Appears in the 32nd Annual International Symposium on Computer Architecture (ISCA)
Madison, Wisconsin, June, 2005

Dynamic Verification of Sequential Consistency

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

In this paper, we develop the first feasibly implementable scheme for end-to-end dynamic verification of multithreaded memory systems. For multithreaded (including multiprocessor) memory systems, end-to-end correctness is defined by its memory consistency model. One such consistency model is sequential consistency (SC), which specifies that all loads and stores appear to execute in a total order that respects program order for each thread. Our design, DVSC-Indirect, performs dynamic verification of SC (DVSC) by dynamically verifying a set of sub-invariants that, when taken together, have been proven equivalent to SC. We evaluate DVSC-Indirect with full-system simulation and commercial workloads. Our results for multiprocessor systems with both directory and snooping cache coherence show that DVSC-Indirect detects all injected errors that affect system correctness (i.e., SC). We show that it uses only a small amount more bandwidth (less than 25%) than an unprotected system and thus can achieve comparable performance when provided with only modest additional link bandwidth.

1. Introduction

The goal of this research is to improve the availability of multithreaded computer systems by *dynamically verifying* that their memory systems are operating correctly. Dynamic verification hardware checks whether a system's execution is correct as it is executing. While *static* verification is beneficial in that it can prove whether a system's implementation satisfies its architectural specification, dynamic verification can detect errors due to physical faults (both transient and permanent), fabrication defects, and design bugs that were not caught by static verification. We explore dynamic verification in this paper, but we note that these two types of verification are complementary and mostly orthogonal. We focus in this paper on errors due to transient physical faults (single event upsets), because they are becoming increasingly problematic, even for combinational logic, due to technological trends [6, 15].

There are two broad approaches to error detection: tailored error detection mechanisms and dynamic verification of invariants. The most obvious approach is the former, in which we develop error detection mechanisms that are tailored to each specific error model being considered. For example, if we consider the "single-bit stuck-at-x" model for the system bus, then adding a parity bit to the bus is sufficient for error detection. We can then construct error detection schemes that compose targeted error detection mechanisms for each error model at each possible error location in the system. This approach, however, is constrained to only detect those errors that are in the models, and it is by definition incapable of detecting errors due to design bugs (if we knew to target them, they would not exist in the first place). Moreover, adding a large set of targeted mechanisms is cumbersome and requires understanding how the error models interact with each other. Most importantly, certain errors are extremely difficult to detect with localized mechanisms (e.g., reordering of snooping cache coherence requests in one section of broadcast tree).

Unlike tailored error detection mechanisms, dynamic verification is an approach that checks that certain high-level invariants are being enforced. It does not consider specific error models, since these error models only matter insofar as they manifest themselves by affecting the correctness invariants. For example, at the microprocessor level, DIVA dynamically verifies that an aggressive microprocessor core has the same external interface as a simple, provably correct checker core [2]. As another example, for shared memory multiprocessors with snooping cache coherence, schemes have been developed for dynamically verifying that all processors observe the same total order of coherence requests, and all coherence upgrades have corresponding downgrades elsewhere in the system [16].

While DIVA and other existing dynamic verification schemes are good steps in the right direction, they do not address the highest level of *end-to-end* [14] correctness in multithreaded (including multiprocessor) memory systems. The end-to-end correctness of a multithreaded memory system is defined by the archi-

tecture's memory consistency model. In this paper, we focus on sequential consistency (SC), the most straightforward consistency model. SC was defined by Lamport such that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [8]. While SC is the most restrictive consistency model, it is used in commercially available systems [19] and recent advances in improving its performance have enhanced its appeal [5]. Future work will address more relaxed consistency models.

In this paper, we present the first two schemes for dynamic verification of SC (DVSC). Without loss of generality, we assume that our multithreaded system model is a shared memory multiprocessor. Both of our DVSC schemes detect low-level errors that matter (i.e., that affect the system's ability to satisfy SC), and we describe our error model in Section 2.. Moreover, both schemes are largely independent of the system's cache coherence protocol. Both DVSC schemes trigger a system recovery if they detect an error, and both are compatible with all-hardware backward error recovery (BER) schemes, such as SafetyNet [17] or ReVive [13], or even software BER. In Section 3., we briefly sketch the first scheme, called DVSC-Direct, which is a proof-of-concept and intuitive starting design point. DVSC-Direct directly verifies the invariant as specified by Lamport. It dynamically constructs a total order of loads and stores and verifies that each load gets the value of the most recent store to that block in the total order. While DVSC-Direct is conceptually simple, its exorbitant use of interconnection network bandwidth motivates our second scheme for DVSC, called DVSC-Indirect. As described in Section 4., DVSC-Indirect implements DVSC by dynamically verifying sub-invariants that, when composed together, have been proven to be equivalent to SC. This approach leads to a vastly more efficient hardware implementation than DVSC-Direct, and we thus present DVSC-Indirect in full detail as the preferred, feasible design point.

In Section 5., we evaluate both DVSC-Direct and DVSC-Indirect using full-system simulation of multiprocessor systems with dynamically scheduled processors and commercial workloads. Our experimental results for systems with both directory and snooping cache coherence show that both systems detect all injected errors. We demonstrate that DVSC-Indirect uses less than 25% more interconnection network bandwidth than an unprotected system. DVSC-Indirect can achieve performance that is comparable to that of an unprotected system, even when provided with only modest additional interconnection network bandwidth.

The main contributions of this paper are:

- The development of DVSC-Indirect, the first feasibly implementable scheme for DVSC.
- Full-system evaluations of DVSC-Direct and DVSC-Indirect that demonstrate that they both detect all injected errors, and DVSC-Indirect is an efficient, viable design.

2. Error Model

In this section, we discuss the errors that DVSC can detect. In general, by virtue of being an *end-to-end* [14] hardware checking mechanism, DVSC can detect single errors in the memory system and many multiple error scenarios. Protecting the processor is an orthogonal issue that is already handled well by other approaches, such as DIVA [2]. The memory system consists primarily of caches, cache controllers, memories, memory controllers, and the interconnection network. We discuss potential errors in each:

Caches and Memories. Bits in caches, memories, and associated state (e.g., coherence state) can be corrupted by faults and thus lead to erroneous data and state.

