Appears in the 13th International Symposium on High-Performance Computer Architecture
Phoenix, Arizona, February, 2007

Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures

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

To provide high dependability in a multithreaded system despite hardware faults, the system must detect and correct errors in its shared memory system. Recent research has explored dynamic checking of cache coherence as a comprehensive approach to memory system error detection. However, existing coherence checkers are costly to implement, incur high interconnection network traffic overhead, and do not scale well. In this paper, we describe the Token Coherence Signature Checker (TCSC), which provides comprehensive, low-cost, scalable coherence checking by maintaining signatures that represent recent histories of coherence events at all nodes (cache and memory controllers). Periodically, these signatures are sent to a verifier to determine if an error occurred. TCSC has a small constant hardware cost per node, independent of cache and memory size and the number of nodes. TCSC's interconnect bandwidth overhead has a constant upper bound and never exceeds 7% in our experiments. TCSC has negligible impact on system performance.

1. Introduction

Two trends motivate increased interest in fault tolerance for multithreaded shared-memory computer architectures. First, multithreaded systems—including traditional multiprocessors, chip multiprocessors, and simultaneously multithreaded processors—have come to dominate the commodity computing market. Second, the industrial roadmap [7] and recent research [17] forecast increases in hardware error rates due to decreasing transistor sizes and voltages. For example, smaller devices are more susceptible to having their charges disrupted by alpha particles or cosmic radiation [21].

Many researchers have developed effective fault tolerance measures for microprocessor cores, using techniques such as redundant multithreading [16, 15, 20] and DIVA [2]. However, to provide fault tolerance in a multithreaded system, the machine must also be able to detect and correct errors in its shared memory system, including errors in the cache coherence protocol. Whereas we can efficiently detect errors in data storage and transmission using error codes, it is far more difficult to ensure the correct execution of a complex, distributed coherence protocol with multiple interacting controllers. To provide comprehensive, end-to-end error detection, recent research has explored online (dynamic) checking of cache coherence. A coherence checker can either operate stand-alone [5,4] or as an integral part of an online memory consistency checker [12, 13] that also detects errors in the interactions between the memory system and the processor cores. Once a coherence checker detects an error, the system can recover to a pre-fault state using one of several existing recovery mechanisms [19, 14]. Coherence checking is a powerful error detection mechanism, but existing coherence checkers are costly to implement, introduce high interconnection network traffic overhead, and do not scale well to large systems. These costs and limitations preclude their use in low-cost commodity systems.

In this work, we develop the *Token Coherence Signature Checker (TCSC)*, which is a low-cost, scalable alternative to prior cache coherence checkers. It can be used by itself to detect memory system errors, or it can be used as part of a memory consistency checker [12, 13]. With TCSC, every cache and memory controller maintains a signature that represents its recent history of cache coherence events. Periodically, these signatures are gathered at a verifier which determines if an error has occurred. The cost advantages of signature-based error detection come at the expense of an arbitrarily small (but non-zero) probability of undetected errors.

This paper makes three main contributions:

- *TCSC is the first signature-based scheme that **completely** checks cache coherence and can detect all types of coherence errors with arbitrarily high probability. The use of signatures significantly lowers hardware costs and interconnection network traffic compared to previous coherence checkers.*

- *TCSC is the first coherence checker that scales to large systems. TCSC has a constant hardware cost per memory and cache controller that is independent of cache and memory size and the number of nodes in the system. TCSC's interconnection network bandwidth overhead has a constant upper bound and never exceeds 7% in our experiments.*
- *TCSC applies to both snooping and directory protocols, and it is the first checker of any kind for Token Coherence [11].*

In Section 2 we introduce TCSC in the context of online checking of *Token Coherence* [11]. In Section 3 we show how the TCSC mechanism can be applied to any invalidation-based snooping or directory coherence protocol by reinterpreting the protocol in terms of Token Coherence. We discuss implementation issues in Section 4. Section 5 analyzes TCSC's error detection capabilities. Section 6 evaluates the hardware costs and interconnection network bandwidth overhead of TCSC. In Section 7 we compare TCSC to related work in coherence checking. We conclude in Section 8.

2. Token Coherence Signature

The abstract idea of TCSC is to compute two signatures at every *node* (i.e., every memory controller and cache controller) for each block the node has held. One signature represents the history of cache coherence states, and the other represents the history of data values. Every node periodically sends its signatures to a verifier that can then determine if at any point in time any block was in conflicting coherence states or if data did not propagate correctly. For ease of explanation, we start by checking Token Coherence [11], which employs a counting scheme to ensure coherence rather than the discrete states in traditional snooping and directory protocols. The use of a countable quantity allows us to express coherence states in terms of a simple mathematical equation and naturally leads to a formula for a coherence state signature. We first develop a simple *per-block* state signature (Section 2.1), and then we extend it to encompass *all* blocks of memory (Section 2.2). Lastly, we show how to use a nearly identical signature to check data propagation (Section 2.3).

2.1. Coherence State Signature for Single Block

Token Coherence (TC) [11] is a low latency cache coherence protocol for unordered interconnects. Like traditional snooping and directory protocols, TC enforces the single-writer/multiple-reader property. However, instead of block states and transition rules, TC uses the following four invariants to coordinate cache accesses:

(1) At all times, each block has T tokens in the system, one of which is the owner token.

(2) A processor can write a block if it holds all T tokens for that block.

(3) A processor can read a block if it holds at least one token for that block.

(4) If a coherence message contains the owner token it must contain data.

These invariants have been formally proven to guarantee coherence in the fault-free scenario [3], and their simplicity makes them attractive for online checking. Each cache controller can locally check Invariants 2 and 3 by performing a redundant token check for every load and store. Nodes can also locally check Invariant 4 when receiving coherence messages. However, nodes cannot independently check Invariant 1, because it describes a global property of the system.

Rather than checking Invariant 1 directly, it is equivalent and more efficient to check changes in the token counts (for both owner and non-owner tokens) rather than the absolute number of tokens held. In all further discussion, we use T to represent either the number of owner tokens per block ($T=T_O=1$) or the number of non-owner tokens per block ($T=T_N$), and we replace Invariant 1 with three sub-invariants:

(1a) Initially, there are T tokens for block B in the system.

