

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

Albert Meixner  
Dept. of Computer Science  
Duke University  
albert@cs.duke.edu

Daniel J. Sorin  
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 experimentally evaluate its performance using full-system simulation of commercial workloads.*

## 1. Introduction

Computer system availability is crucial for the multi-threaded (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 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. Section 4. contains a sketch of the framework correctness proof. In Section 5. we describe a checker design for each invariant and give a SPARCv9 based implementation of DVMC. Section 6. introduces the experimental methodology used to evaluate DVMC. We present and analyze our results in Section 7.. Section 8. compares our work with prior work on dynamic verification.

## 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 multi-threaded 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  $Y$ .

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.

Table 1. **Processor Consistency**

1 <sup>st</sup> \ 2 <sup>nd</sup>	Load	Store
Load	true	true
Store	false	true

## 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 ( $<_m$ ).

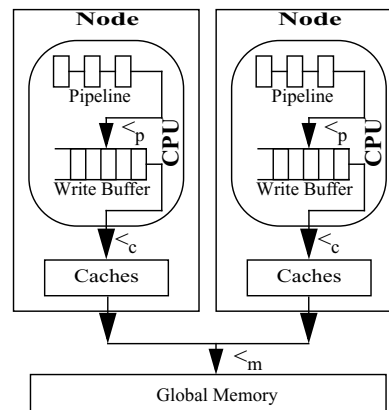


Figure 1. **Operation Orderings in the System**

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 outline a proof that these three invariants ensure memory consistency in Section 4..

**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. DVMC further requires that the system observes the Single-Writer/Multiple-Reader (SWMR) property. This requirement is stronger than coherence, but virtually all coherence protocols use SWMR to ensure coherence. Relaxed consistency models do not strictly require coherence, but all shared-memory systems of which we are aware (including those made by Intel, Sun, IBM, AMD, and HP) are based on a coherent memory system independent of the consistency models that they implement. We do not consider systems without coherent memory 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. Proof Outline

Due to space constraints, we cannot include a formal proof that DVMC’s three invariants ensure consistency, but we provide the proof in a separate technical report [17] and present a sketch of the argument here.

A key observation for understanding DVMC is that in a system with the SWMR property (i.e., virtually all current cache coherent systems) a memory operation performs globally as soon as it accesses the highest level of the local cache hierarchy. Therefore the global ordering of operations from a given processor is identical to the cache access order at that processor. Thus, we can dynamically verify the consistency model ordering requirement in Definition 1 by checking that reorderings between program order and cache access order are valid and that the SWMR property was not violated. The former is ensured by the Allowable Reordering invariant, while the latter is guaranteed by the Cache Coherence invariant.

To check that the data propagation requirement of Definition 1 is satisfied, we need to determine if the value for a given load matches that of the most recent eligible store. We again leverage the SWMR property, which guarantees that the value currently in a cache was written by the most recent globally performed store to that address under the assumption of correct value propagation. The Cache Coherence invariant ensures that the SWMR property holds and also guarantees that the data in the cache matches that written by the store. Finally, the Uniprocessor Ordering invariant ensures that a load will receive the correct value when it is preceded in program order by a store that accesses the same address and performs after the load.

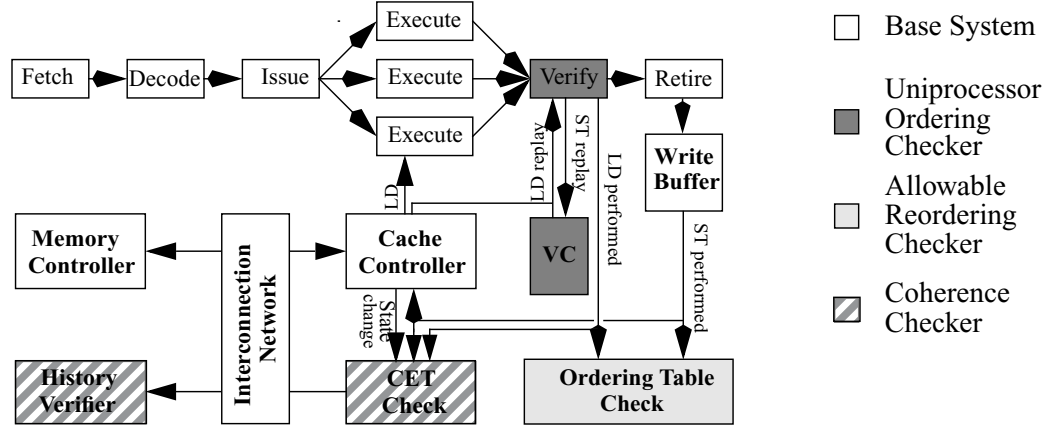


Figure 2. **Simplified pipeline for DVMC.** Single node shown. Several structures (memory, caches, ...) omitted for clarity.

