Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures*

Albert Meixner[1] and Daniel J. Sorin[2]

[1]Dept. of Computer Science
Duke University
albert@cs.duke.edu

[2]Dept. of Electrical and Computer Engineering
Duke University
sorin@ee.duke.edu

Abstract

Multithreaded servers with cache-coherent shared memory are the dominant type of machines used to run critical network services and database management systems. To achieve the high availability required for these tasks, it is necessary to incorporate mechanisms for error detection and recovery. Correct operation of the memory system is defined by the memory consistency model. Errors can therefore be detected by checking if the observed memory system behavior deviates from the specified consistency model. Based on recent work, we design a framework for dynamic verification of memory consistency (DVMC). The framework consists of mechanisms to verify three invariants that are proven to guarantee that a specified memory consistency model is obeyed. We describe an implementation of the framework for the SPARCv9 architecture, and we experimentally evaluate its performance using full-system simulation of commercial workloads.

1 Introduction

Computer system availability is crucial for the multithreaded (including multiprocessor) systems that run critical infrastructure. Unless architectural steps are taken, availability will decrease over time as implementations use larger numbers of increasingly unreliable components in search

*This technical report includes material from the following paper: Albert Meixner and Daniel J. Sorin. "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architecture." *International Conference on Dependable Systems and Networks (DSN)*, June 2006. This technical report includes a proof of correctness (Appendix A) that is not in that paper, additional empirical data on DVMC scalability., as well as a complete description of the cache coherence checker that was omitted due to space constraints.

of higher performance. Backward error recovery (BER) is a cost-effective mechanism [26, 21] to tolerate such errors, but it can only recover from errors that are detected in a timely fashion. Traditionally, most systems employ localized error detection mechanisms, such as parity bits on cache lines and memory buses, to detect errors. While such specialized mechanisms detect the errors that they target, they do not comprehensively detect whether the *end-to-end* [24] behavior of the system is correct. Our goal is end-to-end error detection for multithreaded memory systems, which would subsume localized mechanisms and provide comprehensive error detection.

Our previous work [16] achieved end-to-end error detection for a very restricted class of multithreaded memory systems. In that work, we designed an all-hardware scheme for *dynamic verification* (online checking) of sequential consistency (DVSC), which is the most restrictive consistency model. Since the end-to-end correctness of a multithreaded memory system is defined by its memory consistency model, DVSC comprehensively detects errors in systems that implement sequential consistency (SC). However, DVSC's applications are limited because SC is not frequently implemented.

In this paper, we contribute a general framework for designing dynamic verification hardware for a wide range of memory consistency models, including all those commercially implemented. Relaxed consistency models, discussed in Section 2, enable hardware and software optimizations to reorder memory operations to improve performance. Our framework for dynamic verification of memory consistency (DVMC), described in Section 3, combines dynamic verification of three invariants to check memory consistency. In Section 4 we describe a checker design for each invariant and give a SPARCv9 based implementation of DVMC. Section 5 introduces the experimental methodology used to evaluate DVMC. We present and analyze our results in Section 6. Section 7 compares our work with prior work on dynamic verification. In Appendix A, we for-

mally prove that the mechanisms from Section 4 verify the three invariants introduced in Section 3 and that these invariants guarantee memory consistency.

2 Background

This work addresses dynamic verification of shared memory multithreaded machines, including simultaneously multithreaded microprocessors [27], chip multiprocessors, and traditional multiprocessor systems. For brevity, we will use the term *processor* to refer to a physical processor or a thread context on a multithreaded processor. We now describe the program execution model and consistency models.

2.1 Program Execution Model

A simple model of program execution is that a single thread of instructions is sequentially executed in *program order*. Modern microprocessors maintain the illusion of sequential execution, although they actually process instructions in parallel and out of program order. To capture this behavior and the added complexity of multi-threaded execution, we must be precise when referring to the different steps necessary to process a memory operation (an instruction that reads or writes memory). A memory operation *executes* when its results (e.g., load value in destination register) become visible to instructions executed on the same processor. A memory operation *commits* when the state changes are finalized and can no longer be undone. In the instant at which the state changes become visible to other processors, a memory operation *performs*. A more formal definition of performing a memory operation can be found in Gharachorloo et al. [9].

2.2 Memory Consistency Models

An architecture's memory consistency model [1] specifies the interface between the shared memory system and the software. It specifies the allowable software-visible interleavings of the memory operations (loads, stores, and synchronization operations) that are performed by the multiple threads. For example, SC specifies that there exists a total order of memory operations that maintains the program orders of all threads [12]. Other consistency models are less restrictive than SC, and they differ in how they permit memory operations to be reordered between program order and the order in which the operations perform. These reorderings are only observed by other processors, but not by the processor executing them due to the in-order program execution model.

We specify a consistency model as an *ordering table*, similar to Hill et al. [11]. Columns and rows are labeled with the memory operation types supported by the system, such as load, store, and synchronization operations (e.g., memory barriers). When a table entry contains the value *true*, the operation type OP_x in the entry's row label has a performance ordering constraint with respect to the operation type in the entry's column label OP_y. If an ordering constraint exists between two operation types, OP_x and OP_y, then all operations of type OP_x that appear before any operation Y of type OP_y in program order must also perform before Y.

Table 1 shows an ordering table for processor consistency (PC). In PC, an ordering requirement exists between a load and all stores that follow it in program order. That is, any load X that appears before any store Y in the program order also has to perform before Y. However, no ordering requirement exists between a store and subsequent loads. Thus, even if store Y appears before load X in program order, X can still perform before Y.

TABLE 1. Processor Consistency

1st \ 2nd	Load	Store
Load	true	true
Store	false	true

A truth table is not sufficient to express all conceivable memory consistency models, but a truth table can be constructed for all commercially implemented consistency models.

3 Dynamic Verification Framework

Based on the definitions in Section 2 we devise a framework that breaks the verification process into three invariants that correspond to the three steps necessary for processing a memory operation (shown in Figure 1). First, memory operations are read from the instruction stream in program order ($<_p$) and executed by the processor. At this point, operations impact microarchitectural state but not committed architectural state. Second, operations access the (highest level) cache in a possibly different order, which we call cache order ($<_c$). Consistency models that permit reordering of cache accesses enable hardware optimizations such as write buffers. Some time after accessing the cache, operations perform and become visible in the globally shared memory. This occurs when the affected data is written back to memory or accessed by another processor. At the global memory, cache orders from all processors are combined into one global memory order ($<_m$).

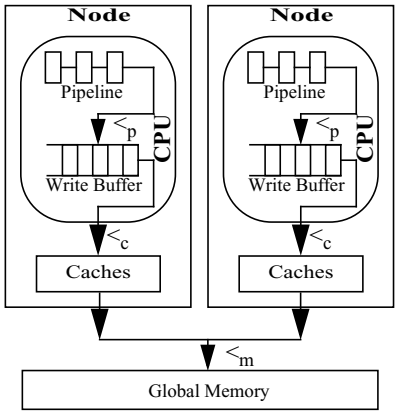

Figure 1. Operation Orderings in the System

Each of the three steps described above introduces different error hazards, which can be dealt with efficiently at the time an operation takes the respective step. The basic idea of the presented framework is to dynamically verify an invariant for every step to guarantee it is done correctly and thus verify that the processing of the operation as a whole is error-free. The three invariants *(Uniprocessor Ordering*, *Allowable Reordering*, and *Cache Coherence)* described below are sufficient to guarantee memory consistency as defined below, which we derive from Gharachorloo et. al [9]. We formally prove that these three invariants ensure memory consistency in Appendix A.2.