Cache/Memory Controllers. Coherence controllers, which are essentially just finite state machines, can have their state or outputs corrupted by faults and thus perform incorrect actions. These erroneous actions include: sending an erroneous message (corrupted or replicated), dropping a message (either incoming or outgoing), and operating incorrectly on the cache and memory state.

Interconnection Network. Within the interconnection network, faults can cause errors such that messages are: corrupted (any field, including the destination), dropped, replicated, misrouted, and reordered (if ordering is supposed to be enforced). We include faults that cause errors in messages that our DVSC schemes send.

3. DVSC with DVSC-Direct

DVSC-Direct takes the most direct approach to DVSC. It dynamically constructs the total order of loads and stores and then verifies that this total order satisfies SC. While conceptually simple, we will show that its implementation costs make it infeasible. We briefly present DVSC-Direct as an intuitive first step and to motivate the more efficient design in DVSC-Indirect, rather than to present it in detail as the preferred approach. A more in-depth description of DVSC-Direct can be found in a technical report [11] (in which DVSC-Direct is referred to by its development name, Clouseau). We assume, without loss of generality, that our multithreaded system model is a shared memory multiprocessor. DVSC-Direct is largely independent of the cache coherence protocol (as is DVSC-Indirect).

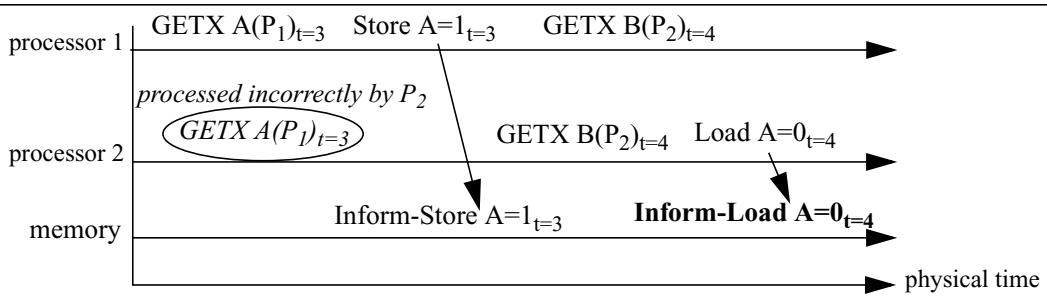


Figure 1. DVSC-Direct in operation in snooping system. GETX $A(P_i)$ denotes the observation of a coherence request made by Processor P_i for Exclusive permission to block A. The subscript is the logical time of the request (we only use major time for simplicity in this example). Initially A=0. The boldface Inform-Load from P_2 reveals that SC was violated, since it should have had the value 1.

3.1. Design and Operation

To construct the total order of memory operations, each processor *informs* the accessed block's home memory node for every load and store. An Inform message consists of: the block address, whether the operation is a load or store (referred to as Inform-Load and Inform-Store), the data value, the *logical time* of the operation, and a short sequence number (for detecting dropped Informs). 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 [7]. We choose two logical time bases—one for snooping and one for directories—based on their ease of implementation. In both logical time bases, time is a 3-tuple: *<major, minor, ProcID>*, similar to Plakal et al. [12]. For a snooping protocol, the major 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 major 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 [17]. To create a total order of all operations, multiple operations by the same processor at the same major time are given monotonically increasing minor times, and operations at the same major and minor times at different processors are arbitrarily ordered by processor ID (since they are causally unrelated). For both types of coherence protocols, there exist numerous other options for logical time bases, but we choose these for simplicity.

Given this logically timestamped stream of Informs, the memory controllers can dynamically verify the total order of loads and stores in a distributed fashion. First, though, they sort them in a priority queue called the Verification Window Buffer (VWB), in order to process them in logical time order (they may arrive at the memory controllers out of logical time order). Each memory controller conceptually maintains shadow copies of each block in its Verification Memory, and it updates its Verification Memory based on the incoming Informs. An Inform-Store writes the value of the Verification Memory block. For an Inform-Load, the memory controller compares the value of the Inform-Load to that of the Verification Memory block and, if they are not equal, declares a violation of SC and triggers system recovery. This algorithm verifies SC, not just cache coherence, despite distributing the verification of different block addresses across the home memory controllers for those blocks. The verifications of these different blocks are not independent, since the logical time base orders accesses to different blocks; thus, distributing the verification of this total order does not hinder DVSC-Direct's ability to verify SC. In Figure 1, we illustrate an example of DVSC-Direct detecting an SC violation in a snooping system.

3.2. Implementation Costs

The reason that DVSC-Direct is impractical, despite its conceptual simplicity, is the interconnection network bandwidth used by Informs. While the Verification Memory also appears to be a major expense, it can be made manageable with caching techniques. However, there is no such silver bullet for Inform bandwidth. A completely unoptimized DVSC-Direct system uses bandwidth like a system without caches. We have explored several sophisticated techniques for compressing Informs, but even with complicated algorithms and hardware the Inform bandwidth is still prohibitive.

We presented DVSC-Direct because it is the most direct approach to implementing DVSC, but we leave the details of its implementation to a technical report [11] so that we can now focus on DVSC-Indirect, a practical, implementable approach to DVSC.

4. DVSC with DVSC-Indirect

Motivated by the implementation challenges of DVSC-Direct, DVSC-Indirect takes a different approach to DVSC. Instead of directly verifying the SC invariant, DVSC-Indirect verifies sub-invariants that have been proven to be equivalent to SC. There are any number of ways to construct SC from sub-invariants, and we choose one particular set of sub-invariants that has already been proven to be equivalent to SC [12]. Intuitively, at a high level, DVSC-Indirect dynamically verifies that cache coherence is performed correctly and that processors correctly interact with the cache coherence protocol in order to implement SC. The advantage of this approach is that we will demonstrate it to be far easier to implement efficiently. The key is that DVSC-Indirect's bandwidth overhead is proportional to the amount of coherence traffic, unlike DVSC-Direct's which is proportional to the far greater number of loads and stores.

