Clouseau: Probabilistic Dynamic Verification of Multithreaded Memory Systems

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

[1]Department of Computer Science, Duke University

[2] Department of Electrical and Computer Engineering, Duke University

Abstract

Dynamic verification enables a system to improve its availability by checking that its execution is correct as it is running. While high performance and low power are desirable, correctness—despite hardware faults and subtle design bugs—is most important. For multithreaded systems, memory system correctness is defined by the memory consistency model. Thus, dynamically verifying memory consistency would ensure that the entire memory system is operating correctly.

We present the first implementable design for probabilistic dynamic verification of sequential consistency (pDVSC) in multithreaded systems. The system dynamically creates a total order of memory operations (loads and stores) and verifies that this total order obeys SC. In the theoretical world of systems without resource constraints, DVSC would have to consider the entire total order, but we show how to leverage resource constraints to verify only a sliding window of the total order. While we cannot bound the size of this window and still eliminate all false verifications (false positives or negatives), we can implement probabilistic verification *and make the probability of false verification arbitrarily small.*

We use full-system simulation of a multithreaded system running commercial workloads to evaluate our first implementation of pDVSC, called Clouseau. Clouseau's implementation costs are kept reasonable via extensive compression and caching of the data that is used for dynamic verification. Clouseau, combined with backward error recovery, improves availability by recovering from injected errors. Clouseau adds only negligible performance overhead. While Clouseau adds to system design complexity, we believe this is a small price to pay for improving system availability.

1 Introduction

> *The scaling of the design complexity and the increasing transistor count will greatly increase the potential for failures to occur. In this case, relaxing the requirement for 100% correctness in both transient and permanent failures ... may reduce the cost of manufacturing, verification, and testing.* ["Error-Tolerant Design" Grand Challenge from the 2003 International Technology Roadmap for Semiconductors].

While systems strive for high performance, low power consumption, and low design complexity, system correctness is most important. The definition of correctness depends on the context. For example, in the context of a single-threaded uniprocessor, the DIVA *dynamic verification* scheme [2] continuously checks that the execution of a speculative out-of-order processor core matches

that of a simple, provably correct core. Dynamic verification improves availability by detecting hardware errors (e.g., bit flips) before they can irrevocably corrupt system state. Unlike *static* verification (e.g., model checking), which catches design bugs before fabrication, dynamic verification can detect hardware errors in the field that are due to both physical faults and design bugs. Combining dynamic verification with backward error recovery (BER) further improves availability by allowing the system to recover to a pre-fault, uncorrupted checkpoint. As hardware fault rates rise [6, 21]—due to shrinking transistors, denser wires, and more complicated designs—and society becomes increasingly dependent on computational infrastructure, an architectural approach like dynamic verification is necessary for achieving desired availability. At the architectural level, we can comprehensively address system-wide issues that are beyond the scope of circuit-level solutions, without the performance degradation of software solutions. While software fault rates are also rising, this is an orthogonal problem that we do not address in this research.

Our goal is to dynamically verify multithreaded memory systems—such as SMT, chip multiprocessors, and conventional multiprocessors—since these increasingly prevalent systems are often used as servers. Servers comprise important infrastructure, and thus their availability is important. While previous work has shown how to dynamically verify a single-threaded processor core [2] and certain memory system invariants [22], until now we have not been able to dynamically verify *end-to-end correctness* [20] for a multithreaded memory system. For a multithreaded (or multiprocessor) system, the end-to-end correctness of its memory system is determined by its *memory consistency model*. An architecture's consistency model specifies the allowable interleavings of memory operations (loads and stores) among the threads accessing shared memory. End-to-end verification of memory consistency detects all lower-level errors in the memory system (e.g., flipped bits in cache coherence messages) that affect its correctness, since correctness is defined by the consistency model. In this paper, we focus on sequential consistency (SC), a common and intuitive model that specifies that the memory operations must appear to execute in a total order that respects the program orders for each thread [13]. More relaxed memory models exist, and ongoing research is extending our approach to these models.

Our approach to dynamic verification of SC (DVSC) dynamically constructs a total order of all memory operations using *logical time* (i.e., a time base that reflects causality). The system continuously verifies that this total order satisfies SC. Thus, it checks to make sure that every load obtains the value of the most recent store (in the total order) to that location. While our approach to dynamic verification is conceptually straightforward, implementing it efficiently is the challenge.