(1b) A node can never hold less than 0 or more than T tokens for block B.

(1c) If a node sends (receives) N_t tokens for block B at time t, then another node must receive (send) the same number of tokens N_t for block B at the same time t.

Invariant 1c assumes instantaneous transfers of tokens between nodes, although in practice tokens spend non-zero time in transit. To satisfy the invariant despite these latencies, we consider the receiving node to possess the tokens during the entire transmission. To accurately account for the transmission time in its token history, the receiver must know when the tokens were sent. For this purpose, nodes timestamp each token-carrying outgoing message with the time t of sending.

When we mention tokens being sent at a given "time", we are referring to the *logical time* at which this event occurs. For purposes of TCSC, logical time can be any time base that respects causality and one additional constraint: it allows a node to send or receive only a single message per logical time step. We will discuss the details of how we maintain logical time in Section 4.1.

The original Invariant 1 can now be checked by ensuring the three Invariants 1a, 1b, and 1c. Invariant 1a can easily be checked because initially all tokens are owned by the memory controllers. Each node can locally check Invariant 1b. For checking Invariant 1c, each node computes two token count signatures for *B* to record exchanges of owner tokens, $TCS_{token,owner}(B)$, and non-owner tokens, $TCS_{token,non}(B)$, during the checking interval *I*. For brevity, in the remainder of the paper, we use $TCS_{token}(B)$ to refer to both signatures. For each arrival/departure of tokens for block *B*, the node updates $TCS_{token}(B)$ to reflect the number of tokens (N_t) that arrived/departed at time *t*, where *t* is the time at which the tokens were *sent* (for both arrivals and departures) and N_t is positive for arrivals and negative for departures, according to the following equation:

$$TCS_{token}(B) = \sum_{t \in I} N_t \cdot (T+1)^t$$

Periodically, every node sends these two token signatures (for owner and non-owner tokens) to a central verifier. If the verifier determines that the signatures for both owner and non-owner tokens sum to zero, then the number of tokens for *B* in the system must be constant for any time *t* included in the signature. To ensure that *B*'s state was updated correctly, a node does not obtain N_t directly from the message, but instead computes it by comparing the number of tokens held before and after processing the message.

As presented thus far, there are two challenges in implementing this scheme: coordination of signature collection and signature growth. Collecting signatures from all nodes is easiest to coordinate by transmitting signatures at regular intervals. However, because token receivers use the timestamps on incoming messages to update their signatures, a node must not send its signature to the verifier until it has received all messages sent to it before the collection time. For this purpose, we add a grace period after each collection time during which the nodes wait for in-flight messages to arrive. During the grace period, token events that occur after the collection time are recorded in a secondary signature, because they must not affect the signature that will be sent during collection. Once the original signature is sent to the verifier, the new collection time is determined and the secondary signature becomes the new primary. This scheme is guaranteed to be correct only if no message lingers in the interconnect longer than the grace period. Thus, the grace period specifies a fixed time limit for message delivery, and *false positives* (detections of "errors" that did not occur) can occur if delivery takes too long. This is not a major issue for two reasons. First, we expect the checking intervals to be orders of magnitude larger than the average delivery time, and the grace periods can be as long as the checking interval or longer if we use more than one secondary signature. Second, a severe delay in message delivery can legitimately be considered a fault. Any checking scheme that does not limit the maximum message delivery time will also be unable to detect dropped messages, because dropping a message is equivalent to an arbitrarily long delay.

The second implementation challenge is that storage required for the sum computed in $TCS_{token}(B)$ grows by $log_2(T+1)$ bits at every token transfer and quickly becomes very large. Because no lossless compression scheme can *guarantee* a smaller signature or bound the growth to a fixed size, we use hashing to map the original unbounded signature to a smaller, fixed size set of numbers. To be able to check signatures by summing them, the hash of a sum of two signatures must be easy to compute from the sum of the signature's hashes. Fortunately a simple modulo computation suffices:

$$TCS_{\text{token}}(B) = \left[\sum_{t \in I} N_t \cdot (T+1)^t \right] \bmod n$$

With this modification the signature size is now constant, but multiple distinct token histories can potentially result in the same signature and lead to *false negatives* (undetected errors). However, with a sufficiently large *n*, the probability of false negatives can be made arbitrarily small.

2.2. Coherence State Signature for All Blocks

Even with a constant signature size, maintaining and verifying one signature for every cache block is prohibitively expensive. Low-cost verification requires a constant-size, per-node signature that covers *all* blocks. The easiest way to obtain a single signature is to sum up the token signatures for all blocks (i.e., $TCS_{token} = \Sigma_B TCS_{token}(B)$), but such a signature would not detect tokens being accounted to an incorrect block address. Instead, we take advantage of the fact that the logical time base we use allows only a single message to be sent or received per node per logical time step, and the N_t tokens sent or received by a node at time *t* therefore all belong to the same block. Thus, a single signature can be used for multiple blocks, if we check that the signature refers to tokens for the same block for any time *t*. This is done by computing separate signatures for block addresses ($TCS_{addr,owner}$ and $TCS_{addr,non}$) similar to the token signatures, except we replace the token counts (N_t) with block addresses (A_t), and we replace both T_O and T_N with the highest address in the address space, $max\{A\}$. If, due to a fault, received tokens are attributed to a different address than the address for which they were sent, the address signatures will not sum to zero and a coherence error will be detected.

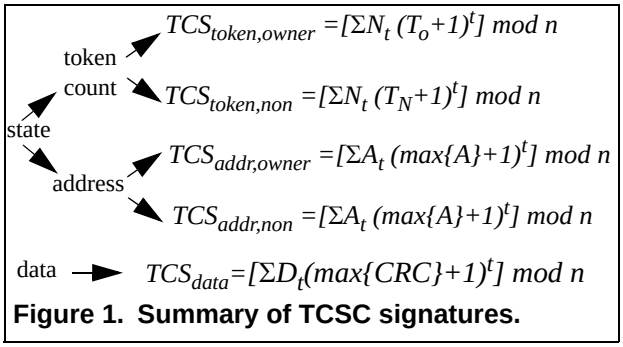

Figure 1. Summary of TCSC signatures.