As with DVSC-Direct, DVSC-Indirect performs DVSC in a way that is largely independent of the system model and cache coherence model. We also assume again, without loss of generality, that our multithreaded system model is a shared memory multiprocessor. We will present experimental results in Section 5. for DVSC-Indirect with both directory and snooping cache coherence protocols.

4.1. Constructing SC from Sub-Invariants

In the context of *static* verification, Plakal et al. [12] proved that SC is equivalent to one processor invariant (called Fact 1 in their paper) and three lemmas that place restrictions on (a) when loads and stores can occur with respect to per-block *coherence epochs*, and (b) what data values can be communicated between epochs. An epoch is an interval of logical time during which a node has Shared (read-only) or Exclusive (read-write) access to a block of data [12]. Epochs depend strictly on coherence permissions and not on the data. For example, as soon as a processor gets permissions to a block (e.g., by observing its coherence request on the bus in a snooping system), its epoch begins, even if the data response for that request has not arrived yet. Plakal et al. consider a memory operation to be *bound* to a coherence transaction if permission to perform the operation is obtained via that transaction. Fact 1 and the three lemmas are as follows, with informal and more intuitive descriptions afterwards in italics:

Fact 1: Let *LD* be a load from word *w* of block *B* at processor p_i that is bound to transaction *T*. Let *ST* be the last store to word *w* of block *B* by p_i (if any) prior to *LD* in p_i's program order.

(a) If *ST* is also bound to transaction *T*, then the value loaded by *LD* equals the result of *ST*.

(b) Otherwise, the value loaded by *LD* equals the value of word *w* of block *B* received by p_i in response to *T*.

→ *If a processor performs a load, that load either returns the value of a store it just performed, if any, or the value it received for the block otherwise.*

Lemma 1: Exclusive epochs for block *B* do not overlap with either Exclusive or Shared epochs for block *B* in logical time.

→ *Processors cannot have conflicting epochs for the same block at the same (logical) time.*

Lemma 2: (a) Every load/store operation on block *B* at processor p_i is contained in some epoch for *B* at p_i and is bound to the transaction that caused that epoch to start. (b) Furthermore, every store operation on *B* at p_i is contained in some Exclusive epoch for *B* at p_i and is bound to the transaction that caused that epoch to start.

→ *Processors perform loads and stores in appropriate epochs.*

Lemma 3: If block *B* is received by node *N* at the start of epoch $[t_1, t_2)$, then each word *w* of *B* equals the most recent store to *w* prior to t_1 or the initial value in the memory, if there is no store to *w* prior to logical time t_1.

→ *Correct values are passed among processors and memory controllers between epochs.*

Plakal et al. *statically* verified that a specific system satisfied this fact and these lemmas and thus satisfied SC, assuming no physical faults or defects. Their proof is formal and elegant, but without reading it carefully it may not be immediately obvious why these sub-invariants are equivalent to SC. DVSC-Indirect's goal in using these lemmas is different than that of Plakal et al.—DVSC-Indirect seeks to *dynamically* verify that a running system is satisfying SC, which enables DVSC-Indirect to detect errors (due to faults and defects) as well as design bugs that may not have been uncovered during static verification. Because the sub-invariants are proven to be equivalent to SC, dynamically verifying them is equivalent to dynamically verifying SC.

4.2. Dynamically Verifying Sub-Invariants

We now describe how DVSC-Indirect dynamically verifies these sub-invariants. DVSC-Indirect does not have independent mechanisms for dynamically verifying each sub-invariant; it is instead an integrated approach. As one of many possible approaches to dynamically verifying the fact and three lemmas from Section 4.1., DVSC-Indirect combines hardware for dynamically verifying the epoch invariants with DIVA [2] dynamic verification at each processor.

DIVA dynamically verifies the correctness of a complex dynamically scheduled processor by using simple, provably correct checkers at the commit stage. A processor with DIVA presents a simple, in-order abstraction to the memory system, even though the processor itself uses dynamic scheduling and speculation. DIVA does not appreciably slow down performance, because the checkers can keep up with the superscalar core by using it as a prefetcher and branch predictor. DVSC-Indirect relies on DIVA to dynamically verify Fact 1 and parts of Lemmas 2 and 3. Any mechanism to guarantee that memory operations are performed in program order is sufficient for this purpose. We chose DIVA, because it is a well-known design. Other than DIVA, DVSC-Indirect does not require modification to the processor core.

DVSC-Indirect dynamically verifies the epoch invariants—epochs do not conflict and data is transferred correctly between epochs—with two mechanisms. We describe them abstractly here and then present some of the implementation details in Section 4.3.. 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. This check helps to verify Lemma 2. Second, whenever an epoch for a block ends at a cache, the cache controller sends the block address and epoch information, including epoch end time, in an Inform-Epoch message (along with a short sequence number to detect dropped Inform-Epochs) to the home memory controller for that block. Epochs can end either as a result of a coherence request—another node's Request-ReadOnly (if epoch is Exclusive), another node's Request-ReadWrite, or this node's Writeback-Exclusive—or as a result of silently evicting a Shared block from its cache. Thus, Inform-Epoch traffic is proportional to coherence traffic plus some extra traffic for otherwise silent evictions of Shared blocks, instead of being proportional to load/store traffic as in DVSC-Direct. Because the coherence protocol operates independently of DVSC-Indirect, 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 any other epochs (Lemma 1), and (b) the correct block data is transferred from epoch to epoch (Lemma 3 and Fact 1b). The memory controller performs these checks using per-block epoch information it keeps in its directory-like Memory Epoch Table (MET).

Fact 1a. Fact 1a degenerates to uniprocessor memory ordering, i.e., a load by a processor gets the value of the most recent store by that processor to the same address. We use DIVA to dynamically verify Fact 1a.

Fact 1b. Processor p_i can only perform LD if it has a Shared or Exclusive epoch at that time, and this epoch starts with transaction *T*. If no previous store by p_i was bound to *T*, then the value loaded by LD has to have been received in response to *T*. DVSC-Indirect dynamically verifies that this value was correctly received by comparing the block data of LD's subsequent Inform-Epoch to the block data that was sent (i.e., the block data in the MET). DVSC-Indirect dynamically verifies that p_i loads the value within an epoch by checking that it has an epoch for that block and that the data is available.