In Section 2, we provide background material on SC, including why it is fundamentally impossible to dynamically verify every load for most practical system models. However, we can use *probabilistic dynamic verification* to check almost all loads. By using more hardware resources, we can make the probability of false verification—false positive or false negative—arbitrarily small.

In Section 3, we present an overview of our first implementation of probabilistic DVSC (pDVSC), called Clouseau. Clouseau relies on having threads logically *inform* the home memory controller for each memory operation (load or store), and the memory controllers process these Inform messages to verify SC in a distributed manner. Clouseau is independent from the cache coherence protocol (i.e., it works with snooping, directories, or other types of protocols), and it interfaces to a system-wide checkpoint/recovery mechanism, in order to recover the system to a pre-fault checkpoint if verification detects an error. Section 4 discusses how to optimize the implementation of Clouseau so as to not require undue amounts of extra interconnect bandwidth and memory storage. To optimize Clouseau, we use extensive compression and caching of the data that is used for verification purposes.

In Section 5, we use full-system simulation and commercial workloads to evaluate Clouseau. We show that Clouseau has negligible impact on performance, and we demonstrate that it detects subtle errors that we inject into the system. We also show how our implementation can use varying amounts of hardware to achieve an arbitrarily low probability of false verification. We study several schemes for compressing the interconnection network bandwidth used by Clouseau, including some semantic-based techniques that are particularly effective. Clouseau's hardware represents added cost and complexity, but we believe it is worth it to improve system availability.

In Section 6, we discuss related work in dynamic verification and logical time ordering. In Section 7, we conclude the paper with the insights that we have gained from this work.

2 Dynamic Verification of SC

This section presents the benefits and challenges of DVSC, as well as the idea for our pDVSC design.

2.1 Why DVSC Improves Availability

The correctness of a memory system is defined by its memory consistency model. Being able to detect all violations of SC would enable a system with an SC architecture to detect any lower-level hardware error that could manifest itself as a violation of SC. DVSC improves availability by detecting errors that could otherwise irrevocably corrupt data. Combining DVSC with backward

error recovery (BER), as we do in this work, further improves availability by allowing the system to recover to a pre-fault, uncorrupted checkpoint and resume execution (after possibly reconfiguring, in the case of a hard fault).

Currently, systems detect lower level hardware errors, such as bit flips, with localized mechanisms like error detecting codes. However, instead of trying to detect every type of low level error model, we would prefer an end-to-end test detection mechanism that subsumes the low level error models. Moreover, an end-to-end mechanism can detect error models that (a) were not anticipated, (b) result from subtle design bugs, or (c) are not detectable with strictly localized mechanisms. Consider, for example, a processor that continues to load a block after it has been invalidated from its cache (due to a transient fault in the cache controller). This error model is undetectable with strictly localized error detection mechanisms. We will show later how Clouseau can detect this error.

In this paper, we use DVSC to detect single hardware errors in the memory system and interconnection network that cause a violation of sequential consistency. Errors that do not lead to a violation of consistency (e.g., a flipped bit in a data block that is not accessed again) do not matter! Our targeted errors—which are the result of physical faults or subtle design bugs—include corrupted cache coherence messages, improperly processed messages, and errors in the cache controllers. Our goal is to avoid *false verifications*, which can be divided into *false positives* (i.e., DVSC detects an "error" that did not occur) and *false negatives* (i.e., DVSC fails to detect an error that did occur). While a false positive incurs a performance penalty for needlessly recovering the system to a "pre-fault" state, it is not as dangerous as a false negative. Given a single error scenario, an error in the DVSC checker hardware itself can fortunately only lead to a false positive; a DVSC error cannot cause a false negative unless another error in the system had occurred (the error that DVSC fails to detect), which is a double error scenario. In addition to single errors, DVSC detects multiple errors as long as none of the errors are in the DVSC hardware itself and the errors do not mask each others' effects.

2.2 Why DVSC is Difficult

The memory consistency model, which is specified as part of the architecture (just like the ISA), determines the allowable orderings of memory accesses among different threads or processors. Sequential consistency (SC) is the most intuitive memory consistency model, and it is the model on which we focus in this paper. As defined by Lamport [13], SC requires the *appearance* of a total order of memory operations (i.e., loads and stores) such that (a) the total order respects

the program order of each thread, and (b) every load in the total order obtains the value of the most recent store (in the total order) to the same location.