To provide error detection guarantees (see Section 5), TCSC requires that *max{A}* be smaller than *n*, which could be a problem for systems with very short checksums (small *n*) and large address spaces. In these systems, we can replace A_t with a hash of the address, and we can replace *max{A}* with the maximum hash value. However, today's CPUs typically use fewer than 64 bits for physical addresses and thus a modest 64-bit checksum will be sufficient to avoid address hashing.

2.3. Data Propagation Signature

Up to this point, all of our detection efforts focused on checking coherence states, but to truly check coherence our mechanism must also check data propagation. To clarify, we extend Invariants 2 and 3.

(2') A processor can write a block if it holds all *T* tokens for that block. *Between receiving the T^{th} token and the first write to the block, it must contain data identical to the data after the last write by the previous owner.*

(3') A processor can read a block if it holds at least one token *and holds data identical to the data at the block's current owner.*

These revisions to Invariants 2 and 3 demand that all readers observe the same data values and that modifications made by a writer are passed on to all future readers, i.e., data is stored and propagated correctly. We require EDC on caches and memories to detect corruption during storage. TCSC checks data propagation by computing a *data signature*, TCS_{data}, that is identical to the signatures developed in Sections 2.1.-2.2., except that N_t or A_t is replaced with the CRC checksum of the transmitted data (D_t), and *T* or *max{A}* is replaced by the maximum CRC value (65535 in our implementation).

2.4. Summary and Example of Operation

TCSC maintains 5 signatures at every node, as shown in Figure 1. Two pairs of signatures—token count and address—are used to detect violations of the Single-Writer/Multiple-Reader property and another signature is used to ensure correct data propagation. The former signatures come in pairs to handle the two types of tokens (owner and non-owner) introduced in the TC invariants. Periodic checking of the sums of these signatures allows the verifier to detect violations of any of the TC rules. As we will mathematically show in Section 5, TCSC can detect any single error and a large class of multiple error scenarios.

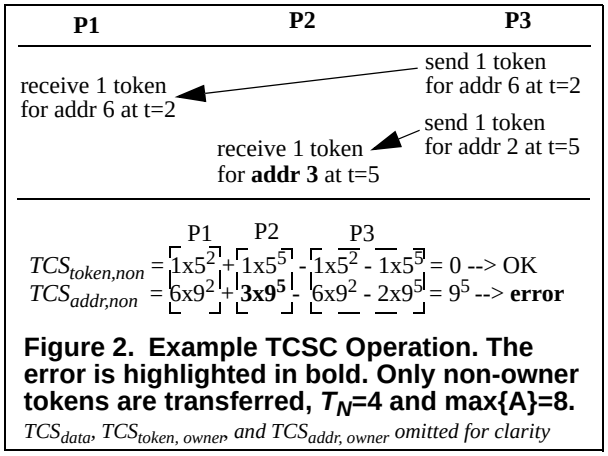

Figure 2. Example TCSC Operation. The error is highlighted in bold. Only non-owner tokens are transferred, T_N=4 and max{A}=8.
TCS_{data}, $TCS_{token,\ owner}$, and $TCS_{addr,\ owner}$ omitted for clarity

We provide an illustrative example of a simple error scenario in Figure 2. It shows how TCSC can detect when tokens arrive for a different address than that for which they were sent, which violates Invariant 1c.

3. Mapping MOSI to Token Coherence

Token Coherence is elegant and might become popular in the future, but it is not yet used in commercial systems. Almost all current systems use some variant of invalidation-based snooping or directory coherence protocols. In these protocols, a block can be in one of 4 distinct states: **M**odified, **O**wned, **S**hared or **I**nvalid. A snooping or directory protocol can be expressed in terms of tokens, if we view all MOSI states as named token counts and interpret state transitions as token transfers. This re-interpretation lets us express the original protocol as a TC protocol without adding redundant state. We do not consider the **E**xclusive state in this work, because it had little impact on the performance of our coherence protocols. Support for the E-state in TC requires a *dirty* flag in addition to the token count as well as some modification to the TC invariants [10] that do not affect the basic operation of TCSC.

Table 1 shows the token values assigned to each state at both the cache and memory controllers. Any state transition will change the token value of a block at a given node, and this change must be offset by another change in the opposite direction occurring at the same

Table 1. Cache and memory states

	Cache		Memory	
	Non-Owner Tokens	**Owner Tokens**	**Non-Owner Tokens**	**Owner Tokens**
M	T_N	1	0	0
O	0	1	T_N-#Sharers	0
S	1	0	T_N-#Sharers	1
I	0	0	T_N	1

time. The implicit tokens consumed and produced by state transitions are used to compute the signatures in the same way as tokens exchanged in TC. As with TC, we use a logical time base, and tokens can be temporarily in-flight in the interconnect.

The implicit tokens represented by the MOSI states are no different from the ones maintained explicitly in Token Coherence. Thus, if the invariants are observed, the system is guaranteed to be coherent. Invariants 2 and 3 are obeyed if no reads are performed in the Invalid state and writes are limited to the Modified state. Cache controllers can locally check these two invariants by redundantly checking cache states during accesses. Invariant 4 can be checked by using the data signature described in Section 2.3 to ensure that no node enters an owner state without receiving data from the previous owner. Invariant 1 is checked by computing the token values of each transition and recording them in the TCS. Violations of Invariant 1 caused by corruption of state information are detected using EDC on states and tags.

Although the basic mapping of MOSI to TC is simple and does not require additional resources, there are three intricacies that complicate this approach. First, most MOSI protocols allow caches to silently evict Shared (read-only) blocks. TC does not support silent evictions, because Invariant 1 requires tracking all tokens at all times. Therefore, in TC, a cache that evicts a Shared block must transfer its implicit token(s) to the home node via a *Put-Shared (PUTS)* cache coherence request. Because TCSC verifies MOSI protocols by mapping them to TC, it also requires explicit PUTS messages for evicted blocks. These additional requests are the primary source of TCSC traffic overhead.

Second, in the *Shared* and *Owned* states, the number of tokens held by the memory controller is determined by the number of sharers for that block. This information is not available in snooping systems and thus must be added to implement TCSC in snooping systems. Directory systems that can remember only a limited number of sharers before falling back to broadcast also need an additional field to store the number of sharers. The storage overhead in directory systems is much smaller, because the address tags are already present and only $log_2(\#nodes)$ bits must be added.