Lemma 1. For every Exclusive Inform-Epoch that arrives, the memory controller checks its MET to determine if this Exclusive epoch overlaps with other Shared or Exclusive epochs. Similarly, for every Shared Inform-Epoch that arrives, the memory controller checks for overlap with other Exclusive epochs. If the memory controller detects conflicting epochs, the system has violated Lemma 1 and thus violated SC.

Lemma 2. For every load, the cache controller checks the CET to ensure that there is a Shared or Exclusive epoch for that block. For every store, the cache controller checks the CET to ensure that there is an Exclusive epoch for that block. Since an epoch starts with a transaction and DIVA ensures that memory operations are logically performed in order, DVSC-Indirect thus dynamically verifies that each memory operation is bound to the transaction that caused the epoch to start.

Lemma 3. By dynamically verifying Lemmas 1 and 2, DVSC-Indirect ensures a total order of Exclusive epochs and that all stores occur within these Exclusive epochs. By comparing the data block at the beginning of each epoch (Shared and Exclusive) to the data block at the end of the most recent Exclusive epoch, DVSC-Indirect dynamically verifies that each epoch begins with a block whose words are comprised of the most recent word values or the original word values of the block (if no stores have been performed to these words yet). DIVA ensures that the most recent word values are those of the most recent stores, since DIVA dynamically verifies that stores are logically performed in program order.

By dynamically verifying this fact and these three lemmas, DVSC-Indirect dynamically verifies SC. In Figure 2, we illustrate an example of DVSC-Indirect detecting a violation of Lemma 1 (and thus SC) due to a fault that causes a processor (P_1 in the example) not to

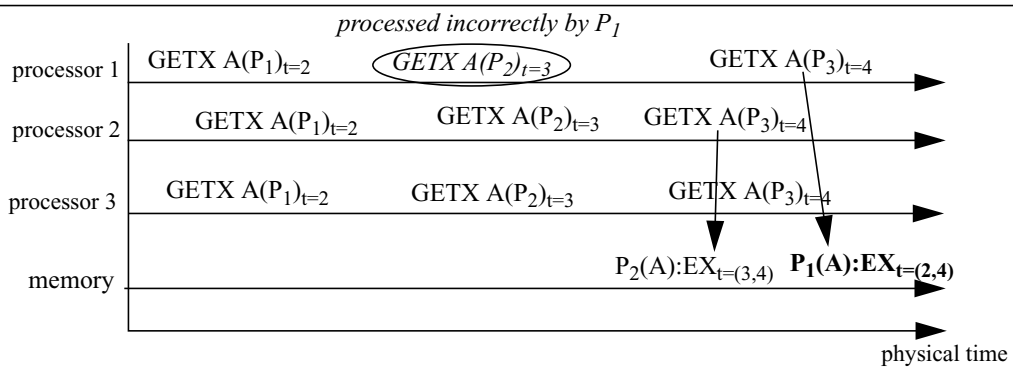


Figure 2. DVSC-Indirect in operation in a snooping system. GETX A(P_i) denotes the observation of a coherence request made by Processor P_i for Exclusive permission to block A. P_i(A):EX denotes an Exclusive Inform-Epoch for block A by P_i. Subscripts denote logical times. Originally, block A is Invalid in all caches. The boldface Inform-Epoch from P_1 reveals that SC was violated, since it overlaps with the previously observed Exclusive Inform-Epoch from P_2 (and violates Lemma 1).

observe another processor's coherence request for Exclusive access.

4.3. Implementation

In this section, we present the implementation of the design in Section 4.2. in more detail.

Cache Controller and CET Operation. Each cache has its own CET, which is physically separate from the cache to avoid slowing cache accesses. There is one CET entry for each cache line, but each entry is only 34 bits. Each CET entry includes: the type of epoch (1 bit to denote Shared or Exclusive); 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). DVSC-Indirect adds 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 uncorrectable. An alternative design would use an error detecting code (EDC), but that would require a backward error recovery scheme that could tolerate this error model (and SafetyNet, which we use in this paper, does not tolerate this error model). 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; logical time and data block at beginning of epoch; and logical time and, if Exclusive, data block at end of epoch (a Shared epoch will have the same data block at the end as at the beginning, and this is ensured by use of ECC on caches).

Memory Controller and MET Operation. The memory controllers receive a stream of Inform-Epochs. To simplify the verification process, we require that memory controllers process Inform-Epochs in the logical time order of epoch start times. Thus, incoming Inform-Epochs are sorted by a priority queue that is very similar to the VWB used by DVSC-Direct. The MET at each memory controller, which can be physically collocated with the directory, maintains the following state per block for which it is the home node: latest end time of any Shared epoch (16 bits), latest end time of any Exclusive epoch (16 bits), and data block at end of latest Exclusive epoch (hashed to 16 bits and initialized from main memory). For every Inform-Epoch it processes, it checks for errors and then updates this state if necessary. To check for epoch overlap errors, it compares the start time of the Inform-Epoch with the latest end times of Shared and Exclusive epochs in the MET. If a Shared Inform-Epoch's start time is earlier than the latest Exclusive epoch's end time, this is an error. Similarly, if an Exclusive Inform-Epoch's start time is earlier than either the latest Shared or Exclusive epoch's end time, this is also an error. To check for epoch data 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 Exclusive epoch. If they are not equal, this is an error. Similar to the caches, DVSC-Indirect adds 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-Exclusive; otherwise, silent corruptions of memory state would be uncorrectable.

Logical Time. As with DVSC-Direct, DVSC-Indirect requires the use of a logical time base. Similar to DVSC-Direct, any number of logical time bases will work, just as long as they observe causality. DVSC-Indirect uses the same logical time bases as DVSC-Direct for systems with snooping and directory cache coherence. Unlike DVSC-Direct, however, DVSC-Indirect

only needs to explicitly timestamp coherence events instead of the more numerous loads and stores, so it only uses the major logical time (not a 3-tuple, as in DVSC-Direct). Ties in logical time can be broken arbitrarily by the VWB (e.g., by arrival order) since there is no causal ordering between events at the same logical time.