In the theoretical world of physically unconstrained systems, DVSC is impossible. This is because SC allows any reordering of memory operations if that reordering preserves its two invariants. Thus, in theory, dynamic verification must consider the entire execution before determining if it obeyed SC. For continuous workloads (i.e., workloads without well-defined endpoints, such as web servers or databases), there is no end of the execution and thus no way to verify the execution. Moreover, even for finite workloads (e.g., scientific computations) in which the system could potentially record the entire execution for future analysis, determining if the execution satisfies SC is still NP-complete [11].

2.3 Why Probabilistic DVSC is Feasible

The abstract DVSC model is that of a single checker sequentially processing the sequence of reads and writes performed by all processors during the execution. In theory, this checker has to consider the entire execution (and, even then, the problem is NP-complete). However, in practice, implementable hardware can only reorder memory accesses so much, and so the system can dynamically verify a sliding window of prior execution (and not the entire execution). Implementation constraints enable us to simplify the problem with the following three keys:

(1) While DVSC is NP-complete if given the entire execution, it is only *O(nlogn)*, where *n* is the number of reads/writes in the entire execution, if the checker is provided with the total order of reads and writes for every address [11].

(2) A recent scheme for *static* (offline) verification [24, 18] uses logical time clocks to construct a total order of all system reads and writes. We adapt this scheme for use in DVSC hardware.

(3) A recent technique for *static* verification of SC [5] develops assumptions that limit the size of *n* in the *O(nlogn)* problem. These assumptions constrain the size of the sliding window of reads/writes that needs to be considered at any point. This result does not strictly apply to *dynamic* verification, but we leverage the insight gained from it. In the case of dynamic verification, there is an engineering tradeoff between the verification window size that we choose and the non-zero probability of false verification. We cannot enlarge the verification window to eliminate the probability of false verification without making assumptions about the system model (e.g., interconnection network latency has a fixed bound). Thus, we are fundamentally limited to probabilistic DVSC (pDVSC), i.e., there is some probability that the window will not be big enough.

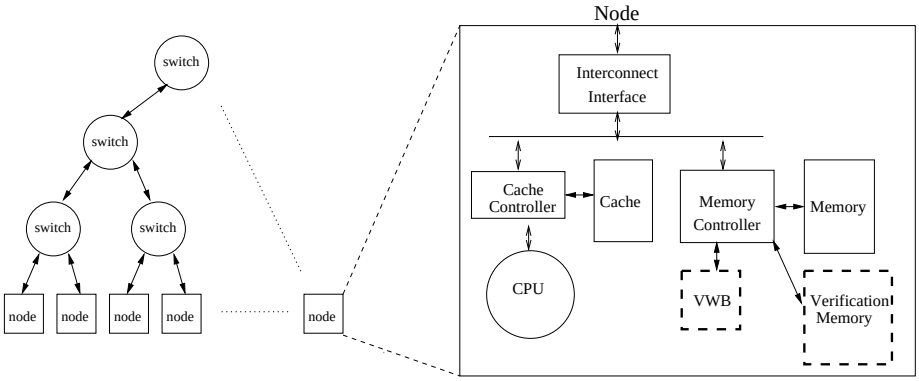


Figure 1. System Model with Address Network. Data/Inform network (2D torus) not shown.

Conclusion. Synthesizing the first two keys reveals that we can use logical clocks to create the total order of reads and writes and reduce the problem complexity from NP-complete to *O(nlogn)*. The third key inspires our strategy for limiting *n*.

3 Unoptimized Clouseau

Based on observations in Section 2.3, we now present Clouseau, our first pDVSC implementation. In this section, for purposes of explanation, we develop an implementation that is correct but not efficient enough to be feasible. Then, in Section 4, we show how to optimize this implementation in order to make its bandwidth and storage costs reasonable.