Definition 1: *An execution is consistent with respect to a consistency model with a given ordering table if there exists a global order* $<_m$ *such that*

- *for X and Y of type* OP_x *and* OP_y*, it is true that if* $X <_p Y$ *and there exists an ordering constraint between* OP_x *and* OP_y*, then* $X <_m Y$*, and*
- *a load Y receives the value from the most recent of all stores that precede Y in either the global order* $<_m$ *or the program order* $<_p$*.*

Uniprocessor Ordering. On a single-threaded system, a program expects that the value returned by a load equals the value of the most recent store in program order to the same memory location. In a multithreaded system, obeying *Uniprocessor Ordering* means that every processor should behave like a uniprocessor system unless a shared memory location is accessed by another processor.

Allowable Reordering. To improve performance, microprocessors often do not perform memory operations in program order. The consistency model specifies which reorderings between program order and global order are legal. For example, SPARC's Total Store Order allows a load to be performed before a store to a different address that precedes it in program order, while this reordering would violate SC. In our framework, legal reorderings are specified in the ordering table.

Cache Coherence. A memory system is *coherent* if all processors observe the same history of values for a given memory location. A coherent memory is the basis for all shared-memory systems of which we are aware (including those made by Intel, Sun, IBM, AMD, and HP), although relaxed consistency models do not strictly require coherence. Beyond coherence DVMC requires that the memory system observes the Single-Writer/Multiple-Reader (SWMR) property. Although this requirement is stronger than coherence, we consider it part of the cache coherence invariant because virtually all coherence protocols use SWMR to ensure coherence. We do not consider systems without coherent memory or SWMR in this paper.

A system that dynamically verifies all three invariants in the DVMC framework obeys the consistency model specified in the ordering table, *regardless of the mechanisms used to verify each invariant*. Our approach is conservative in that these conditions are sufficient but not necessary for memory consistency. General consistency verification without the possibility of false positives is NP-hard [10] and therefore not feasible at runtime. DVMC's goal is to detect transient errors, from which we can recover with BER. DVMC can also detect design and permanent errors, but for these errors forward progress cannot be guaranteed. Errors in the checker hardware added by DVMC can lead to performance penalties due to unnecessary recoveries after false positives, but do not compromise correctness.

4 Implementation of DVMC

Based on the framework described in Section 3, we added DVMC to a simulator of an aggressive out-of-order implementation of the SPARC v9 architecture [28]. SPARC v9 poses a special challenge for consistency verification, because it allows runtime switching between three different consistency models: Total Store Order (TSO), Partial Store Order (PSO), and Relaxed Memory

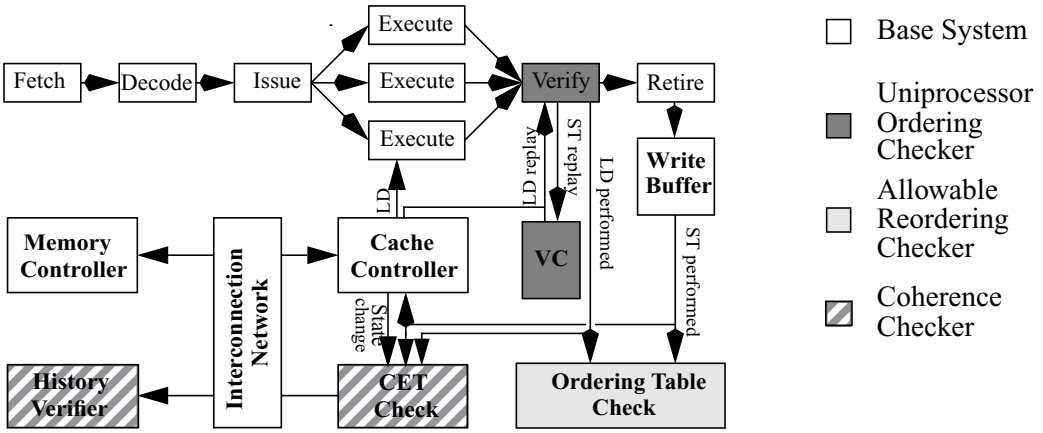

Figure 2. Simplified pipeline for *DVMC.* *Single node shown. Several structures (memory, caches, MET,...) omitted for clarity.*

TABLE 2. Total Store Order

1st \ 2nd	Load	Store
Load	true	true
Store	false	true

TABLE 3. Partial Store Order

1st \ 2nd	Load	Store	Stbar
Load	true	true	false
Store	false	false	true
Stbar	false	true	false

*Note: **Stbar** provides Store-Store ordering and is equivalent to **Membar #SS***

TABLE 4. Relaxed Memory Order

1st \ 2nd	Load	Store	Membar
Load	false	false	#LS | #LL
Store	false	false	#SL | #SS
Membar	#LL | #SL	#LS | #SS	false

***#LL**: Load-Load Ordering, **#LS**: Load-Store Ordering*
***#SL**: Store-Load Ordering, **#SS**: Store-Store Ordering*

TABLE 5. Implemented Optimizations

Model	Optimization	Effect
TSO	In-Order Write Buffer	Moves store cache misses off the critical path
PSO	Out-of-Order Write Buffer	Optimized store issue policy to reduce write buffer stalls and coherence traffic
RMO	Out-of-Order Load Execution	Eliminate pipeline squashes caused by load-order mis-speculation

Order (RMO). TSO is a variant of Processor Consistency, a common class of consistency models that includes Intel IA-32 (x86). PSO is a SPARC-specific consistency model that relaxes TSO by allowing reorderings between stores. RMO is a variant of Weak Consistency that is similar to the consistency models for PowerPC and Alpha. DVMC enables switching between models by using three ordering tables (Table 2-Table 4). Atomic read-modify-write operations (e.g., *swap*) must satisfy ordering requirements for both store and load. SPARC v9 also features a flexible memory barrier instruction (Membar) that allows exact specification of operation order in a 4-bit mask.

The bitmask contains one bit for load-load (LL), load-store (LS), store-load (SL), and store-store (SS) ordering. To incorporate such membars, Table 4's entries in the *Membar* rows and columns contain masks instead of boolean values. A boolean value is obtained from the mask by computing the logical *AND* between the mask in the instruction and the mask in the table. If the result is non-zero, ordering is required.

We started with a baseline system that supports only sequential consistency but obtains high performance through load-order speculation and prefetching for both loads and stores. We then implemented the optimizations described in Table 5 to take advantage of the relaxed consistency models. The remainder of the section describes the three verification mechanisms that were added to the system, as shown in Figure 2.

4.1 Uniprocessor Ordering Checker

Uniprocessor Ordering is trivially satisfied when all operations execute sequentially in program order. Thus, *Uniprocessor Ordering* can be dynamically verified by comparing all load results obtained during the original out-of-order execution to the load results obtained during a subsequent sequential execution of the same program [8, 5, 3]. Because instructions commit in program order, results of sequential execution can be obtained by replaying all memory operations when they commit. Replay of memory accesses occurs during the *verification stage*, which we add to the pipeline before the retirement stage. During replay, stores are still speculative and thus must not modify architectural state. Instead they write to a dedicated *verification cache* (VC). Replayed loads first access the VC and, on a miss, access the highest level of the cache hierarchy (bypassing the write buffer). The load value from the original execution resides in a separate structure, but could also reside in the register file. In case of a mismatch between the replayed load value and the original load value, a *Uniprocessor Ordering* violation is signalled. Such a vio-

lation can be resolved by a simple pipeline flush, because all operations are still speculative prior to verification. Multiple operations can be replayed in parallel, independent of register dependencies, as long as they do not access the same address.

