgraders exist. Temporarily ignoring the possibility of silent downgrades from Shared (WritebackShared requests), we can solve this problem by having the response from the owner to the upgrader (i.e., the data or the acknowledgment) include the number of downgraders, since the owner can keep track of this number. However, most current protocols do allow silent WritebackShared requests, so we solve this problem differently. Our solution to not knowing the number of downgraders for a RequestForExclusive is to assume that all other cache controllers and the home memory controller are downgraders. As shown in Table 1, a RequestForExclusive requestor increments by the request's address multiplied by the number of nodes in the system (P), and the other P-1 cache controllers and the home memory controller decrement by the address (regardless of whether they have the block). A RequestForShared requestor increments by the address, and the owner who satisfies the RequestForShared request (either the memory controller or a cache controller in O or M) decrements by the address. A WritebackExclusive requestor decrements by the address and the memory controller increments by the address now that it is the owner. All of these transactions sum to zero if no errors occur, as shown in the equation for C_{CL}.

The primary cost of this signature analysis scheme is extra hardware, since the latency of performing the signature analysis is hidden. A controller holds the current value of its signature in an additional register. Each controller also requires hardware for re-computing the signature upon the arrival of every coherence request. Thus, they each need an

TABLE 1. Coherence-level update function[a]

	Event *E*	State	*S(i,k) = U_{CL}[S(i,k-1), E(i,k)]*
cache controller	Own ReqForShared	I->S	*S(i,k-1) + Addr(E)*
	Own ReqForExclusive	I,S->M	*S(i,k-1) + P*Addr(E)*
	Own WritebackExclusive	O,M->I	*S(i,k-1) - Addr(E)*
	Other ReqForShared	I,S	*S(i,k-1)*
		O,M	*S(i,k-1) - Addr(E)*
	Other ReqForExclusive	I,S,O,M	*S(i,k-1) - Addr(E)*
	Other WritebackExclusive	I,S,O,M	*S(i,k-1)*
memory controller	ReqForShared	I,S	*S(i,k-1) - Addr(E)*
		O,M	*S(i,k-1)*
	ReqForExclusive	I,S,O,M	*S(i,k-1) - Addr(E)*
	WritebackExclusive	O,M	*S(i,k-1) + Addr(E)*
	address not at home		*S(i,k-1)*

a. Shaded entries are updates with no effect, i.e., *S(i,k) = S(i,k-1)*.

3. To avoid inherent aliasing due to situations in which *Addr(E)=0*, we can use more sophisticated constants, such as *Addr(E)* recoded to have an equal number of zeros and ones. To simplify notation, though, we simply use *Addr(E)* in this discussion.

adder and a multiplier, but the multiplier need only multiply by P and therefore is not a full 64-bit multiplier.

Summary. This signature analysis scheme detects all single instances in which a coherence transaction incurs a mismatched number of coherence upgrades and downgrades.[4] For example, it can detect if a sharer did not downgrade permission to a block after receiving an invalidation. Without global information, a system cannot detect this fault. It also detects many multiple error situations, although not all. For example, a byzantine fault that caused an upgrader to downgrade and a downgrader to upgrade would go undetected.

6 Checking Message-Level Invariants

In this section, we develop a signature analysis scheme for detecting message-level errors. The end-to-end invariant we check is that every cache and memory controller observes the same total order of broadcast requests. We will show that this invariant checker can detect corrupted, dropped, or reordered messages.

We first develop a simplified update function, U_{ML}, and we will gradually describe a more sophisticated example that reduces inherent aliasing. A simple update function, U_{ML}, adds the address of the k^{th} coherence request, *Addr(E)*, to the current value of the signature, *S(i,k-1)*.

$$U_{ML}[S(i,k-1),E] = S(i,k-1) + Addr(E) \quad \text{(EQ 3)}$$

The check function, C_{ML}, detects if any of the N coherence controllers ($N=2P$) did not observe the same sequence of broadcasts as the rest of the components:

$$C_{ML}[S(0,nT_c),S(1,nT_c),\ldots,S(N-1,nT_c)] = \left(\begin{array}{l} false, \text{ if } S(0,nT_c) = S(1,nT_c) = \ldots = S(N-1,nT_c) \\ true, \text{ otherwise} \end{array} \right) \quad \text{(EQ 4)}$$

Combining C_{ML} with this simple U_{ML} detects corrupted messages, some lost messages, but not reordered messages. First, we discuss inherent aliasing that hides lost messages. Imagine the case in which a fault causes cache controller *i* to lose an incoming address message for address 22, and this was the $T_c{}^{th}$ message. Moreover, the $T_c{+}1^{th}$ message is also for address 22. At this point, cache controller *i* computes the "correct" signature and sends it to the centralized checker, and the check does not detect this error due to aliasing. A simple solution to this problem is to compute U_{ML} based on more fields of the message than just the address, such as the requestor (*Req(E)*) and requestor's transaction ID (*ID(E)*)[5], for example. We denote concatenation with a vertical line.