3.1 System Model

While we could model any multithreaded system with any cache coherence protocol, since pDVSC is generally independent of these issues, we choose a symmetric multiprocessor (SMP) with a MOSI snooping cache coherence for Clouseau. We save a discussion of the impact of different system models for Section 3.6. The system is illustrated in Figure 1 with Clouseau additions highlighted. Nodes in the multiprocessor include a processor, two levels of cache, a portion of the shared memory, and an interface to the Address and Data/Inform interconnection networks. For pDVSC purposes, nodes also contain a verification window buffer (VWB) and Verification Memory which will be discussed later. Nodes communicate via two switched (i.e., not bus) interconnection networks. Address messages (coherence requests) use a logical tree that can implement totally ordered broadcast (which is necessary for snooping coherence, not for pDVSC). Data and Inform messages use a 2-dimensional torus interconnection network with point-to-point ordering (not shown in Figure 1).

Checkpoint/Recovery. Clouseau improves availability by itself, by detecting errors before they corrupt system state, but it can improve it further when used in conjunction with backward error

recovery (BER). For BER, we choose SafetyNet [23], an all-hardware scheme that can hide the long error detection latencies incurred by Clouseau. SafetyNet periodically checkpoints the system state. If Clouseau detects an error, SafetyNet recovers the state to the *recovery point*, the old checkpoint most recently validated error-free. A system-wide checkpoint includes the state of the processor registers, memory values, and coherence permissions. SafetyNet handles I/O with the standard solution of buffering inputs and delaying outputs until validation [7].

3.2 High Level View

Conceptually, the system dynamically constructs a total order of all loads and stores using logical time, a time basis that respects causal dependences [12], and it dynamically verifies that a sliding window of this total order satisfies SC. To construct the total order, processors assign logical time timestamps to all loads and stores that they perform. For each of these operations, they inform the memory controller at the home node of the memory block that was accessed. An Inform message, which is a unicast from the processor to the memory block's home memory, specifies if the operation was a load or store (i.e., Inform-Load or Inform-Store), what data values were read or written, and the logical time of the operation. Given this information, the memory controllers can dynamically verify this total order in a distributed fashion. Each memory controller logically maintains shadow copies of each block in its Verification Memory. It updates its Verification Memory based on the incoming Informs which it buffers in a Verification Window Buffer before processing in logical time order. An Inform-Store updates the value of the Verification Memory block. For an Inform-Load, the memory controller compares its value to that of the Verification Memory block and, if they are not equal, declares a violation of sequential consistency and triggers system recovery. This algorithm verifies sequential consistency, 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[1]; thus, distributing the verification of this total order does not hinder our ability to verify SC.

The verification of the memory operations is delayed in physical time (and pipelined with the active execution), so that the system can process them in logical time order (which, due to skew between physical and logical time, may not be determined until some amount of physical time elapses). This pipelining is shown in Figure 2, in which memory operations (denoted with "X") are

1. Accesses to address X and address Y do not have to be verified by the same memory controller. The relationship between these accesses is reflected in their logical times.

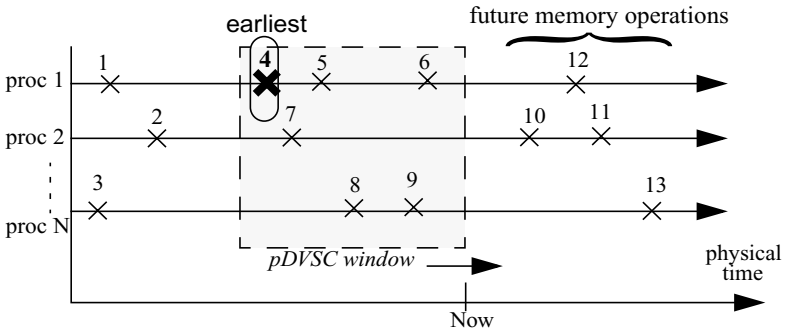


Figure 2. Pipelining pDVSC with Execution. An "X" is a memory operation.

labeled with their logical times. As the verification window widens, it is less likely that a memory operation in the future of the window has an earlier logical time than the earliest memory operation in the window (at logical time 4, in the figure). Processing a memory operation out of logical time order could lead to a false verification, but we can make this probability arbitrarily small by widening the verification window.