Table 2. **Total Store Order**

	2 <sup>nd</sup>	
1 <sup>st</sup> \	Load	Store
Load	true	true
Store	false	true

Table 3. **Partial Store Order**

	2 <sup>nd</sup>		
1 <sup>st</sup> \	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**

	2 <sup>nd</sup>		
1 <sup>st</sup> \	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

## 5. 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 Order (RMO). TSO is a variant of Processor

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

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 sec-

tion describes the three verification mechanisms that were added to the system, as shown in Figure 2.

### 5.1. 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 violation 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.

### 5.2. 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  $\text{seqX} > \max\{OP_y\}$  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.

### 5.3. Cache Coherence Checker

Static verification of *Cache Coherence* is a well-studied problem [19,20] and more recently methods

Table 6. 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

have been proposed for dynamic verification of coherence [6, 25]. Although *any* coherence verification mechanism is sufficient for DVMC, we reuse the one introduced as part of DVSC [16], which supports both snooping and directory protocols and scales well to larger systems. A detailed description can be found in our earlier work, but we present a brief sketch here.

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. For every epoch at a processor, it sends an *inform message* containing epoch start and end times as well as block value checksums to one of the history verifiers co-located with each memory controller. Each history verifier uses the inform messages to check for coherence violations on its assigned blocks.

The implementation of this *Cache Coherence* checker requires a *Cache Epoch Table* (CET) at each cache and a *Memory Epoch Table* (MET) at each memory controller to keep track of the epoch histories. Each verifier also uses a priority queue to sort incoming informs by timestamp before processing.

## 6. Experimental Methodology

We performed our experiments using Simics [13] full-system simulation of 8-node multiprocessors. We configured the cycle-accurate TFSim processor simulator [15] as shown in Table 6, and we adapted it to support timing simulation for the SPARC v9 consistency models TSO, PSO, and RMO, as well as SC. The systems were configured with either a MOSI directory coherence protocol or a MOSI snooping coherence pro-

Table 7. Workloads

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

Table 8. 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
<i>For Directory Protocol</i>	
Network	2D torus, 2.5 GB/s links, unordered
<i>For Snooping Protocol</i>	
Address Network	bcast tree, 2.5 GB/s links, ordered
Data Network	2D torus, 2.5 GB/s links, unordered
<i>Coherence Verification</i>	
Priority Queue	256 entries
Cache Epoch Table	34 bits per line in cache
Memory Epoch Table	48 bits per line in any cache

ocol. 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 8. Timing information was computed using a customized version of the Multifacet GEMS simulator [14].

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 7 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 7 shows the average fraction of 32-bit memory operations executed for each benchmark during our experiments.

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.

## 7. Evaluation

We used simulation to empirically confirm DVMC’s error detection capability and gain insight into its impact