$$U_{ML}'[S(i,k-1),E] = S(i,k-1) + (Addr(E)|Req(E)|ID(E)) \quad \text{(EQ 5)}$$

The scheme described thus far still suffers from inherent aliasing that may not detect reordered messages, an error that was discussed earlier as being possible due to switched interconnects. An update function based on addition, which is commutative, will not detect these errors, since adding Message A before Message B produces the same signature as adding them in the other order. To avoid this form of inherent aliasing requires a non-commutative function U_{ML}, such as:

$$U_{ML}''[S(i,k-1),E] = (2 \times S(i,k-1)) + (Addr(E)|Req(E)|ID(E)) \quad \text{(EQ 6)}$$

We have established two necessary qualities for U_{ML}:

- The input per message must be more than just the address, since otherwise repeated addresses can mask dropped messages.
- U_{ML} must be non-commutative, since otherwise the check will not detect re-ordered messages.

The function that we choose, U_{ML}''', is a variant of U_{ML}'' that is easier to implement in hardware. U_{ML}''' rotates *S(i,k-1)* one bit to the left (denoted by *S(i,k-1)* << *1*) and then Exclusive-ORs (XORs) the address, requestor, and ID of the incoming coherence request:

$$U_{ML}'''[S(i,k-1),E] = [S(i,k-1) \ll 1] \oplus [Addr(E)|Req(E)|ID(E)] \quad \text{(EQ 7)}$$

This function satisfies our two requirements and is also easy to implement in hardware. Similarly, we could have implemented a function using an LFSR, since signature analysis based on LFSRs is non-commutative and thus detects reordering errors, as well as corrupted or lost messages.

The primary cost of this signature analysis scheme is extra hardware, since SafetyNet hides the latency of performing the signature analysis. Each controller requires a shift register to hold *S(i,k)* as well as hardware for performing the update function, U_{ML}'''. Since the signature resides in a shift register, computation of the new signature only requires XOR logic.

Summary. This message-level checker detects all single instances of corrupted messages, dropped messages, and reordered messages.[6] It detects many multiple fault situations, although any fault that affects the reception of the message at every node in the same way will elude detection. Moreover, it detects errors that a system could not detect

4. Recall that we are ignoring finite resource aliasing and signature analysis fault aliasing.

5. A processor can generate a transaction ID by simply counting the number of requests it has made.

6. Recall that we are ignoring finite resource aliasing and signature analysis fault aliasing.

TABLE 2. Target System Parameters

L1 Cache (I and D)	128 KB, 4-way set associative
L2 Cache	4 MB, 4-way set-associative
Memory	2 GB, 64 byte blocks
Miss From Memory	180 ns (uncontended)
Checkpoint Log Buffer	512 kB total, 72 byte entries
Interconnect	bcast tree, link b/w=6.4 GB/s
Checkpoint Interval	300 broadcast requests

with strictly local information. For example, in a broadcast tree, a fault can potentially lead to reordering of messages, which violates the required total order. However, a processor that receives two messages out-of-order cannot detect that reordering has occurred without obtaining information from other nodes (e.g., comparing the order of message reception with other nodes).

7 Evaluation

We evaluate an SMP that performs both signature analysis checks just developed and invokes SafetyNet BER if either check fails. Since we cannot build a modern SMP, we evaluate our design with simulation. We present our simulation methodology [2], benchmarks, and results.

7.1 Simulation Methodology

We simulate a 16-node (*P=16, N=32)* target system with the Simics full-system, multiprocessor, functional simulator [12], and we extend Simics with a memory hierarchy simulator to compute execution times. Each node in our modern SMP consists of a processor, two levels of cache, cache controller, some portion of the shared memory, memory controller, and a network interface.

Simics. Simics is a system-level architectural simulator developed by Virtutech AB. We use Simics/sun4u, which simulates Sun Microsystems's SPARC V9 platform architecture (e.g., used for Sun E6000s) in sufficient detail to boot unmodified Solaris 8. Simics is a functional simulator only, and it assumes that each instruction takes one cycle to execute (although I/O may take longer), but it provides an interface to support detailed memory hierarchy simulation.

Processor Model. We use Simics to model a processor core that, given a perfect memory system, would execute four billion instructions per second and generate blocking requests to the cache hierarchy and beyond. We use this simple processor model to enable tractable simulation times for full-system simulation of commercial workloads. While an out-of-order processor model might affect the absolute values of the results, it would not qualitatively change them (e.g., whether an error is detected).