One implementation challenge is the need to represent logical times with a reasonably small number of bits while avoiding wraparound problems. One upper bound on timestamp size is that a cache controller cannot wait so long to send an Inform-Epoch that the backward error recovery (BER) mechanism would not be able to recover to a pre-error state if that Inform-Epoch revealed a violation of SC. By keeping the number of bits in a logical time reasonably 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. DVSC-Indirect's method of scrubbing old logical times is to remember to check that an epoch time is not going to wrap around. DVSC-Indirect remembers 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 Shared epochs and a single processor ID ($\log_2$[number of processors] bits) for tracking an open Exclusive epoch. Whenever there is an open epoch, the MET entry does not need the last Shared/Exclusive 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). DVSC-Indirect scrubs 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. A *universal* hashing function that reduces a data block down to n bits has a 2^{-n} probability of *aliasing*, which is mapping two different data blocks to the same data block. For DVSC-Indirect, we use CRC-16, a simple but not universal hash function to hash data blocks down to 16 bits. Aliasing 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, one 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 has a probability of 1/65535 of false negatives for all other blocks.

I/O. Memory consistency models, such as SC, do not consider I/O accesses. However, commercial systems often specify consistency models that include I/O requests. Future work will address the issue of how to incorporate I/O into the DVSC framework. For now, one simple solution for handling DMA transfers from I/O space into memory space is to create a special I/O epoch for each transfer; thus, the data block value written by the I/O device will be correctly propagated from epoch to epoch for DVSC-Indirect. DVSC-Indirect currently does not consider I/O accesses that do not modify the memory space.

4.4. Summary of DVSC-Indirect

DVSC-Indirect performs DVSC while overcoming the interconnection network bandwidth problem associated with DVSC-Direct. We show in Section 5. that DVSC-Indirect's bandwidth usage is far less than that of DVSC-Direct. DVSC-Indirect still has costs, though, including: the need for DIVA (or another microprocessor-level dynamic verification scheme), a CET at each cache hierarchy, an MET and VWB at each memory controller, a timestamp scrubbing FIFO at each cache and memory controller, and ECC on caches and memories (same as DVSC-Direct). While DVSC-Indirect adds several storage structures, none of them are large or complicated. Given the modest interconnection network bandwidth and hardware costs of DVSC-Indirect, we believe that the error detection benefits of DVSC far outweigh these costs.

5. Evaluation

The goals of this evaluation are to determine the error coverage, performance impact, and implementa-

TABLE 1. Processor Parameters

Pipeline Stages	fetch (3),decode (4),execute,retire (3)
Pipeline Width	4
Branch Predictor	YAGS
Scheduling Window	64 entries
Reorder Buffer	128 entries
Physical Registers	224 integer, 192 floating point
DIVA	timing not modeled in detail

tion costs of our implementable design, DVSC-Indirect. For comparison, we evaluate the bandwidth usages of both DVSC-Indirect and DVSC-Direct, since this is the primary difference between the two.

5.1. Methodology

We simulate an 8-node multiprocessor target system with the Simics full-system, multiprocessor, functional simulator [9], and we extend Simics with a memory hierarchy simulator to compute execution times. Each node consists of a processor, two levels of cache, cache controller, some portion of the shared memory, memory controller, support for backward error recovery with SafetyNet [17], and a network interface.

Simics. Simics is a system-level architectural simulator developed by Virtutech AB. We use Simics/sun4u, which simulates Sun Microsystems's SPARC V9 platform architecture (e.g., used for Sun E6000s) in sufficient detail to boot unmodified Solaris 8. Simics is a functional simulator only, but it provides an interface to support detailed timing simulation of the processors and the memory system.

Processor Model. We use TFSim [10], a timing-accurate simulator of a dynamically scheduled microprocessor. This processor implements SC, but it speculates in order to achieve performance closer to that of more relaxed memory models. Table 1 provides the details of the processor that we model. We do not model DIVA's timing in detail.

Memory Model. We have implemented a cycle-accurate memory hierarchy simulator that captures all state transitions (including transient states) in the cache and memory controllers. We have configured it to be able to model both a MOSI directory and a MOSI snooping cache coherence protocol. We model the interconnection network and contention within it. Table 2 presents the design parameters of our target memory systems.

Backward Error Recovery. Our memory system simulator models the details of SafetyNet checkpoint/recovery [17], although other hardware or even software BER schemes could be used. We use the same checkpoint log buffer size (512 kbytes) at each cache and memory controller and the same latencies as Sorin et al. [17]. With a checkpoint interval of 8000 logical cycles for a directory system (10000 for snooping) and four outstanding checkpoints, SafetyNet can recover to a checkpoint of the system that is earlier than any error detected by DVSC-Indirect or DVSC-Direct.