All pDVSC implementations must choose a basis of logical time, but there are generally many viable options. A processor in a system with snooping cache coherence can use the number of broadcast cache coherence requests that it has observed as its logical time.[2] Ties between operations at the same logical time (performed by different processors) can be broken in any number of ways, since their ordering does not matter. Clouseau is not limited to snooping systems, though, since logical time bases for other types of systems, including those with directory coherence, exist and are compatible with SafetyNet [23] and pDVSC.

3.3 Clouseau in Action

In Figure 3, we illustrate how Clouseau detects the example violation of SC we mentioned in Section 2.1. In this example, there are two processors (P1 and P2), two memory blocks (A and B), and one memory controller (for simplicity) which is the home for both blocks A and B. Initially, block A has the value 0. P1 then broadcasts a coherence request for exclusive permission to block A (denoted GETX A(P1)) that is the third request seen so far and thus updates the logical times at P1 and P2 to 3 when they observe the request (which can be at different physical times). P2 observes this GETX request, but a transient fault causes it to not invalidate block A from its cache. P1 then stores the value 1 into block A and informs the memory of this store at logical time 3. P2

2. We assume that the Data/Inform interconnection network provides point-to-point ordering for Inform messages, in which case Informs from a given processor will arrive in timestamp order at the VWB. However, if this were not the case, we could simply use a logical time tuple, <major, minor>, in which the minor time is the number of memory operations (and number of Informs) by the local processor since the most recent update of the major time.

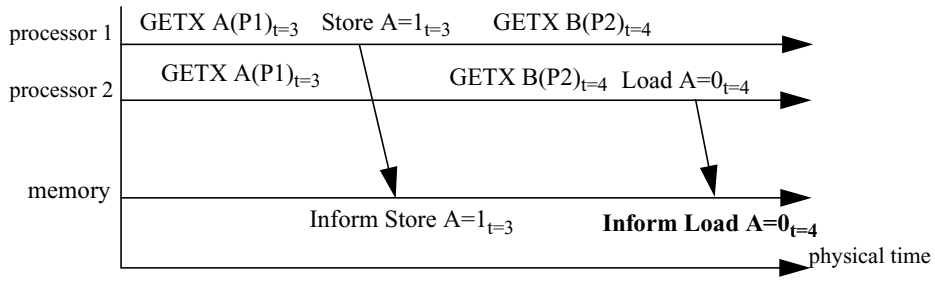


Figure 3. Clouseau Detects SC Violation. GETX A is a coherence request for Exclusive permission to block A. The P# in parentheses is the requesting processor. The subscript is the

then broadcasts a GETX B(P2) request which updates logical time to 4. P2 then erroneously loads the value 0 from block A, and informs the memory of this load at logical time 4. Assume no other accesses to A or B occur. Some time later, memory processes these Informs from its VWB. The Inform-Store sets the Verification Memory block value to 1. Memory then compares the value of the Inform-Load, 0, to the value of the Verification Memory and detects an SC violation.

3.4 Unoptimized Implementation

Clouseau involves modifying and adding hardware in the processors and memory controllers. This added complexity is the price that is paid for improving system availability.

Processors. Processors must send an Inform message to the home node's memory controller for every load and store that they commit. For the optimized implementation in Section 4, we will add hardware to perform sender-side compression of Informs.

Memory Controllers. Each memory controller has a *Verification Window Buffer* (VWB) to buffer incoming Informs, and it processes them in logical timestamp order. The VWBs of all nodes collectively comprise a distributed window of verification (similar to the window of verification used by Condon and Hu [5]). The VWBs must sort entries by timestamp, since Informs do not have to arrive in timestamp order. Each VWB is a hardware priority queue [16] in which the head of the queue is the oldest (i.e., has the smallest logical time) Inform. The VWB's quiescent state is full, and the memory controller processes the head of the VWB whenever an arriving Inform pushes it out.[3] Thus, the size of the VWB determines the size of the window of execution that Clouseau verifies, and enlarging the VWB reduces the probability of false verification.

3. If no Informs were to arrive for a long time, verification would stall and SafetyNet might run out of storage for holding checkpoint state. Thus, if SafetyNet needs to be able to validate an old checkpoint, then the memory controller can process the VWB.

Each memory controller *logically* maintains shadow copies of each block in its *Verification Memory*, and it updates its Verification Memory based on the Informs it processes. In Section 4, we will discuss how to efficiently implement the Verification Memory.