In consistency models that require loads to be ordered (i.e., loads appear to have executed only after all older loads performed), the system speculatively reorders loads and detects load-order mis-speculation by tracking writes to speculatively loaded addresses. This mechanism allows stores from other processors to change any load value until the load passes the verification stage, and thus loads are considered to perform only after passing verification. To prevent stalls in the verification stage, the VC must be big enough to hold all stores that have been verified but not yet performed.

In a model that allows loads to be reordered, such as RMO, no speculation occurs and the value of a load cannot be affected by any store after it passes the execution stage. Therefore a load is considered to perform after the execution stage in these models, and replay strictly serves the purpose of verifying *Uniprocessor Ordering*. Since load ordering does not have to be enforced, load values can reside in the VC after execution and be used during replay as long as they are correctly updated by local stores. This optimization, which has been used in dynamic verification of single-threaded execution [7], prevents cache misses during verification and reduces the pressure on the L1 cache.

4.2 Allowable Reordering Checker

DVMC verifies *Allowable Reordering* by checking all reorderings between program order and cache access order (described in Section 3) against the restrictions defined by the ordering table. The position in program order is obtained by labeling every instruction X with a sequence number, seqX, that is stored in the ROB during decode. Since operations are decoded in program

order, seqX equals X's rank in program order. The rank in perform order is implicitly known, because we verify *Allowable Reordering* when an operation performs. The *Allowable Reordering* checker uses the sequence numbers to find reorderings and check them against the ordering table. For this purpose, the checker maintains a counter register for every operation type OP_x (e.g., load or store) in the ordering table. This counter, $\max\{OP_x\}$, contains the greatest sequence number of an operation of type OP_x that has already performed. When operation X of type OP_x performs, the checker verifies that $seqX > \max\{OP_y\}$ for all operation types OP_y that have an ordering relation $OP_x <_c OP_y$ according to the ordering table. If all checks pass, the checker updates $\max\{OP_x\}$. Otherwise an error has been detected.

It is crucial for the checker that all committed operations perform eventually. The checker can detect lost operations by checking outstanding operations of all operation types OP_x, with an ordering requirement $OP_x <_c OP_y$, when an operation Y of type OP_y performs. If an operation of type OP_x older than Y is still outstanding, it was lost and an error is detected. In our implementation, we check outstanding operations before Membar instructions by comparing counters of committed and performed memory accesses. To prevent long error detection latencies, artificial Membars are injected periodically. Membar injection does not affect correctness and has negligible performance impact since injections are infrequent (about one per 100k cycles).

The implementation of an *Allowable Reordering* checker for SPARCv9 requires three small additions to support architecture specific features: dynamic switching of consistency models, a FIFO queue to maintain the perform order of loads until verification, and computation of Membar ordering requirements from a bitmask as described earlier.

4.3 Cache Coherence Checker

Static verification of *Cache Coherence* is a well-studied problem [19,20] and more recently methods have been proposed for dynamic verification of coherence [6, 25]. *Any* coherence verification mechanism that ensures the single-writer multiple-reader principle, such as the schemes proposed by Cantin et al. [6] and Sorin et al. [25], is sufficient for DVMC. We decided to use the coherence verification mechanism introduced as part of DVSC [16], because it supports both snooping and directory protocols and scales well to larger systems.[1]

We construct the *Cache Coherence* checker around the notion of an *epoch*. An epoch for block *b* is a time interval during which a processor has permission to read (Read-Only epoch) or read and write (Read-Write epoch) block *b*. The time base for epochs can be physical or logical as long as it guarantees causality. Three rules for determining coherence violations were introduced and formally proven to guarantee coherence by Plakal et al. [18]: (1) reads and writes are only performed during appropriate epochs, (2) Read-Write epochs to not overlap other epochs temporally, and (3) the data value of a block at the beginning of every epoch equals the data value at the end of the most recent Read-Write epoch. Rules (1) and (2) enforce that the single-writer multiple-reader principle is observed, which rule (3) ensures correct propagation of data modified by writes.

The Cache Coherence Checker dynamically verifies the epoch invariants—epochs do not conflict and data is transferred correctly between epochs—with two mechanisms. First, each cache controller maintains a small amount of epoch information state—logical time at start, type of epoch, and block data—for each block it holds. For every load and store, it checks this state, called the Cache Epoch Table (CET), to make sure that the load or store is being performed in an appropriate epoch.

1. The rest of this section includes material from our previous work [16].

Second, whenever an epoch for a block ends at a cache, the cache controller sends the block address and epoch information—begin and end time, block data signature, and epoch type (Read-Write or Read-Only)—in an Inform-Epoch message to the home memory controller for that block. Epochs can end either as a result of a coherence request—another node's GetShared request (if epoch is Read-Write) or another node's GetExclusive request—or as a result of evicting a block from the cache before requesting a new block. Inform-Epoch traffic is proportional to coherence traffic, because in both cases a coherence message is sent or received at the end of the epoch. Because the coherence protocol operates independently of DVMC, sending the Inform-Epoch is not on the critical path, and no new states are introduced into the coherence protocol. Controller occupancy is also unaffected, since the actions are independent and can be done in parallel.

For each Inform-Epoch a memory controller receives, it checks that (a) this epoch does not overlap illegally with other epochs, and (b) the correct block data is transferred from epoch to epoch. The memory controller performs these checks using per-block epoch information it keeps in its directory-like Memory Epoch Table (MET).

Details of Cache Controller and CET Operation. Each cache has its own CET, which is physically separate from the cache to avoid slowing cache accesses. A CET entry corresponds to a cache line and stores 34 bits of information: the type of epoch (1 bit to denote Read-Only or Read-Write); the logical time (16 bits) and the data block (data blocks are hashed down to 16 bits, as we discuss later in this section) at the beginning of the epoch; and a DataReadyBit to denote that data has arrived for this epoch (recall that an epoch can begin before data arrives). We add an error correcting code (ECC) to each line of the cache (not the CET) to ensure that the data block does not change unless it is written by a store; otherwise, silent corruptions of cache state would be uncor-

rectable. An alternative design would use an error detecting code (EDC), but SafetyNet, the backward error recovery mechanism we use in this paper, requires ECC on all cache lines. When an epoch for a block ends, the cache controller sends an Inform-Epoch to the block's home node. An Inform-Epoch consists of the block address, the type of epoch, the logical time for the beginning and end of the epoch, and a checksum of the block data at beginning and end of the epoch. For a Read-Only epoch, the second checksum can be omitted, since the block data cannot change during the epoch.

Details of Memory Controller and MET Operation. The memory controllers receive a stream of Inform-Epochs and have to determine if any of the Read-Write epochs overlap other epochs and if data is propagated correctly between epochs. In a naive approach this would require the MET to store all epochs observed to detect collisions with epochs described by later Epoch-Informs, which is clearly infeasible. To simplify the verification process, we require that memory controllers process Inform-Epochs in the logical time order of epoch start times. Since the order in which Epoch-Informs arrive is already strongly correlated with the Epoch begin time, incoming Inform-Epochs can be sorted by timestamp in a small fixed size priority queue.