Third, besides the four stable MOSI-states, high-performance coherence implementations allow a (possibly large) number of transient states to handle split transactions and other optimizations. Each of these states must also be assigned a token value and be treated like any stable state with regard to signature computations. We do not present these mappings, because they are highly implementation-specific and our optimized protocol implementations contain close to 40 transient states. However, we can trivially derive the token values of most transient states from stable states. For example, after a *Get-Shared* request is sent for an Invalid (no tokens) block in a directory protocol, that block will be in a transient state until a response from the directory arrives. The transient state also represents zero tokens.

4. Implementation Issues

Signature computation does not require complex structures, occurs off the critical path, is latency tolerant, and needs to be performed only at interconnect speed rather than CPU speed. These factors make the implementation of TCSC simple and cheap, but some care is still needed to avoid unnecessary hardware costs and performance penalties. We describe the implementation of logical time (Section 4.1), the addition of PUTS requests to an existing coherence protocol (Section 4.2), and the implementations of signature computation (Section 4.3) and signature verification (Section 4.4).

4.1. Implementing Logical Time

TCSC requires a discrete time base to order its histories of token transfers and data transfers. We use logical time, which is a time base that is both causal and locally monotonically increasing in physical time. Similar to Lamport's original logical time base [8], all nodes maintain a local clock counter and timestamp all outgoing messages that contain data or transfer access permissions. Clocks are updated according to two rules. First, upon sending or receiving a message, increase the clock by one. Second, if a message is received with a timestamp greater than the local time, set the clock to the message timestamp plus one and consider the message to be received at the updated time. The cost of implementing this time base is the addition of a short (16-bit) timestamp to the message header. TCSC already requires this field in the header for accounting of tokens and therefore providing logical time has no extra cost. TCSC does not timestamp request messages but only

token-carrying messages, because the coherence invariants address only how or when tokens move between nodes, not how or when those movements were initiated. Thus, the time of requests is irrelevant for checking the invariants.

For systems with snooping coherence, we use an optimized logical time scheme in which every node increments its logical time whenever it observes a broadcast coherence request. Token transfers between nodes happen instantaneously, because the sender of a broadcast request and all destinations see the request at the same logical time. This effect makes message timestamps unnecessary, leading to reduced overhead.

4.2. Optimizing Shared Writebacks

TCSC disallows silent evictions of cache blocks. Thus, evictions of Shared blocks—which are generally silent for snooping and directory protocols—now require PUTS transactions that increase interconnect traffic and controller occupancy. However, we minimize this overhead by piggy-backing each PUTS onto a subsequent coherence request to the same home node. A PUTS of block *A* is issued only when block *A* is evicted from the cache, which occurs only after a miss to another block, *B*, that maps to the same cache set. Thus, the PUTS of *A* is immediately followed by a *Get-Shared* (GETS) or *Get-Exclusive* (GETX) coherence request for *B*. Both the PUTS and GET (either GETS or GETX) pertain to the same cache set and thus have a large number of common address bits. Hence, we can piggy-back the PUTS onto the GET in systems in which (a) all bits used to select the block's home node are also used to select the cache set, or (b) all requests are broadcast. Most CPUs use the least significant bits (above the block offset bits) for set selection. Using bits above the set selection bits for home node selection is only necessary for large numbers of nodes or very coarse interleaving of addresses mapping to different nodes, both of which are uncommon. For example, in an 8-node system with a 2MB 4-way cache, any interleaving of 64KB or finer will allow piggy-backing. With piggy-backing, the PUTS does not require a separate message header and needs to hold only the address bits not among the common bits. In our simulated system, this reduces the cost of a PUTS from 8 bytes (4 header+4 address) to 3 bytes (address-8 shared bits) added to the GET.

We can further reduce the bandwidth used by PUTS messages by piggy-backing an *implicit PUTS* onto the GET request. Instead of sending the address of the block that was evicted, we send the cache way of the evicted block and let the directory determine the address. To make this optimization work, the memory controller must remember, for each sharer of a block, in which cache way the block resides. The cache controller, which typically knows the way in which a block will reside before a GET request, augments each GET request with the target cache way using $log_2(\#ways)$ bits per request. Upon reception of a GET request, the memory controller can use the set bits of the full requested address and the cache way to determine which block (if any) was held in that location at the requestor's cache and generate an implicit writeback. The cost for the reduced traffic overhead is the additional memory controller storage for remembering cache way information.

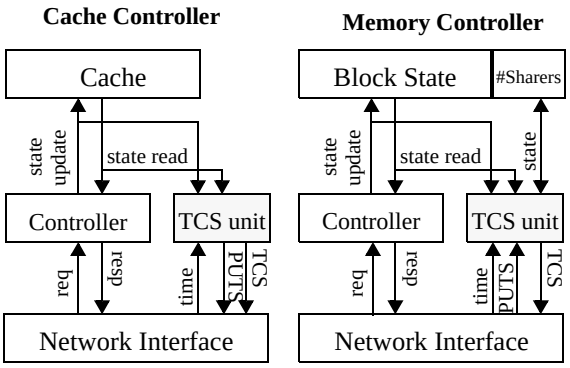

Figure 3. Implementation of TCS computation

4.3. Implementing Signature Computation

Signatures are computed in separate TCS units, as shown in Figure 3, to minimize the impact on the normal operation of the cache and directory controllers. The TCS unit computes signatures based on the logical times and coherence states it receives from the network interface and coherence controller (for either cache or memory), respectively. With the exception of infrequent transmissions of the computed signature to the verifier, there is no feedback from the TCS unit into the rest of the system. Thus, as long as signature computation throughput exceeds the arrival rate of coherence messages, TCS computation does not affect message processing latency or throughput.

The memory controller's TCS unit can also be designed to process PUTS messages (in both snooping and directory systems) in order to reduce contention for the memory controller itself. For snooping systems and directory systems that do not maintain full sharer bitmasks (e.g., sparse directories), our TCSC implementation has already added a table for tracking the number of sharers of each block. The added table is accessed only by the TCS unit, so the TCS unit can process PUTS messages without interfering with the memory controller. For directories that do maintain full sharer bitmasks (and thus know the sharer counts without added tables),

it is generally preferable to just process the PUTS messages at the memory controller. Full directories do not scale and are used only in small systems, in which the extra PUTS contention for the memory controllers is not likely to be a problem.