Memory Model. We have implemented a memory hierarchy simulator that supports the MOSI broadcast snooping cache coherence protocol. The simulator captures all state transitions (including transient states) of our coherence protocol in the cache and memory controllers. We model the interconnection network and the contention within it, including the small additional contention due to SafetyNet messages. In Table 2, we present the design parameters of our target memory system. With a checkpoint interval, T_c, of 300 broadcast coherence requests and four outstanding checkpoints, SafetyNet can tolerate fault detection latencies that are greater than the latency for global communication.

SafetyNet. Our memory system simulator models the details of the SafetyNet support for checkpoint/recovery. For evaluating overhead for checkpointing register state, we model a conservative latency of 100 cycles. We conservatively charge eight cycles for logging store overwrites (8 bytes/cycle for 64 byte cache blocks), but these are only about 0.1% of instructions.

We integrate the global reduction of the local signatures with the existing mechanism for validating checkpoints in SafetyNet. For each checkpoint that a cache or memory controller agrees to validate, it computes both signatures based on the T_c coherence requests (i.e., address messages) it processed in that checkpoint interval and sends these signatures to the system service processor (i.e., a central controller often found in servers, such as the Sun E10000 [6]). The service processor performs the checks C_{CL} and C_{ML}.[7] If the checks detect no errors (i.e., C_{CL}=C_{ML}=*false*), it completes the validation by notifying every node. Otherwise, it triggers a system recovery.

7.2 Workloads

Commercial applications are an important workload for high availability systems. As such, we evaluate our system with four commercial applications and one scientific application, described briefly in Table 3 and in more detail by Alameldeen et al. [2]. Using a methodology described by Alameldeen et al. [2] to address the variability in runtimes for commercial workloads, we simulate each design point multiple times with small, pseudo-random perturbations of memory latencies to cause alternative scheduling paths and provide statistically meaningful results. Error bars in our performance results represent one standard deviation in each direction from the mean.

7.3 Results

In this section, we present the results of our experiments with the two end-to-end invariant checkers used in parallel. We injected errors into the system (at an absurdly high rate) and observed their impact on the system. The only quantitative results concern the performance impact of invariant

7. In the original SafetyNet paper, a coherence controller sent a "blank" message to the service processor to indicate that it had validated a checkpoint. Here, we add two signatures as a payload to that message.

TABLE 3. Workloads

OLTP: Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 1 GB 10-warehouse database stored on five raw disks and an additional dedicated database log disk. There are 8 simulated users per processor. We warm up for 10,000 transactions, and we run for 500 transactions.
Java Server: SPECjbb2000 is a server-side java benchmark that models a 3-tier system with driver threads. We used Sun's HotSpot 1.4.0 Server JVM. Our experiments use 24 threads and 24 warehouses (~500 MB of data). We warm up for 100,000 transactions, and we run for 50,000 transactions.
Static Web Server: We use Apache 1.3.19 (`www.apache.org`) for SPARC/Solaris 8, configured to use pthread locks and minimal logging as the web server. We use SURGE to generate web requests. We use a repository of 2,000 files (totalling ~50 MB). There are 10 simulated users per processor. We warm up for ~80,000 requests, and we run for 5000 requests.
Dynamic Web Server: Slashcode is based on a dynamic web message posting system used by `slashdot.com`. We use Slashcode 2.0, Apache 1.3.20, and Apache's `mod_perl` 1.25 module for the web server. MySQL 3.23.39 is the database engine. The database is a snapshot of `slashcode.com`, and it contains ~3,000 messages. A multithreaded driver simulates browsing and posting behavior for 3 users per processor. We warm up for 240 transactions, and we run for 50 transactions.
Scientific Application: We use *barnes-hut* from the SPLASH-2 suite [17], with the 16K body input set. We measure from the start of the parallel phase to avoid measuring thread forking.

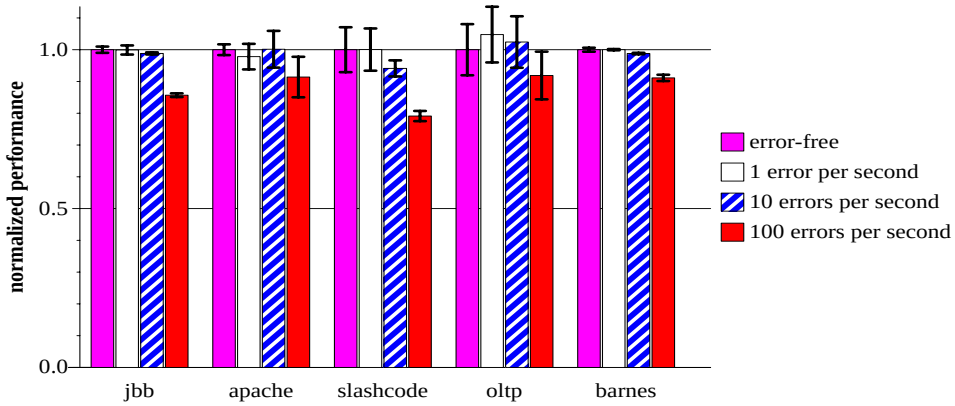

FIGURE 5. Performance as a function of error rate

checking. As would be expected from results shown in Sorin et al. [16], the performance degradation caused by SafetyNet overhead is low. Remarkably, as shown in Figure 5, the performance impact even in the presence of ten errors (i.e., ten system recoveries) per second is low for all benchmarks.