The MET at each memory controller maintains the following state per block for which it is the home node: latest end time of any Read-Only epoch (16 bits), latest end time of any Read-Write epoch (16 bits), and data block at end of latest Read-Write epoch (hashed to 16 bits). For every Inform-Epoch the verifier processes, it checks for rule violations and then updates this state. To check for illegal epoch overlapping, the verifier compares the start time of the Inform-Epoch with the latest end times of Read-Only and Read-Write epochs in the MET. Epochs overlap illegally if either a Read-Only Inform-Epoch's start time is earlier than the latest Read-Write epoch's end time or if an Read-Write Inform-Epoch's start time is earlier than either the latest Read-Only or

Read-Write epoch's end time. To check for data propagation errors the memory controller compares the data block at the beginning of the Inform-Epoch to the data block at the end of the latest Read-Write epoch. If they are not equal data was propagated incorrectly.

The MET only contains entries for blocks that are present in at least one of the processor caches. When a block without an MET entry is requested by a processor, a new entry is constructed by using the current logical time as the last end time of an Read-Write epoch and by computing the initial checksum from the data in memory. Similar to the caches, we add an error correcting code (ECC) to each line of the memory (not the MET) to ensure that a data block does not change unless it is written by a Writeback of a Read-Write block; otherwise, silent corruptions of memory state would be uncorrectable.

Logical Time. The Cache Coherence Checker requires the use of a logical time base. Logical time is a time basis that respects causality (i.e., if event A causes event B, then event A has a smaller logical time), and there are many potential logical time bases in a system [8]. We choose two logical time bases—one for snooping and one for directories—based on their ease of implementation. For a snooping protocol, the logical time for each cache and memory controller is the number of cache coherence requests that it has processed thus far. For a directory protocol, the logical time is based on a relatively slow, loosely synchronized physical clock that is distributed to each cache and memory controller. As long as the skew between any two controllers is less than the minimum communication latency between them, then causality will be enforced and this will be a valid basis of logical time [26]. To create a total order of all operations, ties in logical time can be broken arbitrarily by the priority queue (e.g., by arrival order) since there is no causal ordering between events at the same logical time.For both types of coherence protocols, there exist numerous other options for logical time bases, but we choose these for simplicity.

One implementation challenge is the need to represent logical times with a small number of bits while avoiding wraparound problems. One upper bound on timestamp size is that a cache controller must send an Inform-Epoch before the backward error recovery (BER) mechanism becomes unable to recover to a pre-error state if that Inform-Epoch revealed a violation of memory consistency. By keeping the number of bits in a logical time small (we choose 16 bits), we can bound error detection latency and guarantee that BER can always recover from a detected error. The key engineering tradeoff is that we want to use enough bits in a logical time so that we do not need to frequently "scrub" the system of old logical times that are in danger of wraparound, but not so many bits that we waste storage and bandwidth. Old logical times can lurk in the CETs and METs due to very long epochs. Our method of scrubbing old logical times is to remember to check that an epoch time is not going to wrap around. We remember to check by keeping a small FIFO (128 entries in our experiments) at each CET—every time an epoch begins, the cache inserts into the FIFO a pointer to that cache entry and the logical time at which the epoch would wraparound. By periodically checking the FIFO, we can guarantee that a FIFO entry will reach the head of the FIFO before wraparound can occur. When it reaches the head, if the epoch is still in progress, the cache controller sends an Inform-Open-Epoch to the memory controller. This message—which contains the block address, type of epoch, block data at start of epoch, and logical time at start of epoch—notifies the memory controller that the epoch is still in progress and that it should expect only a single Inform-Closed-Epoch message sometime later. The Inform-Closed-Epoch only contains the block address and the logical time at which the epoch ended. To maintain state about open epochs, each MET entry holds a bitmask (equal to the number of processors) for tracking open Read-Only epochs and a single processor ID ($\log_2$[number of processors] bits) for tracking an open Read-Write epoch. Whenever there is an open epoch, the MET

entry does not need the last Read-Only/Read-Write logical time, so these logical times and the open epoch information can share storage space if we add an OpenEpoch bit. This saves 11 bits per MET entry in our implementation (if the number of processors is less than the number of bits in a logical time). We scrub METs in a similar fashion to CETs, by using a FIFO at the memory controllers.

Data Block Hashing. An important implementation issue is the hashing of data block values in the CETs, METs, and Inform-Epochs. Hashing is an explicit tradeoff between error coverage, storage, and interconnection network bandwidth. We use CRC-16 as a simple function to hash data blocks down to 16 bits. Aliasing (i.e., two distinct blocks mapping to the same hash value) represents a probability of a *false negative* (i.e., not detecting an error that occurs). By choosing the hash function and the value of n, we can make the probability of a false negative arbitrarily small. For example, CRC-16 will not produce false negatives for blocks with fewer than 16 erroneous bits, and it has a probability of 1/65535 of false negatives for blocks with 16 or more incorrect bits.

5 Experimental Methodology

We performed our experiments using Simics [13] full-system simulation of 8-node multiprocessors. The systems were configured with either a MOSI directory coherence protocol or a MOSI snooping coherence protocol, and the simulated processors provide support for the SPARC v9 models TSO, PSO, and RMO, as well as SC. All systems use SafetyNet [26] for backward error recovery, although any other BER scheme (e.g., ReVive [21]) would work. Configurations of the directory and snooping systems are shown in Table 6. Timing information was computed using a customized version of the Multifacet GEMS simulator [14]. We adapted the cycle-accu-

TABLE 6. Memory System Parameters

L1 Cache (I and D)	32 KB, 4-way, 64 byte lines
L2 Cache	1 MB, 4-way, 64 byte lines
Memory	2 GB, 64 byte blocks
For Directory Protocol	
Network	2D torus, 2.5 GB/s links, unordered
For Snooping Protocol	
Address Network	bcast tree, 2.5 GB/s links, ordered
Data Network	2D torus, 2.5 GB/s links, unordered
Coherence Verification	
Priority Queue	256 entries
Cache Epoch Table	34 bits per line in cache
Memory Epoch Table	48 bits per line in any cache

rate *TFSim* processor simulator [15] to support timing simulation of relaxed consistency models, and we configured it as shown in Table 7.

Because DVMC primarily targets high-availability commercial servers, we chose the Wisconsin Commercial Workload Suite [2] for our benchmarks. These workloads are described briefly in Table 8 and in more detail by Alameldeen et al. [2]. Although SPARC v9 is a 64-bit architecture, portions of code in the benchmark suite were written for the 32-bit SPARC v8 instruction set. Since these code segments were written for TSO, a system configured for PSO or RMO must switch to TSO while executing 32-bit code. Table 8 shows the average fraction of 32-bit memory operations executed for each benchmark during our experiments.

TABLE 7. Processor Parameters

Pipeline Stages	fetch,decode,execute,retire
Pipeline Width	4
Branch Predictor	YAGS
Scheduling Window	64 entries
Reorder Buffer	128 entries
Physical Registers	224 integer, 192 FP
Write Buffer	24 entries

TABLE 8. Workloads

Name	**Description**	**32bit-code**
apache 2	Static web server	5.7%
oltp	*TPCC*-like workload using *IBM DB2*	38.9%
jbb	*SPECjbb 2000* - 3-tier java system	<0.01%
slashcode	Dynamic website using *apache*, *perl* and *mysql*	21.7%
barnes	*barnes-hut* from *SPLASH2* benchmark suite	<0.01%

To handle the runtime variability inherent in commercial workloads, we run each simulation ten times with small pseudo-random perturbations. Our experimental results show mean result values as well as error bars that correspond to one standard deviation.