4.4. Implementing Signature Verification

The signature verifier is a centralized component and could be a potential bottleneck. In small systems, this is not an issue, because signature collections are infrequent and do not cause large amounts of traffic. In large systems, multiple verifiers, interleaved by address range or time interval, can be used to avoid contention for the network links near a central verifier. To further increase scalability, signatures can be aggregated for groups of nodes by adding the signatures locally and sending their sum to the verifier. Aggregation can also be done hierarchically to scale to extremely large systems.

The duration of the verification interval in TCSC is not resource-constrained because all storage requirements and computation times are constant. Verification intervals must be shorter than the recovery period of the checkpoint mechanism used, but can otherwise be chosen freely. In general, short verification intervals will slightly increase bandwidth consumption due to more frequent signature collections and shorten the grace period for delayed messages. Long verification intervals increase the probability of multiple error scenarios.

5. Analysis of Error Detection Capabilities

The signature hashing we use to obtain a constant size signature can lead to *aliasing*, i.e., two distinct token histories mapping to the same signature. Aliasing can cause false negatives (undetected errors), but we can minimize this probability by choosing the signature constants *n* and *T* appropriately. The same arguments apply for the address and data signatures, if we replace *T* with *max{A}* or *max{CRC}*, so we just focus here on the token signatures. In this section we analyze the error coverage provided by TCSC. Computing the exact probability of false negatives would require knowledge about the distribution of errors in the system, which varies greatly from system to system.

5.1. Analysis of Single Error Detection

Because hardware errors in microprocessors are very infrequent, we now make the common assumption of *single error* scenarios, i.e., only a single error occurs within a checking interval. For single errors, we can *eliminate* the possibility of false negatives by careful selection of *n* and *T*.

We discuss the only four single error scenarios possible in TCSC: a transaction with incorrect token count, incorrect time, incorrect address, or incorrect data. *All low-level single errors manifest themselves as one of these scenarios.* Because errors in the address or data are detected using the same equation as errors in token transfers, we discuss only two scenarios in detail: incorrect token count and incorrect time.

For the first scenario, assume that an error causes a node at time *t* to record the arrival or departure of N_t' tokens when N_t tokens were actually transferred, thus violating Invariant 1. In a single error scenario, only the summands for time *t*, $N_t \cdot (T+1)^t$ and $N'_t \cdot (T+1)^t$, differ between the computed signature and the signature for the error-free scenario. Thus, a false negative occurs only if $N_t \cdot (T+1)^t \bmod n = N'_t \cdot (T+1)^t \bmod n$. We can rewrite this as $(N'_t - N_t) \cdot (T+1)^t \bmod n = 0$ or $\exists k : (N'_t - N_t) \cdot (T+1)^t = (k \cdot n)$. A simple divisibility argument shows that this will never be the case if we choose *T* and *n* such that *T+1* and *n* are coprime. The right side of the equation is obviously divisible by *n;* therefore the left side must also be divisible by *n* to satisfy the equation. Because *(T+1)* and *n* are coprime, no factors of $(T+1)^t$ and *n* cancel out and therefore $(N'_t - N_t)$ must be divisible by *n*. Because $(N'_t - N_t) \neq 0$, this is possible only if $|N'_t - N_t| \geq n$. Because no more than *T* tokens can be transferred per time step, $|N'_t - N_t| \leq T$. Therefore, if we choose *T* and *n* such that $T < n$ and $\mathrm{GCD}(T+1, n) = 1$, then TCSC will detect any single incorrect token count. The same argument can also be used to show that TCSC can detect any burst of incorrect token counts up to a length of $\frac{\log n}{\log(T+1)}$ time steps.

The second single error scenario is the shift of a token transfer supposed to occur at time *t* to a different time *t'*. A false negative can occur only if the terms for the correct and incorrect transfer are equal: $N_t \cdot (T+1)^t \bmod n = N_t \cdot (T+1)^{t'} \bmod n$. Because $0 < N_t < n$, we can simplify this equation to $(T+1)^{t-t'} \bmod n = 1$. By setting $c = (T+1)$ and $e = t - t'$, we obtain the equation $c^e \bmod n = 1$. The smallest such *e* is called the *multiplicative order* of *c* modulo *n*. The multiplicative order of *(T+1)* mod *n* therefore determines the smallest value of *(t-t')* that can lead to a false negative, i.e., the maximum time shift that is guaranteed to be detected. If we choose *n* such that the multiplicative order of *(T+1)* mod *n* is larger than the checking intervals, TCSC detects any single delayed transfer.

In our experiments we used $n = 2^{64}$, *T=#processors*, $max\{A\} = 2^{40}$, and $max\{CRC\} = 2^{16}$. All constants are powers of two. Thus, *n* is coprime to *T+1* (for *T>1*),

max{A}+1, and *max{CRC}+1*. We empirically checked for all processor configurations that the multiplicative orders of *T+1*, *max{A}+1*, and *max{CRC}+1* modulo *n* are larger than $2^{16}-1$, the maximum *t* representable by the 16-bit timestamps used.

5.2. Analysis of Multiple Error Detection

When multiple errors occur during a checking interval, there is a non-zero probability that TCSC will not detect them. For any given multiple error scenario (e.g., two corrupted messages), detectability depends on the exact history of token transactions during the checking interval. The exact probability of false negatives depends on the error distribution in the system, but for a large number of uniformly distributed incorrect token transactions it converges to n^{-1}.

5.3. Experimental Validation

In addition to this analytical evaluation of error coverage and error detection latency, we also experimentally tested TCSC's error detection capabilities using the simulation infrastructure and benchmarks described in Section 6. We randomly injected various errors—corrupted, dropped, rerouted and duplicated messages, incorrect cache transitions, corrupted cache state—into the system and continued simulation until the next signature collection was complete. TCSC detected all errors and the detection latency was about half the checking interval, as expected. It is infeasible to experimentally evaluate the probability of undetected errors because, for reasonably large signatures (64-bits or more), undetected errors are so infrequent that they would require extremely long simulation runs to be measurable. We do not experimentally evaluate error detection latency, because it is determined entirely by the frequency at which the signatures are collected.