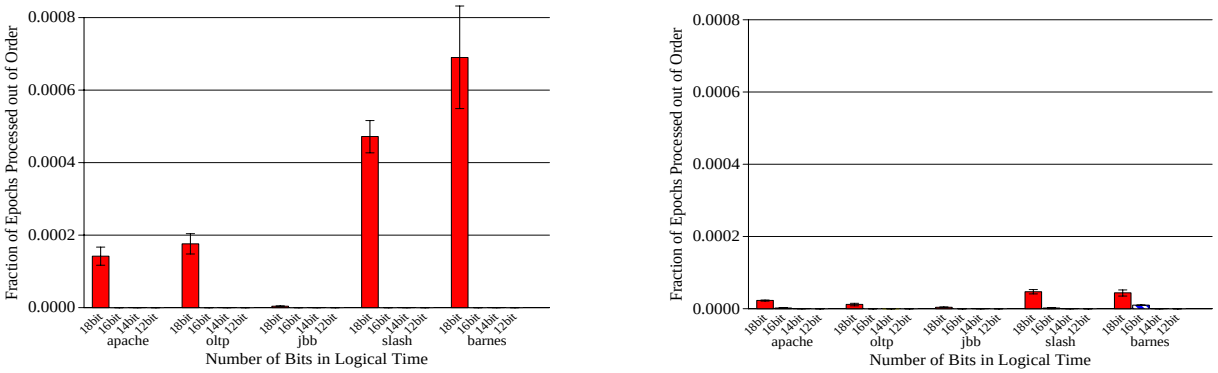
TABLE 2. Memory System Parameters

L1 Cache (I and D)	32 KB, 4-way set associative
L2 Cache	1 MB, 4-way set-associative
Memory	2 GB, 64 byte blocks
For Directory Protocol	
Interconnection Network	2-dimensional torus, 2.5 GBytes/s links, no ordering
Logical Time Base	loosely synch. physical clock
For Snooping Protocol	
Address (request) Network	broadcast tree, 2.5 GBytes/s links, totally ordered
Data (response) Network	2-dimensional torus, 2.5 GBytes/s links, no ordering
Logical Time Base	number of snooping requests processed thus far
Backward Error Recovery (SafetyNet)	
Checkpoint Log Buffer	512 kbytes each
Checkpoint Interval	8000 log. cycles for directory, 10000 log. cycles for snooping
For DVSC-Direct	
VWB	1024 entries
Verif. Memory Cache	4K entries,4-way set assoc.
For DVSC-Indirect	
VWB	256 entries
Cache Epoch Table	one entry per line in cache: each entry is 34 bits
Memory Epoch Table	one entry per line in memory: each entry is 48 bits
Log. Time Scrubbing FIFO	128 entries
Logical Time Size	16 bits

Benchmarks. Commercial applications are an important workload for high availability multithreaded systems. As such, we evaluate our system with four commercial applications from the Wisconsin Commercial Workload Suite and one scientific application. These workloads are described briefly in Table 3 and in more detail by Alameldeen et al. [1]. To handle the runtime variability inherent in commercial workloads, we run each simulation three times with small pseudo-random perturbations. Our experimental results show mean result values as well as error bars that represent one standard deviation below and above the mean.

5.2. Experiment #1: Error Coverage

In this experiment, we seek to demonstrate the ability of DVSC, both DVSC-Direct and DVSC-Indirect, to



(a) Directory System

(b) Snooping System

Figure 3. Out-of-order processing of Inform-Epochs to the same block, as a function of logical time size (in number of bits). None of these reorderings led to a false positive.

TABLE 3. Wisconsin Commercial Workload Suite and a SPLASH benchmark

OLTP: Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 1 GB 10-warehouse database on five raw disks and an additional dedicated database log disk. There are 8 simulated users per processor. We warm up for 10,000 transactions, and we run for 100.
Java Server: SPECjbb2000 is a server-side java benchmark that models a 3-tier system with driver threads. We used Sun's HotSpot 1.4.0 Server JVM. Our experiments use 24 threads and 24 warehouses (~500 MB of data). We warm up for 100,000 transactions, and we run for 10,000.
Static Web Server: We use Apache 1.3.19 (`www.apache.org`) for SPARC/Solaris 8, configured for pthread locks and minimal logging. We use SURGE to generate web requests. We use a repository of 2,000 files (totalling ~50 MB). There are 10 simulated users per processor. We warm up for ~80,000 requests, and we run for 1,000.
Dynamic Web Server: Slashcode is based on the dynamic web message posting system `slashdot.com`. We use Slashcode 2.0, Apache 1.3.20, and Apache's `mod_perl` 1.25 module for the web server. MySQL 3.23.39 is the database engine. The database is a snapshot of `slashcode.com` with ~3,000 messages. A multithreaded driver simulates browsing and posting for 3 users per processor. We warm up for 240 transactions, and we run for 20.
Scientific Application: We use *barnes-hut* from the SPLASH-2 suite [18], with the 16K body input set. We measure from the start of the parallel phase.

detect low-level errors. We thus injected single errors into the simulated system, including:

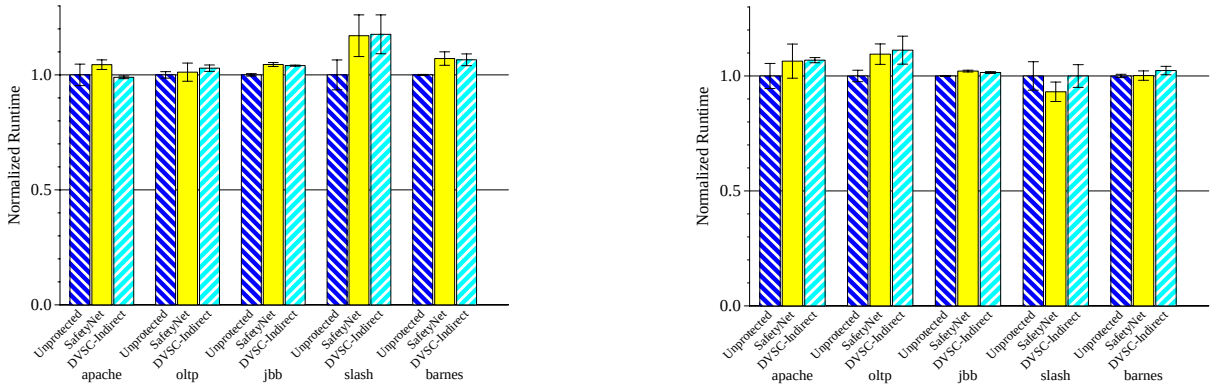
- corrupted, dropped, misrouted, reordered, and duplicated messages (for all kinds of messages)
- corrupted cache and memory blocks, including associated block state

We did not inject errors into the processor core, since DIVA has already been shown to detect and correct processor errors and to provide an in-order abstraction to the memory system [2].