6 Evaluation

We used simulation to empirically confirm DVMC's error detection capability and gain insight into its impact on error-free performance. In this section, we describe the results of these experiments, and we discuss DVMC's hardware costs and interconnect bandwidth overhead.

6.1 Error Detection

We tested the error detection capabilities of DVMC by injecting errors into all components related to the memory system: the load/store queue (LSQ), write buffer, caches, interconnect switches and links, and memory and cache controllers. The injected errors included data and address bit flips; dropped, reordered, mis-routed, and duplicated messages; and reorderings and incorrect forwarding in the LSQ and write buffer. For each test, an error time, error type, and error location were chosen at random for injection into a running benchmark. After injecting the error, the simulation continued until the error was detected. Since errors become non-recoverable once the last checkpoint taken before the error expires, we also checked that a valid checkpoint was still available at the time of detection. We conducted these experiments for all four supported consistency models with both the directory and snooping systems. DVMC detected all injected errors well within the SafetyNet recovery time frame of about 100k processor cycles.

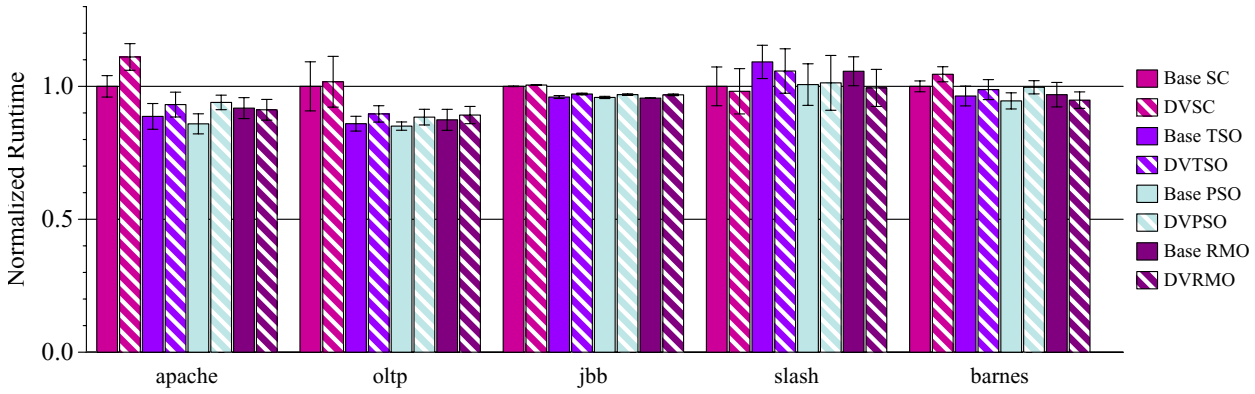

Figure 3. Workload Runtimes for Directory Coherence

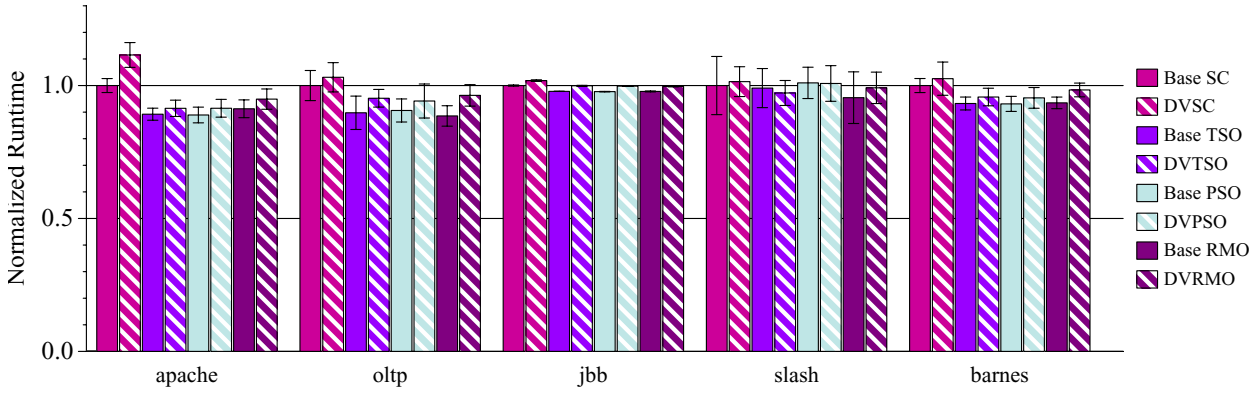

Figure 4. Workload Runtimes for Snooping Coherence

6.2 Performance

Besides error detection capability, error-free performance is the most important metric for an error detection mechanism. To determine DVMC performance, we ran each benchmark for a fixed number of transactions and compared the runtime on an unprotected system and a system implementing DVMC with different consistency models. We considered *barnes* to be a single transaction, and we ran it to completion.

6.2.1 Baseline System

Before looking at DVMC overhead, we compare the performance of unprotected systems (no DVMC or BER) with different memory consistency models. The "Base" numbers in Figure 3 and Figure 4 show the relative runtimes, normalized to SC. The addition of a write buffer in the TSO

system improves performance for almost all benchmarks. PSO and RMO do not show significant performance benefits and can even lead to performance degradation, although they allow additional optimizations that are not legal for TSO. In our experiments, the *oldest store first* strategy implicitly used by TSO performs well compared to more complicated policies. Non-speculative reordering of loads also turns out to be of little value, because load-order mis-speculation is exceedingly rare, affecting less than 0.1% of loads. Whereas the benefits from optimizations are limited, relaxed consistency models need to obey memory barriers which, even when implemented efficiently, can make performance worse than TSO.

Although most benchmarks show the expected benefits of a write buffer and the expected overhead incurred by verification, some of the *slash* results are counter-intuitive. Highly contended locks make *slash* sensitive to changes in write access timing, as indicated by high variance in run time, and it benefits from reduced contention caused by additional stalls present in SC [22].

6.2.2 DVMC Performance Overhead

DVMC can potentially degrade performance in several ways. The *Uniprocessor Ordering* checker requires an additional pipeline stage, thus extending the time during which instructions are in-flight and increasing the occupancy of the ROB and the physical registers. Load replay increases the demand on the cache and can cause additional cache misses. Coherence verification can degrade system performance due to interconnect bandwidth usage for inform messages. SafetyNet, the BER mechanism used during our tests, also causes additional interconnect traffic. Only the *Allowable Reordering* checker does not have any influence on program execution, since it operates off the critical path.