6. Evaluation

We now evaluate TCSC in terms of its impact on interconnection network traffic (Section 6.1), performance (Section 6.2), and hardware cost (Section 6.3).

6.1. Interconnection Bandwidth Overhead

Interconnection network bandwidth overhead is the most important cost of TCSC, because it affects system cost and performance overhead. We first present a theoretical analysis of the worst-case bandwidth overheads. and then compare them to experimental results.

6.1.1. Worst-Case Analysis

TCSC has a bounded worst-case overhead per coherence transaction that depends on the coherence protocol. A coherence transaction comprises all of the messages required to obtain access to a block and dispose of it later. The minimum costs for obtaining access consist of a GET request and the transfer of the data itself. The cost of disposing a block can include a PUT request or the cost of the Invalidate request received by the sharer and the following acknowledgment. All computations assume the same 64-byte blocks and 8-byte headers used in our simulations.

Token Coherence. TCSC's overhead is caused by the 2-byte timestamp on every token-carrying message (Data, PUT, and Ack). Every coherence transaction involves exactly two token-carrying messages—one to receive the tokens and one to give them away—for a total of 4 bytes. The worst case overhead is therefore 4/(size of smallest transaction). A minimum transaction requires a GET request, data transfer, and PUTS request, totalling 8+72+8=88 bytes in a system without TCSC. Thus, the worst case overhead is 4/88 (4.54%).

Snooping. No timestamps are added to the messages and all bandwidth overhead is caused by PUTS messages. The worst case is a PUTS in every transaction, and the worst-case overhead is *(size of PUTS)/(size of minimum transaction)*. Determining the size of a transaction is problematic because it involves broadcast requests and therefore depends on the network topology. Because PUTS requests are not broadcast and therefore not affected by topology, a physically shared bus with "free" broadcasts maximizes TCSC's snooping overhead and will be used as basis for this computation. A minimum transaction consists of a GET request and the data message, requiring a total of 80 bytes to be transferred in a system without TCSC. A single PUTS requires an 8-byte message and causes a worst-case overhead of 8/80 or 10%. Piggy-backing of PUTS requests decreases the worst-case overhead to 5%.

Directory. Directory protocols require both additional PUTS messages and timestamps on several messages. Consequently they have a worst-case overhead of 12/80 or about 15%, which is larger than the worst case for snooping or TC. Piggy-backing of PUTS requests decreases worst-case overhead to 10%.

TCSC requires additional bandwidth to transmit signatures from the different nodes to the signature verifier. This happens so infrequently that the additional traffic is negligible compared to the overheads described above. Because both of our logical time bases increment time based on message transmissions, a fixed minimum number of messages and bytes is guaranteed to be transmitted between signature collections. Our simulated 8-

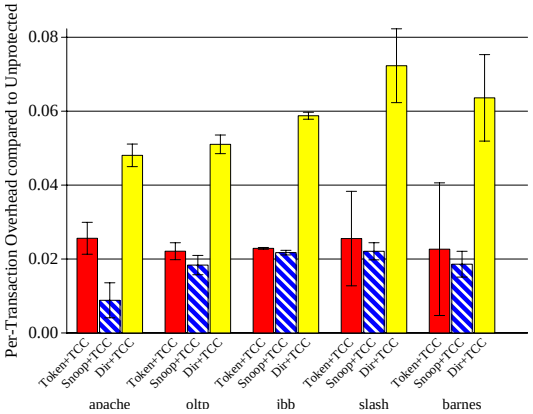

Figure 4. TCSC traffic overhead per coherence transaction

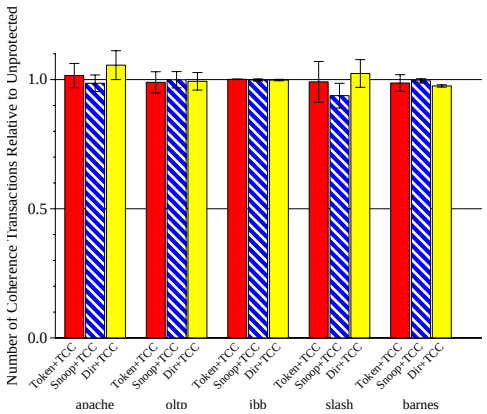

Figure 5. Total number of coherence transactions with TCSC

processor systems (TC, snooping, and directory) have a worst case 0.072% overhead for signature collection.

6.1.2. Experimental Evaluation

This section describes simulation experiments we performed to determine TCSC's bandwidth overheads for various system configurations. We used a modified version of the GEMS simulation toolkit [9] that accurately models the timing of the processors, coherence protocols, etc., of the 8-node multiprocessor systems described in Table 2. Our benchmarks are several commercial applications from the Wisconsin Commercial Workload Suite [1]. To handle the natural variability in multithreaded workloads, we simulated each system configuration ten times. Error bars in our figures represent 95% confidence intervals.

TCSC's bandwidth overhead depends on the sizes of the timestamps and cache blocks used. Because TCSC requires the transmission of a constant number of bytes per coherence transaction, larger blocks reduce the overhead because more data is transferred per transaction. The size of the timestamps is determined by the frequency of signature checking and can be chosen by the designer. We chose a checking interval of 20,000 logical time steps, which allows TCSC to detect errors in time to recover using a backward error recovery mechanism such as SafetyNet [19] or ReVive [14]. It is also long enough to avoid false positives due to waiting for in-flight tokens. All results show the bandwidth overhead of a system with TCSC over the unprotected, baseline protocol ($Overhead=BW_{TCSC}/BW_{base}-1.0$). We present results for three systems: TC with the TokenB performance protocol [11], snooping, and directory

Table 2. Simulated system

System	8-node SMP with Token Coherence, Snooping and Directory
Network Topology	Torus for Token and Directory; Broadcast tree for Snooping
Network Link BW	5GB/s
CPU	2GHz, 4-wide superscalar, out-of-order SPARCv9
Cache	32KB, 4-way L1 I/D; 2MB, 4-way L2 Unified
Memory	2GB
Token Signature	64bit signature (T=#CPUs, $n=2^{64}$) 16bit timestamp, 20000 logical timestep verification interval
Message Size	8B Control Messages; 72B Data Messages (8B Header+64B Payload)

The per-transaction bandwidth overheads shown in Figure 4 are significantly lower than the worst-case numbers. TC is closest to its worst case, with an average of about 2.5%, but it also has the most consistent amount of additional traffic and very low overhead in absolute terms. Snooping stays far below its worst case overhead and overall generates the least amount of additional traffic, with values ranging from about 1% to just over 2%. This indicates that most blocks in the unprotected system are not evicted silently, but either require writebacks or are invalidated by requests from other caches. As expected, directory incurs the highest overhead, but it does not come close to its worst case. Overhead for directory ranges from 5% to 7%, indicating that the kind of minimal length transactions used to compute worst-case overhead are infrequent.