Our experimental results, for both snooping and directory systems, showed that both DVSC-Direct and DVSC-Indirect detected all injected errors. For DVSC-Indirect, the only possibilities of false negatives are if (a) an error in cache or memory state is undetectable by the ECC, or (b) hashing the block values leads to aliasing that masks an error. None of these scenarios occurred in our experiments. For DVSC-Indirect, the only possibility of false positives is if (a) the VWB is not large enough to prevent out-of-order processing of Inform-Epochs to the same block, and (b) at least one of the reordered Inform-Epochs to the same block is for an Exclusive epoch. In Figure 3a (3b), for a directory (snooping) system, we plot the fraction of Inform-Epochs to the same block that are processed out of order as a function of the logical time size, since the number of bits in a logical time determines the frequency of logical time scrubbing and is related to potential reorderings. We observe that, for all logical time sizes, there is a miniscule fraction (often zero) of Inform-Epochs to the same block that are processed out of order; moreover, *none* of these reorderings leads to a false positive since zero reorderings involved Exclusive epochs. Intuitively, this result is not surprising, since the only scenario in which reordering to the same block can practically occur is when a very long Shared epoch wholly contains another very short Shared epoch.

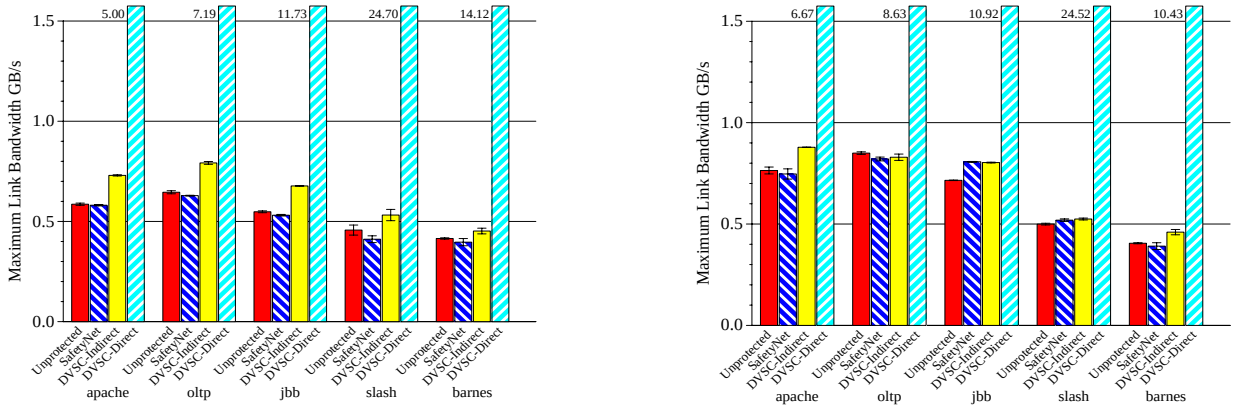
5.3. Experiment #2: Performance

In this set of experiments, we seek to determine the performance impact of DVSC-Indirect, as compared to a system without any fault protection (i.e., no DVSC-Indirect and no SafetyNet). We compare these systems in an error-free scenario. The primary causes of *poten-*



(a) Directory System (b) Snooping System

Figure 4. Error-Free performance of DVSC-Indirect compared to unprotected system



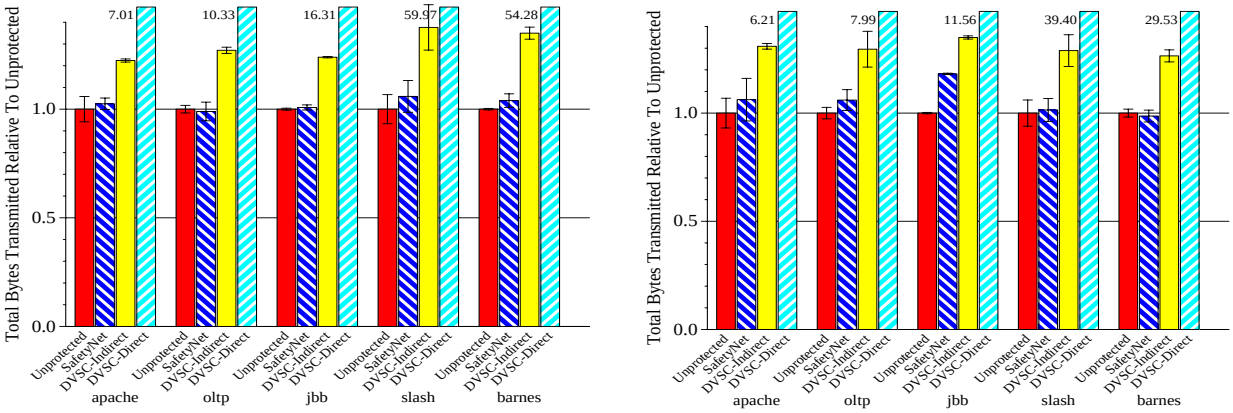
(a) Directory System (b) Snooping System

Figure 5. Bandwidth on most-utilized link in interconnection network

tial slowdown due to DVSC-Indirect are SafetyNet and increased interconnection network congestion due to Inform-Epochs. In Figure 4a, we plot the normalized runtime for three systems: unprotected directory, directory with SafetyNet, and directory with DVSC-Indirect (and SafetyNet). Figure 4b plots the results of the same experiment for the snooping systems. We observe that, in both systems, performance degradation due to DVSC-Indirect is minimal and usually nearly equivalent to that of just SafetyNet. The mean results for some of the runtimes suggest implausible situations (e.g., a system with SafetyNet outperforming an unprotected system), but the error bars show that these discrepancies are not significant. While previous work has shown that SafetyNet does not incur a significant performance penalty [17], it is an interesting result that Inform-Epoch traffic does not significantly degrade performance even for the modest interconnection network link bandwidth provided by these systems. The bandwidth results in Section 5.4. reveal that DVSC-Direct would require a system to provide significantly more link bandwidth than the 2.5 GB/sec assumed here to avoid performance degradation due to Inform congestion.