First we examine the total impact of all these factors. We run the benchmarks on an unprotected baseline system and a system implementing full DVMC as well as SafetyNet BER. The

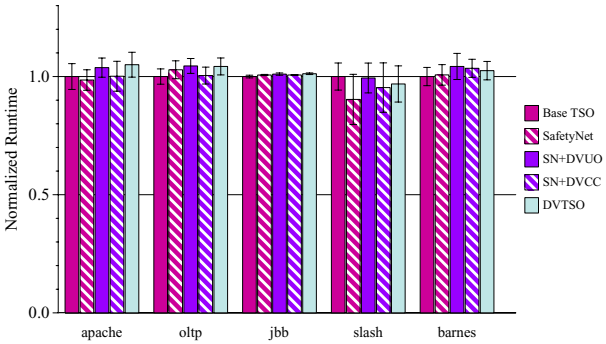

Figure 5. Verification mechanism runtimes for TSO

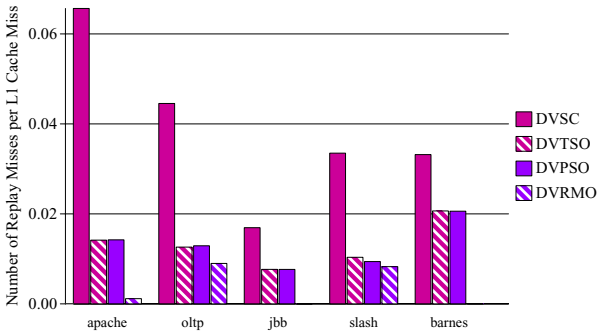

Figure 6. Cache misses during replay

benchmarks are run for all four supported consistency models and both the directory and snooping coherence systems. Figure 3 and Figure 4 show the running times of all benchmarks normalized to an unprotected system implementing SC. Despite the numerous performance hazards described, we observed no slowdown exceeding 11%. The worst slowdowns occur with SC, which is rarely implemented in modern systems. In all but 4 out of 40 DVMC configurations, the overhead is limited to 6%. Because the performance overheads are greater with the directory system and similar for TSO, PSO, and RMO, the rest of this section focuses on a directory-based system with TSO.

To study the impact of the different DVMC components, we run the same experiments with a system that only implements BER using SafetyNet (SN), a system with BER that only verifies cache coherence (SN+DVCC), a system with BER and uniprocessor ordering verification (SN+DVUO), and full DVMC with BER (DVTSO). The results of these experiments for a system implementing TSO and directory coherence are shown in Figure 5. These experiments show that Uniprocessor Ordering Verification is the dominant cause of slow-down and, although each mechanism—SafetyNet, Uniprocessor Ordering Verification, and Coherence Verification—adds a small amount of overhead in most cases, full DVTSO is no slower than SN+DVUO. Figure 5 also

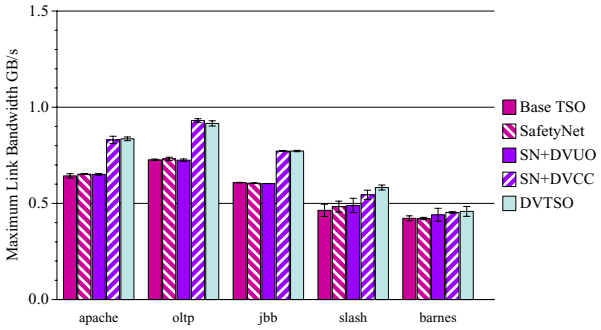

Figure 7. Interconnect Traffic for TSO

shows some unexpected speedups on *slash* when SafetyNet is added. With *slash*, SafetyNet slightly delays some writes, which can reduce lock contention and lead to a performance increase.

Figure 6 shows the number of L1 cache misses during replay normalized to the number of L1 cache misses during regular execution. Replay misses are rare, because the time between a load's execution and verification is typically small. Most replay cache misses occur when a processor unsuccessfully tries to acquire a lock and returns to the spin loop. Thus, the miss has little impact on actual performance.

DVMC can increase interconnection network utilization in two ways: the *Cache Coherence* checker consumes bandwidth for inform messages, and load replays can initiate additional coherence transactions. For the directory system with TSO, Figure 7 shows the mean bandwidth on the highest loaded link for different workloads and mechanisms. DVCC imposes a consistent traffic overhead of about 20-30%, and load replay does not have any measurable impact.

6.2.3 Scaling Sensitivity

To gain insight on how well DVMC would scale to different multiprocessor systems we ran experiments with different numbers of processors and different interconnect link bandwidth and compared the benchmark runtimes on an unprotected system to a system featuring SafetyNet and

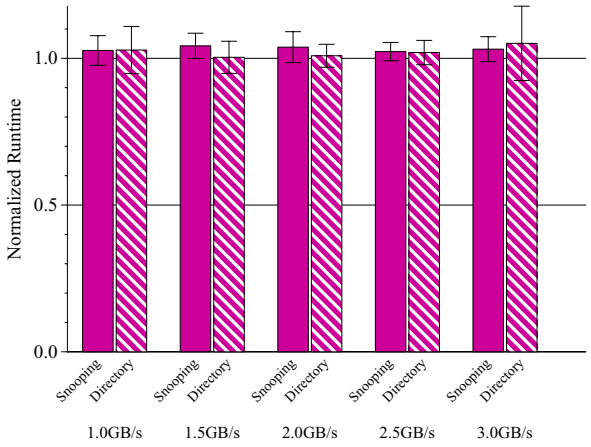

Figure 8. DVTSO overhead vs. link bandwidth

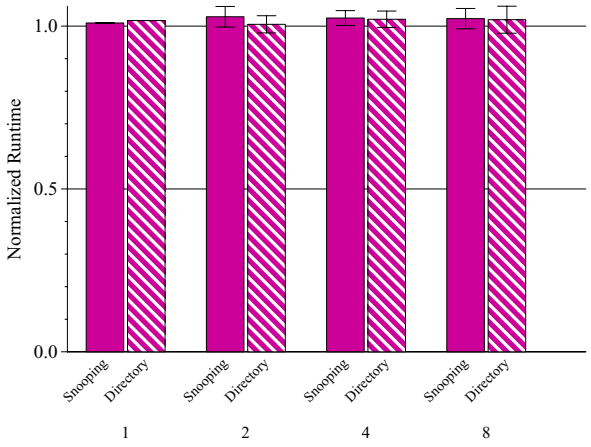

Figure 9. DVTSO overhead vs. number of processors

DVMC. All experiments were performed for both directory and snooping protocols implementing TSO.

Figure 8 shows the average over all benchmark runtimes of an 8-node system with DVTSO normalized to an unprotected system, for different link bandwidths ranging from 1GB/s to 3GB/s. Although performance overheads vary between the different link bandwidths, these variations are not statistically significant and do not show any clear correlation between link bandwidth and performance overhead. In all configurations, the full link bandwidth is only used during brief burst periods, while most DVMC related messages are transmitted during idle times between bursts. Thus, DVMC traffic has little impact on total system performance as long as the transmission can be delayed until traffic bursts are over.

Figure 9 shows the average over benchmark runtimes of systems with one to eight processors and 2.5GB/s links. The graph does not show any strong correlation between system size and DVMC performance overhead. This result is expected as DVMC traffic is all unicast and increases linearly with overall traffic, such that the bandwidth consumption stays constant.

6.3 Hardware Cost

The hardware costs of DVMC are determined by the storage structures and logic components required to implement the three checkers, but not the BER mechanism, which is orthogonal. Information on the implementation cost of SafetyNet [26], ReVive [21] and other BER schemes can be found in the literature. The *Uniprocessor Ordering* checker is the most complex checker, since it requires the addition of the VC and a new stage in the processor pipeline. These changes are non-trivial, but all required logic components are simple and storage structures are small (32-256 byte). The *Allowable Reordering* checker is the simplest and smallest checker. It requires a LSQ-sized FIFO, a set of sequence number registers, sequence numbers in the write buffer, the ordering tables, and comparators for the checker logic. The *Cache Coherence* checker also does not require complex logic, but it incurs greater storage costs. Our CET entries are 34 bits, leading to a total CET size of about 70 KB per node. The MET requires 102 KB per memory controller, with an entry size of 48 bits, but it is not latency sensitive and can be built out of cheap, long-latency DRAMs. The MET contains entries for blocks that are currently present in at least one processor cache. Entries for blocks only present at memory are constructed from the current logical time and memory value upon a cache request. To detect data errors on these blocks, DVMC requires ECC on all main memory DRAMs.