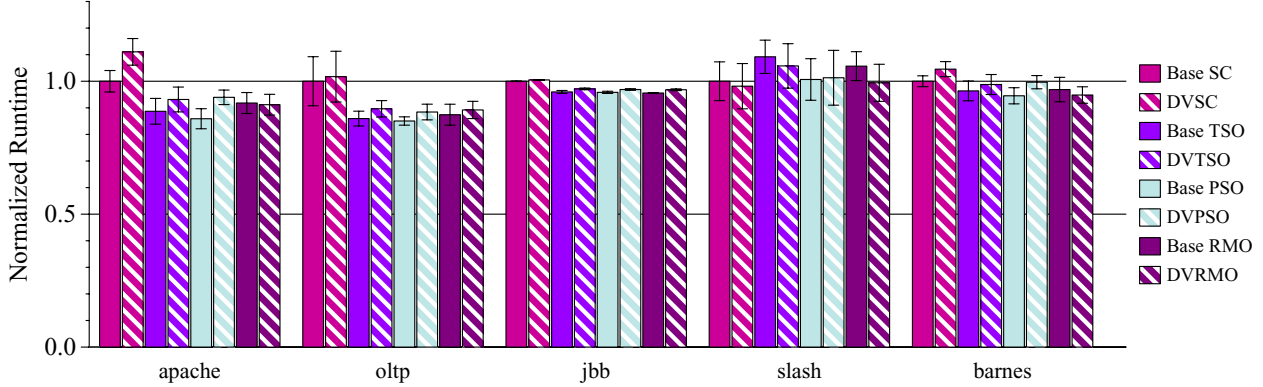


Figure 3. Workload runtimes for directory coherence

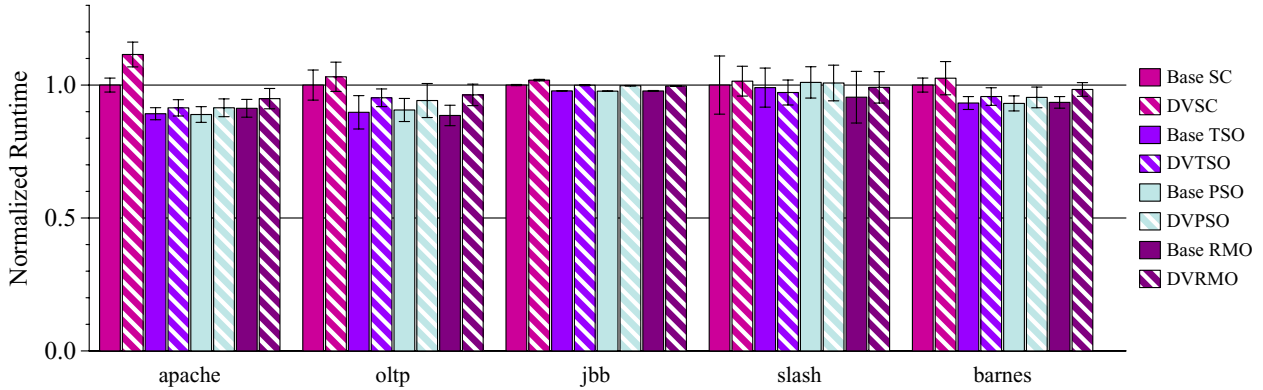


Figure 4. Workload runtimes for snooping coherence

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.

### 7.1. 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.

### 7.2. 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.

**7.2.1. 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

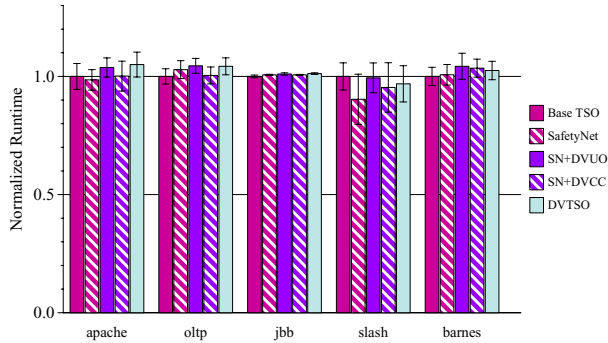


Figure 5. Verification mechanism runtimes

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].

**7.2.2. 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 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

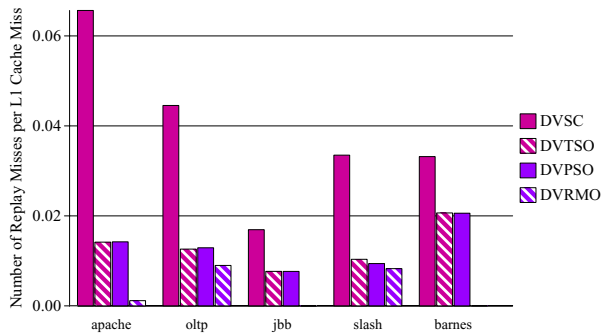


Figure 6. Cache misses during replay

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 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.

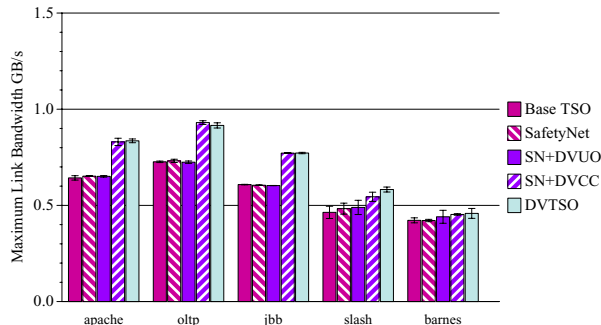


Figure 7. Interconnect traffic

### 7.3. 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.

## 8. 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.

## 9. 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 suggests 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. 1, 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.