Despite low per-transaction overheads, the total overhead of TCSC could be worse if more coherence transactions were needed to finish a benchmark run. TCSC does not directly impact the number of coherence transactions, but multithreaded workloads are timing-sensitive and even small changes can impact runtime.

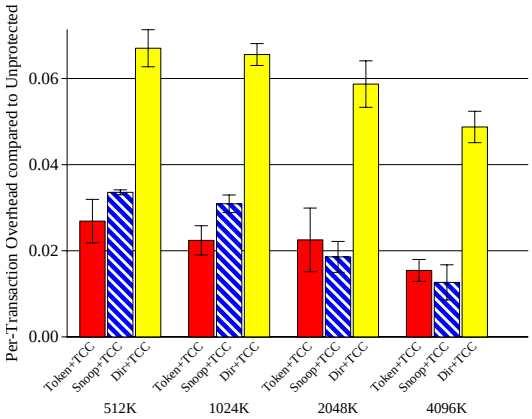

Figure 6. TCSC overhead vs. L2 cache size

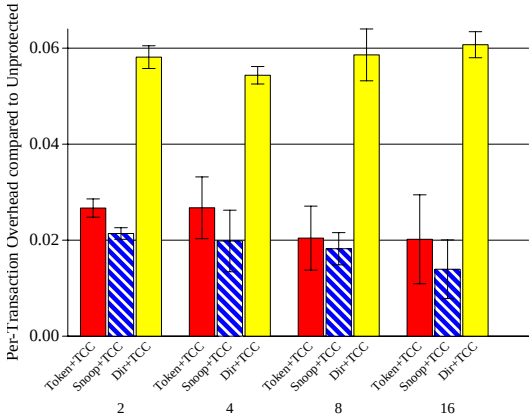

Figure 7. TCSC overhead vs. number of CPUs

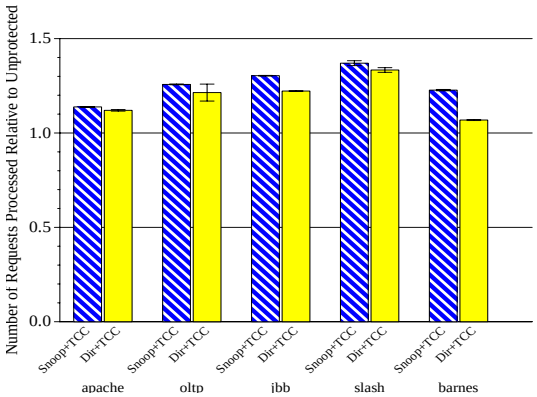

Figure 8. PUTS overhead at memory

Figure 5 shows that TCSC has no statistically significant impact on the number of coherence transactions. TCSC's total bandwidth overhead is the product of the number of transactions and the per-transaction overhead. The standard deviation of this product is larger than the expected TCSC overhead, which would make the overhead difficult to quantify accurately. Therefore, we use the more stable *overhead-per-transaction* metric for all bandwidth overhead figures.

Figure 6 shows the impact of L2 cache size on TCSC's per-transaction bandwidth overhead. As expected, the overheads for snooping and directory decrease with larger caches, because PUTS requests are less frequent due to fewer capacity misses. For TC, where PUTS requests are also necessary without TCSC, there is still a slight downward trend, because the relative number of high-overhead GETS-PUTS transactions compared to other transactions is reduced.

Figure 7 shows TCSC's ability to scale to systems with varying numbers of processors. For snooping and TC, the relative TCSC per-transaction overhead drops when we add more processors to the system, because both of these protocols use broadcast requests that cause overall transaction costs to rise whereas the per-transaction cost for TCSC remains constant. This effect is not present in directory, which has similar overhead for all configurations. Although the scalability experiments were limited to 16-processor systems by the simulation environment, these results along with the worst-case analysis indicate that TCSC can scale to larger systems.

Finally we assess the message processing overhead at the memory controllers due to PUTS messages, assuming for now no offloading of PUTS processing to the TCS unit. Figure 8 shows the request throughput at the memory controllers (normalized to Unprotected) for snooping and directory. TC is not shown because PUTS messages already exist in the base protocol. In our simulated system, even the maximum overhead of 30% did not negatively impact performance (see Section 6.2), because the controller throughput was sufficient to handle the increased load without adding latency. If the overhead causes controller throughput to become a limiting factor, PUTS processing can be offloaded onto the TCS unit as discussed in Section 4.3.

6.2. Performance Overhead

Our cycle-accurate timing experiments (graphs not shown due to space constraints) show no statistically significant ($>1\sigma$) slowdown when running with TCSC. This result is not surprising, because signature computation is off the critical path for both cache and directory controllers and does therefore not impact processing latency or through-put for coherence messages. The added interconnection network bandwidth consumption is too small too severely affect benchmark runtimes.

6.3. Hardware Costs

TCSC requires the addition of one or more signature verifiers, as well as hardware at each memory and cache

controller to compute and store the signatures. The amount of storage required for the 5-part signature depends on the choice of *n*. Practical values for *n* will be in the range of 2^{32} to 2^{128}, thus requiring a total of 20 to 80 bytes of storage per controller. This storage requirement is independent of the cache size.