5.4. Experiment #3: Bandwidth Usage

The interconnection network bandwidth usage of DVSC-Indirect is critical. In fact, it is DVSC-Direct's costly usage of bandwidth that inspired DVSC-Indirect. In this experiment, we compare the link bandwidth usage on the most-utilized link of the interconnection network for: an unprotected system, DVSC-Indirect, and DVSC-Direct (with aggressive compression of Informs). For snooping, which has two interconnection networks (address and data), the most-utilized link is always in the data network. Unlike the other system models, the experiments for DVSC-Direct use interconnection network links with unbounded available bandwidth, since 2.5 GB/s is not sufficient. Figure 5a (5b) shows the results for the directory (snooping) system, normalized to the unprotected system. The results show that DVSC-Direct uses a whopping 8-53 times more bandwidth than an unprotected system. However, for DVSC-Indirect, the directory system incurs only 8-25%



(a) Directory System

(b) Snooping System

Figure 6. Total interconnection network traffic relative to unprotected system

more bandwidth than an unprotected system, while snooping only uses between 0-15% more bandwidth. In snooping, the Inform traffic spreads out more on the lightly loaded links. For further insight into this effect, we studied the total traffic communicated by the entire interconnection network (address plus data for snooping). In Figure 6a (6b), we compare the total number of bytes transferred in the entire interconnection network for directory (snooping) systems. The results show that the snooping system with DVSC-Indirect communicates 26-30% more traffic than the unprotected snooping system. While the link bandwidth experiment only studied mean bandwidths, we are assured that peak instantaneous bandwidths are not a major problem for DVSC-Indirect since otherwise its performance in Section 5.3. would be much worse than that of the unprotected system.

The increase in the number of messages sent with DVSC-Indirect is larger than the increase in consumed bandwidth, because Informs are small. DVSC-Indirect generates 44-66% more messages for directories and 33-38% more messages for snooping than the respective unprotected system. In systems that cannot handle the larger number of messages, multiple Informs can easily be bundled into a single message, since Informs are not latency sensitive.

One other concern we had with respect to interconnection network bandwidth was DVSC-Indirect's sensitivity to the logical time size (i.e., number of bits). With larger times, each Inform-Epoch becomes slightly larger and uses more bandwidth. However, larger times also reduce the need for timestamp scrubbing, since timestamp wraparound occurs less frequently; thus, larger times lead to somewhat fewer Inform-Open-Epochs. Results (not shown) reveal that varying the logical time size between 12 and 18 bits has a negligible effect. This result is positive in that it shows that DVSC-Indirect's design is robust with respect to this parameter.

5.5. Discussion

We have experimentally demonstrated that DVSC-Indirect (a) performs DVSC and thus detects errors that affect sequential consistency, (b) does not significantly degrade performance, and (c) requires less than 25% more interconnection network bandwidth than an unprotected system. Thus, the only remaining obstacle to DVSC-Indirect's adoption is its hardware cost and complexity. DVSC-Indirect requires DIVA. If DIVA was added solely for DVSC-Indirect's purposes, this would be a questionable cost, but DIVA has been shown to be a valuable and relatively inexpensive scheme for microprocessor fault tolerance. DVSC-Indirect also requires several storage structures. The Verification Window Buffers (VWBs), which order Inform-Epochs, are small—at 1024 entries, we observe no false positives due to reorderings. The Cache Epoch Tables (CETs), which hold epoch state at each cache, have as many entries as each cache, but each entry is small (34 bits). The Memory Epoch Tables (METs) have one entry per block of memory, but once again each entry is small (48 bits). In the presented configuration, the total size of each CET is 68 KB, which is about 2.5 times the size of the cache tag array, and every memory node has a 102 KB MET. The FIFOs for avoiding timestamp wraparound at the cache and memory controllers have only 128 entries. None of these structures is more complicated or requires more bandwidth or ports than pre-existing storage structures, such as the directory.

6. Related Work

There has been a variety of research in dynamic verification. At the single-threaded uniprocessor level,

DIVA uses a simple, provably correct checker processor core to dynamically verify an aggressive speculative processor core [2]. At the multithreaded level, Sorin et al. dynamically verify end-to-end invariants of the cache coherence protocol and interconnection network [16], but they stop short of verifying memory consistency. Cantin et al. propose a scheme to dynamically verify snooping cache coherence protocols [4]. This scheme requires manual construction of the checker protocol and significant extra bandwidth for validation. Cain and Lipasti propose an algorithm based on vector clocks for dynamically verifying SC, but they leave for future work the issue of implementation of the algorithm [3].

7. Conclusions

We have demonstrated how to use DVSC as an end-to-end mechanism to detect errors in multithreaded memory systems. We first presented an intuitive, proof-of-concept DVSC scheme called DVSC-Direct that implements the SC invariant directly. Its drawback is the substantial amount of additional bandwidth usage that this scheme incurs, even after optimizing it with compression techniques. This drawback motivated our second scheme, DVSC-Indirect, which dynamically verifies a set of invariants that taken together have been proven to be equivalent to SC. DVSC-Indirect's approach enables it to use far less bandwidth and thus makes it an attractive option for comprehensive error detection in multithreaded memory systems.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant CCR-0309164, the National Aeronautics and Space Administration under Grant NNG04GQ06G, a Duke Warren Faculty Scholarship, and donations from Intel Corporation. We thank Anne Condon, Mark Hill, Alan Hu, Alvy Lebeck, Jaidev Patwardhan, Ravi Rajwar, Raimund Seidel, David Wood, and the Duke Architecture Reading Group for helpful discussions and comments on this work. We thank David Becker, Tong Li, Carl Mauer, and Min Xu for simulator assistance.

References

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

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

[3] 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.

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

[5] M. D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, 31(8):28–34, Aug. 1998.

[6] International Technology Roadmap for Semiconductors, 2003.

[7] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[8] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, Sept. 1979.

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

[10] 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.

[11] A. Meixner and D. J. Sorin. Clouseau: Probabilistic Dynamic Verification of Multithreaded Memory Systems. Tech. Report 2004-2, Department of Electrical and Computer Engineering, Duke University, Sept. 2004.

[12] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proc. of the Tenth ACM Symp. on Parallel Algorithms and Architectures*, pages 67–76, June 1998.

[13] 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 International Symposium on Computer Architecture*, pages 111–122, May 2002.

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

[15] P. Shivakumar et al. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proc. of the International Conference on Dependable Systems and Networks*, June 2002.

[16] D. J. Sorin, M. D. Hill, and D. A. Wood. Dynamic Verification of System-Wide Multiprocessor Invariants. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 281–290, June 2003.

[17] D. J. Sorin, M.M.K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, pages 123–134, May 2002.

[18] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[19] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.