7 Related Work

In this section, we discuss prior research in dynamic verification. For verifying the execution of a single thread, there have been two different approaches. First, DIVA adds a small, simple checker core that verifies the execution of the instructions committed by the microprocessor [3]. By leveraging the microprocessor as an oracular prefetcher and branch predictor, the simple

checker can keep up with the performance of the microprocessor. Second, there have been several schemes for multithreaded uniprocessors, starting with AR-SMT [23], that use redundant threads to detect errors. These schemes leverage the multiple existing thread contexts to achieve error detection. Unlike DVMC, none of these schemes extend to the memory system or to multiple threads.

For multithreaded systems with shared memory, there are four related pieces of work. Sorin et al. [25] developed a scheme for dynamic verification of a subset of cache coherence in snooping multiprocessors. Although dynamically verifying these invariants is helpful, it is not an end-to-end mechanism, since coherence is not sufficient for implementing consistency. Cantin et al. [6] propose to verify cache coherence by replaying transactions with a simplified coherence protocol. Cain et al. [4] describe an algorithm to verify sequential consistency, but do not provide an implementation. Finally, we previously [16] designed an ad-hoc scheme for dynamic verification of sequential consistency, which does not extend to any other consistency models.

8 Conclusions

This paper presents a framework that can dynamically verify a wide range of consistency models and comprehensively detect memory system errors. Our verification framework is modular, because it checks three independent invariants that together are sufficient to guarantee memory consistency. The modular design makes it possible to replace any of our checking mechanisms with a different scheme to adapt to a specific system's design. For example, the coherence checker adapted from DVSC [16] can be replaced by the design proposed by Cantin et al. [6]. Although we used conventional multiprocessor systems as example implementations, the framework is in no way limited to these types of architectures. The simplicity of the proposed mechanisms sug-

gests that they can be implemented with small modifications to existing multithreaded systems. Although simulation of a DVMC implementation shows some decrease in performance, we expect the negative impact to be outweighed by the benefit of an end-to-end scheme for detecting memory system errors.

Acknowledgments

This work is supported in part by the National Science Foundation under grants CCF-0444516 and CCR-0309164, the National Aeronautics and Space Administration under grant NNG04GQ06G, Intel Corporation, and a Warren Faculty Scholarship. We thank Jeff Chase, Carla Ellis, Mark Hill, Alvin Lebeck, Anita Lungu, Milo Martin, Jaidev Patwardhan, and the Duke Architecture Group for their comments on this work.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[2] A. R. Alameldeen et al. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[3] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

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

[5] H. W. Cain and M. H. Lipasti. Memory Ordering: A Value-Based Approach. In *Proc. of the 31st Annual Int'l Symposium on Computer Architecture*, June 2004.

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

[7] S. Chatterjee, C. Weaver, and T. Austin. Efficient Checker Processor Design. In *Proc. of the 33rd Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pages 87–97, Dec. 2000.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. of the Int'l Conf. on Parallel Processing*, vol. I, pages 355–364, Aug. 1991.

[9] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proc. of the 17th Annual Int'l Symposium on Computer Architecture*, pages 15–26, May 1990.

[10] P. B. Gibbons and E. Korach. Testing Shared Memories. *SIAM Journal on Computing*, 26(4):1208–1244, Aug. 1997.

[11] M. D. Hill et al. A System-Level Specification Framework for I/O Architectures. In *Proc. of the Eleventh ACM Symposium on Parallel Algorithms and Architectures*, pages 138–147, June 1999.

[12] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs.

IEEE Trans. on Computers, C-28(9):690–691, Sept. 1979.
[13] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.
[14] M. M. Martin et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.
[15] C. J. Mauer, M. D. Hill, and D. A. Wood. Full System Timing-First Simulation. In *Proc. of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.
[16] A. Meixner and D. J. Sorin. Dynamic Verification of Sequential Consistency. In *Proc. of the 32nd Annual Int'l Symposium on Computer Architecture*, pages 482–493, June 2005.
[17] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. Tech. Report 2006-1, Dept. of Elec. and Comp. Engr., Duke Univ., Mar. 2006.
[18] M. Plakal et al. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proc. of the Tenth ACM Symposium on Parallel Algorithms and Architectures*, pages 67–76, June 1998.
[19] F. Pong and M. Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Computing Surveys*, 29(1):82–126, Mar. 1997.
[20] F. Pong and M. Dubois. Formal Automatic Verification of Cache Coherence in Multiprocessors with Relaxed Memory Models. *IEEE Trans. on Parallel and Distributed Systems*, 11(9):989–1006, Sept. 2000.
[21] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proc. of the 29th Annual Int'l Symposium on Computer Architecture*, pages 111–122, May 2002.
[22] R. Rajwar, A. Kägi, and J. R. Goodman. Improving the Throughput of Synchronization by Insertion of Delays. In *Proc. of the Sixth IEEE Symposium on High-Performance Computer Architecture*, pages 168–179, Jan. 2000.
[23] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proc. of the 29th Int'l Symposium on Fault-Tolerant Computing Systems*, pages 84–91, June 1999.
[24] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in Systems Design. *ACM Trans. on Computer Systems*, 2(4):277–288, Nov. 1984.
[25] D. J. Sorin, M. D. Hill, and D. A. Wood. Dynamic Verification of System-Wide Multiprocessor Invariants. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, pages 281–290, June 2003.
[26] D. J. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proc. of the 29th Annual Int'l Symposium on Computer Architecture*, pages 123–134, May 2002.
[27] D. M. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd Annual Int'l Symposium on Computer Architecture*, pages 191–202, May 1996.
[28] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

Appendix A: Proof of Framework Correctness

We now prove that dynamic verification schemes that use our framework verify memory consistency. The proof shows that the three verification mechanisms presented in Section 4 verify

their respective invariants (Section A.1) and together ensure memory consistency (Section A.2). To simplify the discussions, we assume that memory is only accessed at the word granularity.

The following proofs rely on a property called *Cache Correctness*, which states that after a store to word *w* in a cache, word *w* in the cache contains the stored value until it is overwritten. This assumption is necessary to make any statements about the contents of a cache after a sequence of stores. It can be dynamically verified using standard techniques such as error correcting codes (ECC).

Definition 2: *A cache is correct, if after a store X to word w is executed by a cache, every subsequent load of word w executed by the cache returns the value specified by X, until another store to* w *is executed.*

Cache Correctness is assumed for all storage structures, including all levels of caches, the VC, CET, MET, and main memory.

A.1 Checkers Satisfy Invariants

This section contains proofs that all of the three checkers verify their respective invariants.

A.1.1 Uniprocessor Ordering Checker Correctness

This property is guaranteed by Dynamic Verification of Uniprocessor Ordering in Section 4.1. For the proof we replace the intuitive description given previously with a formal definition.