SafetyNet. Clouseau and SafetyNet cooperate in logical time. Clouseau verifies execution in logical time, and SafetyNet takes checkpoints in logical time. Clouseau verifies a given logical time (i.e., allows a checkpoint at that logical time to be made the recovery point in case of a future error detection) once *all* memory controllers have verified all incoming Inform messages up to and including that logical time. Clouseau must also verify the Informs quickly enough to enable SafetyNet to validate checkpoints and discard prior checkpoints.

3.5 Assumptions

Clouseau relies on two assumptions. First, as explained in Section 2.1, we assume that we are trying to detect single hardware errors. An error in Clouseau hardware can only cause a false positive but not a false negative, since another error would have to occur for the error in Clouseau to mask the actual error. Second, we must detect errors in our logical time basis. For Clouseau, this means that the system must dynamically verify the total order of coherence requests, and Clouseau uses a previously developed technique for achieving this [22].

3.6 Impact of Alternate System Models

While Clouseau's design is largely independent from the system model we have assumed, there are a couple of issues that depend on the system model. First, the logical time basis will be different for systems without snooping cache coherence, since there are no broadcast coherence requests to count. Other bases of logical time can be derived, though, such as loosely synchronized physical clocks [23]. This basis of logical time has been shown to be effective for systems that implement directory-based cache coherence. As a proof of concept, we have implemented pDVSC on a system with directory based cache coherence, using loosely synchronized physical clocks for logical time. Due to space limitations, however, we do not discuss it further in this paper.

Second, the processor model could affect the bandwidth required to support Informs. Our evaluation in Section 5 assumes a simple in-order processor model (to enable tractable simulation times). A superscalar (but in-order) processor could increase the rate of Informs and thus use more bandwidth, but we can model this effect with a faster processor. Dynamic scheduling, however, would have no effect on Clouseau, since an Inform is sent only when the memory operation commits. Non-binding prefetches and speculative memory operations have no impact on Clouseau. A more sophisticated processor model might lead to new fault models, but it would not affect the results in this paper.

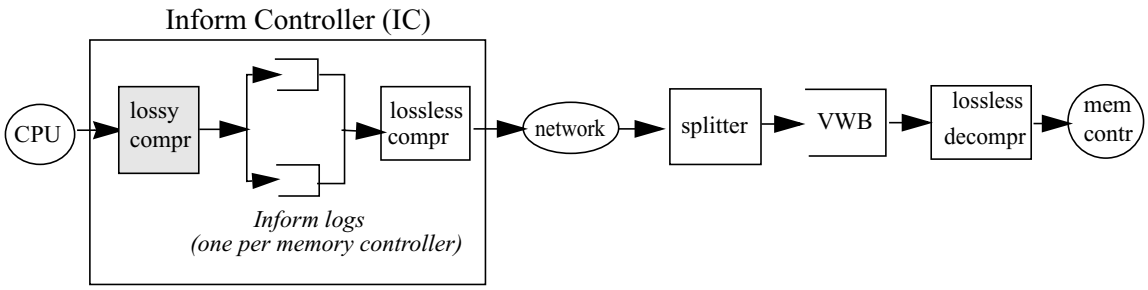


Figure 4. Compression and Decompression. Lossy compression (shaded) is optional.

4 Optimized Implementation of Clouseau

While our high level approach to pDVSC is straightforward, implementing it efficiently poses two primary challenges. First, naively sending an Inform for each load/store would consume far too much interconnection network bandwidth. Second, naively implementing Verification Memory by duplicating main memory would be far too costly. We address these issues in Section 4.1 and Section 4.2, respectively. In Section 4.3, we discuss some implementation details, including I/O.

4.1 Challenge #1: Interconnect Bandwidth

Logically, there is one Inform per memory operation and its size depends on the size of the operation (from 1-64 bytes). Since loads and stores are frequent, a naive implementation would consume interconnection network bandwidth similar to a system without caches, because every load and store would access the memory system! Since communication of Informs is performed off the critical path (i.e., pipelined with the active execution), we can sacrifice latency to reduce its costs. We compress the Inform stream at the sending processors and then uncompress it at the receiving memory controllers. There are a variety of ways to compress Informs, and we explore three of them. The first two are lossless and thus have no impact on the probability of false verification. We designed our compression algorithms such that they can be implemented to handle the high bandwidth stream of loads and stores from the processor. All schemes add an Inform Controller (IC) to each processor, as shown in Figure 4. Each IC has one Inform log per memory controller in the system (i.e., one log per node in the SMP).