Coherence-level Checking. We injected errors into the system, including dropped messages and incorrectly processed messages, and the signature analysis indeed detected all of these errors. This signature analysis scheme can detect errors that are difficult or impossible to detect with localized error detection schemes. If a Shared node processed an incoming RequestForExclusive from another node but did not invalidate its copy of the block, then the system can violate both cache coherence and memory consistency.

Message-Level Checking. To test the ability of this checker to detect errors in this error model, we periodically dropped and reordered messages. The signature analysis scheme successfully detected the errors and triggered SafetyNet recoveries of the system. This signature analysis scheme also can detect some errors that are difficult to detect with localized error detection schemes. Most notably, it is difficult to detect in a broadcast snooping system if the interconnect erroneously reorders two broadcast requests that it delivers to a given node.

8 Related Work

Prior research has explored dynamic verification at different levels. At the intra-processor level, DIVA [3] adds a simple, provably correct checker core that dynamically verifies the aggressively designed core. DIVA can detect device errors in either core and design errors in the aggressive core, but it limits itself to the processor. The Thinking Machines CM-5 [11] dynamically computed a variant of Kirchoff's Current Law (i.e., inflow of messages equals outflow of messages) to determine when all messages had been delivered. Cantin et al. propose a scheme for dynamically verify snooping cache coherence protocols [5]. This scheme uses a validation bus, and a node that changes coherence state broadcasts its new state on this bus so that other nodes can

check that their states are compatible. This scheme, however, requires manual construction of the checker protocol and significant extra bandwidth for validation, and it does not provide a way to integrate it with a recovery mechanism. Cain and Lipasti propose an algorithm based on vector clocks for dynamically verifying sequential consistency, but they leave for future work the issues of implementation and integration of the algorithm with a checkpoint/recovery mechanism [4].

9 Conclusions

In this paper, we have argued for dynamic verification of end-to-end, system-wide invariants in shared memory multiprocessors. We have developed two signature analysis schemes for detecting violations of system-wide invariants, and we have used full-system simulation to demonstrate that they detect the targeted errors while not degrading system performance. The viability of dynamic verification of end-to-end invariants, in conjunction with backward error recovery, enables improved system availability.

While this work applies the end-to-end argument to encompass SMP coherence protocols and interconnects, future work can seek end-to-end approaches that encompass complete SMP hardware (e.g., by integrating processor dynamic verification [3]) or even software. Future work will also address the issue of fault diagnosis.

Acknowledgments

We thank Alvy Lebeck, Milo Martin, the Duke Systems Group, and the Wisconsin Multifacet Group for helpful discussions of this research. We thank Trey Cain and Jason Cantin for feedback on the final draft.

This work is supported in part by the National Science Foundation, with grants EIA-9971256, CCR-0105721, and EIA-0205286, an Intel Fellowship (Sorin), a Warren Faculty Scholarship (Sorin), two Wisconsin Romnes Fellowships (Hill and Wood), Universitat Politècnica de Catalunya and Secretaría de Estado de Educación y Universidades de España (Hill sabbatical), and donations from IBM, Intel, Microsoft, and Sun.

References

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[2] Alaa R. Alameldeen, Milo M.K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2), February 2003.

[3] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, November 1999.

[4] Harold W. Cain and Mikko H. Lipasti. Verifying Sequential Consistency Using Vector Clocks. In *Revue in conjunction with Symposium on Parallel Algorithms and Architectures*, August 2002.

[5] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Dynamic Verification of Cache Coherence Protocols. In *Workshop on Memory Performance Issues*, June 2001. In conjunction with ISCA.

[6] Alan Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998.

[7] David E. Culler and J.P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[8] E.N. Elnozahy, D.B. Johnson, and Y.M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, September 1996.

[9] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[10] S. W. Golumb. *Shift Register Sequences*. Aegean Park Press, revised edition, 1982.

[11] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[12] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[13] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in Systems Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[14] Ashok Singhal et al. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of 4th Hot Interconnects Symposium*, pages 41–52, August 1996.

[15] Daniel J. Sorin. *Using Lightweight Checkpoint/Recovery to Improve the Availability and Designability of Shared Memory Multiprocessors*. PhD thesis, University of Wisconsin, August 2002.

[16] Daniel J. Sorin, Milo M.K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.

[17] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.