Definition 3: *A system obeys Uniprocessor Ordering if any LD Y receives the value of the most recent ST X to the same word w in program order ($X <_p Y$), unless a ST from another processor performs after X performs but before Y performs.*

Proof 1: For the proof we consider two cases.

In the first case, there exists at least one store to *w* that precedes LD Y in program order, but is performed after LD Y. Let ST X be the most recent of these stores in program order. Because ST

X allocates an entry for *w* in the VC when it commits, and this entry can only be freed when X performs, there must be an entry for *w* in the VC at the time Y performs. By Cache Correctness, this entry contains the value of the last committed store to *w*. Since stores commit in program order, this is also the most recent store to *w* in program order, which is X. There can be no store on another processor that performs after X but before Y, as Y performs before X. Thus, Y receives the value of the most recent store in program order, X.

In the second case, all stores to w that precede Y in program order also performed before Y performs. This implies that the VC contains no entry for *w* and the replayed load value has to be obtained from the cache. If the value in the cache is from a store executed by the same processor *p* that executes Y, then by Cache Correctness it must be the value of the last store X that wrote to the cache. As the cache is written when a store performs, X is the last store to *w* performed before Y. When X performed, the corresponding entry in the VC must have been freed, since there are no uncommitted stores to *w*. During deallocation, Uniprocessor Ordering Verification ensures that the value written to the cache by X is equal to the value contained in the VC entry for *w*, which is the value of the most recent store in program order. Since they have to be identical, X is the most recent store in program order. If Y receives a value from a store Z performed on a different processor, then that Z must have overwritten the value written to the cache by X. By definition of performed, Z must have performed after X.

In both cases, the load receives a value from either the most recent store X to *w* in processor *p*'s program order or a store from a different processor performed after X performs.

A.1.2 Allowable Reordering Checker Correctness

Allowable Reordering correctness is directly verified as described in Section 4.2. Again we first give a formal definition of *Allowable Reordering*.

Definition 4: *A reordering of performed operations is allowable, if for any two operations X and Y, where X is of type OPx and Y is of type OPy, it is true that if the consistency model requires an ordering constraint between OPx and OPy and $X <_p Y$ then X performs before Y.*

Proof 2: Assume the above statement is not true and Y performs before X, although $X <_p Y$ and an ordering constraint exists between OPx and OPy.

Since $X <_p Y$ and sequence numbers are assigned in program order, then seqX < seqY. When Y performs, the reorder checker will set max{OPy} to seqY. At the time X performs, seqX < seqY <= max{OPy}. This will make the reorder checker signal an error, since it expects seqX > max{OPy} for all OPx with an ordering constraint OPx<OPy. Therefore, if the above statement is not true, an error will be signalled. If the program executes without error, the statement has to be true.

A.1.3 Cache Coherence Checker Correctness

Cache coherence can be proven using the three rules defined in Section 4.3. These rules can be dynamically enforced as described in the same section. The proof uses a definition of *Cache Coherence* from Gharachorloo et al. [9].

Definition 5: *A system is coherent if all stores to the same word w are serialized in some order and are performed in that order with respect to any processor.*

Proof 3: For the proof of correctness we first construct a total order of operations and then show that this order is observed by all processors.

First, label all operations accessing word *w* with <*logical time, processor ID, perform sequence number*>, where logical time is the begin time of the epoch in which the operation performs, processor ID is the ID of the processor performing the operation, and the perform sequence number is the rank of the operation in the $<_c$ order. These labels are unique, since no operations

executed on the same processor share the same sequence number. Labels can be constructed for all operations, since every operation has to perform within an epoch (rule 1) and processor ID and sequence numbers trivially exist for all operations.

The serialization of stores to *w* required in the definition is obtained by sorting operations by their labels. We refer to this order as the coherence order for *w*. To show that all processors observe the stores to perform in that order, consider any two stores X and Y to the same word.

If X and Y share the same logical time, then they must also share the same processor ID, since by Rule 1 stores can only perform in Read-Write epochs and Read-Write epochs do not overlap (Rule 2). The CPU executing the stores observes them performing in program order, which is verified by the *Uniprocessor Ordering* checker. Any other processor can only observe the stores in a later epoch, since a load must be contained in an epoch (Rule 1), which can not overlap the epoch during which the store is performed (Rule 2). The data observed by the processor performing the load is the data in the cache at the end of the Read-Write epoch containing the stores (Rule 3). By Cache Correctness, the cache contains the value of the later store performed, which is also the later store in program order. Since X and Y share the logical timestamp and processor ID, they appear in the coherence order in program order. As all processors observe the operations in this order, operations perform in coherence order with respect to all processors.

If X and Y have different logical time labels, then all processors necessarily observe the same order, since the stores are contained in different epochs and epochs are globally ordered. As the logical timestamps are different, they determine the order in which X and Y appear in coherence order. All processors observe the operation with the smaller logical timestamp to perform first, thus the operations perform in coherence order with respect to all processors.

A.2 Invariants Satisfy Memory Consistency

A system which dynamically verifies all three invariants will obey the consistency model specified in the ordering table. This is true independent of the mechanism used to verify each characteristic. Thus it is possible to replace any number of the proposed mechanisms with others that might be more appropriate for a given system.

Definition 6: *An execution is consistent with respect to a consistency model with a given ordering table if there exists a global order* $<_m$ *such that*

- *for X and Y of type OPx and OPy, it is true that if* $X <_p Y$ *and there exists an ordering constraint between OPx and OPy, then* $X <_m Y$ *and*
- *a load Y receives the value from the most recent of all stores that precede Y in either the global order* $<_m$ *or the program order* $<_p$.

As with coherence, we will prove consistency by first constructing an order $<_m$ and by then showing that it has the required properties. To construct the global order, operations are labeled with *<logical time, processor ID, perform sequence number>*. The perform sequence number is the rank of the operation in the sequence of all operations performed by a given processor. The labels are unique and a label can be constructed for every operation.

We use *Allowable Reordering Correctness* (Section A.1.2) to show the first property. All X and Y with $X <_p Y$ must be performed by the same processor, otherwise there is no $<_p$ relation between them. From reordering correctness we know that X performs before Y and therefore X has a lower perform sequence number then Y. As X performs before Y, it must also have a lower or equal logical time so as to not violate causality. As the processor IDs for both operations are equal, all components of the label of X are less than or equal to the respective components of Y's label, which implies $X <_m Y$.

To obtain the value returned from load Y, we consider two cases. First, if there exists a store that precedes Y in program order, but not in perform order, then both Y and the store must be from the same processor, otherwise there is no $<_p$ relation between them. By *Uniprocessor Ordering Correctness* (Section A.1.1), Y receives the value from the most recent such store X. Since Y performs before X, no store from another processor can perform after X but before Y. Therefore X is the most recent of stores preceding Y in either $<_m$ or $<_p$. Second, if no store exists that precedes Y in program order but not in perform order, then only stores that perform before Y have to be considered. The order of this sequence of stores is the same as the serialized sequence of stores to *w* used to show *Cache Coherence Correctness*. Coherence verification (Section A.1.3) ensures that all processors observe these stores to perform in the same order. Thus for any processor, Y receives the value of the most recent of these stores in coherence order for *w.* This is also the most recent store to *w* in $<_m$, because both orders use identical labels and all stores to *w* are contained in both $<_m$ and the coherence order for *w.*

Thus for both cases Y receives the value of the most recent store that precedes Y in either $<_m$ or $<_p$.