The computation of the signatures may appear complex because it involves both exponentiation and modulo computation with large numbers. However, careful selection of *n* and *T* (and *max{A}* and *max{CRC}*) enables simple hardware implementations. Within the constraints discussed in Section 5, we can freely choose the constant *n* and the number of tokens per block (as long as the number of tokens per block is greater than or equal to the maximum number of sharers). Therefore, the constant T_N is only required to be greater than or equal to the number of possible sharers. Thus, we can use cheap bit manipulations (bit-shift for exponentiation or bit-wise *AND* for modulus) for *one* of the two computations by picking either *(T+1)* or *n* to be a power of two. Because the two constants must be coprime, we cannot pick both of them to be powers of two. If *n* is not a power of two, we can choose it to be a Mersenne or similar number that allows efficient modulo computations [6]. If *n* is a Mersenne number, signature updates require a total of just one variable shift and two additions. If *(T+1)* is not a power of two, we exploit that *t* is monotonically increasing and we can compute the $(T+1)^t$ term using the result from the previous timestep by a simple multiplication. If *T* is a power of two, a multiplication by *T+1* requires only a constant shift and an addition. Thus, a signature update requires a constant shift, one multiplication, and two additions. Signature verification is done using additions and a comparison against zero, neither of which require complex circuitry.

For snooping protocols and directory protocols that do not maintain full sharer bitmasks, the largest cost is storage space for tracking the number of sharers for each block at the memory controller. In most memory controllers, TCSC requires only $log_2(T+1)$ additional bits per entry in the block state table. Some snooping protocols determine state using wired-OR lines and do not maintain any state table. In these cases, TCSC requires a new lookup table purely for storing sharer counts for each cached block at the memory controller. Alternatively, the sharer count can be maintained by the cache owning the block. This approach adds complexity, because nodes must keep track of the current block owner or broadcast PUTS requests. Systems with full directories or TC already maintain sharer information and do not require additional storage.

7. Related Work in Coherence Checking

Several authors have previously developed implementable checkers for cache coherence. The novelty of TCSC lies in its very low implementation costs, its ability to detect all coherence violations, and its applicability to a wide range of coherence protocols.

Sorin et al. [18] dynamically verify invariants in snooping systems using signatures computed locally and checked periodically at a centralized checker similar to TCSC. These invariants are necessary but not sufficient for coherence. The scheme requires bandwidth for checksum exchanges (<1% overhead) and uses a checksum that is simpler to compute than TCSC. The low overhead is achieved by exploiting properties of snooping protocols, which prevents the mechanism from being applied to other types of protocols, and by sacrificing error coverage. Whereas TCSC can detect errors of any kind with high probability and will detect every single error, Sorin et al.'s scheme is unable to detect certain kinds of errors (e.g., errors caused by operations that are performed correctly but at the wrong time or in the wrong order) and there is no guarantee that even single errors will be detected.

Cantin et al. [5] dynamically verify coherence by replaying transitions between stable states on redundant checker circuits after a transaction completes. This scheme is also limited to snooping protocols and it requires replication of the cache line state information. Thus, the storage requirement is linearly dependent on the cache size, rather than fixed as for TCSC. They add a dedicated snooping bus for verification and do not give bandwidth overhead numbers. To compare it to TCSC, we measured the bandwidth required to replay all requests necessary for full error coverage in the simulated system used for TCSC evaluation. Request replay causes about 20% overhead on average; this is twice the TCSC worst-case overhead and nearly 10 times the observed TCSC overhead for snooping. Unlike TCSC, the verification traffic is broadcast, which limits scalability, and verification of payload data is not addressed.

Our prior work [12, 13] uses coherence checking as part of a mechanism to check memory consistency. DVCC, the coherence checker in that work, causes a 20%-30% increase in interconnect traffic for both directory and snooping protocols in a system comparable to the one used here. Like TCSC, all verification messages are unicast and the payload is verified using checksums. DVCC requires additional verification state for each block and thus storage cost grows with cache size. In our simulated system, storage for DVCC totals about 128KB per cache and 192KB per memory controller.

8. Conclusions

TCSC is a comprehensive error detection mechanism for coherent memory systems, and it has much lower implementation costs than previous schemes. The additional hardware is simple, small in area, and not timing-critical. The worst-case bandwidth overhead is in the 5% to 10% range, but simulation results show that actual overhead is even smaller. The results also show that the bandwidth overhead is significantly less than in existing verification mechanisms: about five to ten times less for a snooping protocol and four to five times less for a directory protocol. No comparable mechanism had previously been proposed for Token Coherence.

These factors make TCSC a compelling option to vastly increase a system's error detection capability without sacrificing cost or performance. The applicability of TCSC is not limited to the hardware coherence mechanisms discussed in this paper, but also includes software DSM systems which often exhibit higher error rates due to less reliable interconnect networks.

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 Fred Bower, Jeff Chase, Carla Ellis, Mark Hill, Alvy Lebeck, Anita Lungu, Milo Martin, and Bogdan Romanescu for their helpful feedback on this research.

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 Int'l Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[3] S. Burckhardt, R. Alur, and M. M. Martin. Verifying Safety of a Token Coherence Implementation by Parametric Compositional Refinement. In *Sixth Int'l Conf. on Verification, Model Checking and Abstract Interpretation*, Jan. 2005.

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

[6] J. Chung and A. Hasan. More Generalized Mersenne Numbers. *Lecture Notes in Computer Science*, 3006:335–347, May 2004.

[7] Int'l Technology Roadmap for Semiconductors, 2003.

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

[9] M. M. Martin et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.

[10] M. M. K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin, Dec. 2003.

[11] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proc. of the 30th Annual Int'l Symposium on Computer Architecture*, June 2003.

[12] 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.

[13] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, June 2006.

[14] 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.

[15] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture*, pages 25–36, June 2000.

[16] 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.

[17] P. Shivakumar et al. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, June 2002.

[18] D. J. Sorin, M. D. Hill, and D. A. Wood. Dynamic Verification of End-to-End Multiprocessor Invariants. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 281–290, June 2003.

[19] D. J. Sorin, M. M. 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 Symposium on Computer Architecture*, pages 123–134, May 2002.

[20] T. N. Vijaykumar, I. Pomeranz, and K. K. Chung. Transient Fault Recovery Using Simultaneous Multithreading. In *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, pages 87–98, May 2002.

[21] J. Ziegler et al. IBM Experiments in Soft Fails in Computer Electronics. *IBM Journal of Research and Development*, 40(1):3–18, Jan. 1996.