Semantic-free lossless compression (referred to as *Lossless*). This scheme losslessly compresses the Inform traffic without exploiting our semantic knowledge of the Informs. The IC takes each memory operation it receives from the processor and inserts it into the Inform log for the corresponding memory controller. The logs enable the IC to package multiple Informs into a single Inform message (up to the maximum message size), including Informs at different logical times.

header (4 bytes)	control 1 (5 bits)	timestamp (5 bits)	control 2 (5 bits)	Inform 1 [address, data]	Inform 2 [address, data]

Figure 5. Example Inform message with two Informs and one timestamp increment.

An Inform message (example shown in Figure 5) has an uncompressed header that includes the logical time of the first Inform in the message. The header is followed by a series of 5-bit control fields for every Inform in the message as well as for every increment in logical time between Informs in the message. Each control field has a bit that distinguishes between an Inform and a timestamp increment. For a timestamp increment, the remaining 4 bits in the control specify the amount of the increment. For an Inform, the remaining 4 bits of the control field specify whether it is an Inform-Load or Inform-Store and what kind of compression is used on the Inform's data. There are three types of compression used: none, variable length, and variable length with history (discussed next). Following the control fields are the Inform fields, which include the address and data for each Inform. The address is always compressed using variable length compression with history, and data is compressed as specified in the control field.

The variable length compression technique, which is based on work by Fenwick [9], creates a compressed word that is composed of 8-bit chunks. The most significant bit of each chunk specifies whether another non-zero chunk exists that is more significant than the given chunk. The other 7 bits are original bits from the uncompressed word. This scheme works well since many data values are small positive integers. To handle small negative integers, the scheme converts negative numbers to positive, which requires an extra sign bit in the least significant chunk of each word. To efficiently compress addresses and data that is not small integers, we can augment variable length compression with history, similar to Farrens and Park [8]. Each Inform log maintains one small (8-entry) history table for addresses and one for data. While the details of this scheme are beyond the scope of this paper, the general idea is that we can use variable length encoding on the difference between the address/data to be compressed and the history table entry that is closest to it (i.e., this difference will look like a small integer, which is easy to compress). The Inform must also carry the index into the history table for the entry against which it is being compared. If history based compression achieves poor compression (less than a factor of 2), the IC inserts the uncompressed data into the history table and replaces the least recently used entry. History based compression requires that the memory controllers, which perform Inform decompression, maintain history tables and update them with the same algorithm as the processors, which is simple to do because they use the same LRU replacement policy[4]. Since neither compression scheme is guaranteed to

produce compressed data that is smaller than the original (particularly for 1-byte and 2-byte accesses), *Lossless* can choose to use no compression for data in these situations. Addresses always use variable length compression with history, since this scheme performs well due to address locality.

Keeping the logs fuller facilitates better packaging of Informs and thus better amortization of message header overhead. The IC can wait to send the packaged Informs in a log to the appropriate home memory controller until (a) the packaged Inform message is too big to hold any more Informs, or (b) the log is full, or (c) a timeout occurs (after a pre-specified amount of logical time, currently set to 32). The timeout mechanism helps to avoid processing Informs out of timestamp order at the VWB, by not letting the skew between Inform timestamps and the current logical time grow too large.

At the memory's home node, the Inform first goes through a splitter that splits the Inform message into separate still-compressed Inform messages that have the same timestamp. Splitting them by timestamp enables the VWB to reorder all Informs by their logical times (which are uncompressed in the headers of the split Inform messages). When the memory controller is ready to process the head of the VWB (i.e., the VWB is full and a new Inform message arrives), the VWB entry is decompressed before being applied to the Verification Memory.

Lossless is the simplest scheme we present in this paper. Simpler schemes exist, such as run length encoding of Informs, but we found them to be much less effective. *Lossless* uses the least hardware of our three schemes, but it achieves the least compression, as we will show in Section 5.

Lossless compression with Combining (*Combining*). Our goal in *Combining* compression is to exploit our semantic knowledge about the Inform stream to combine multiple logical Informs in a given Inform log into a single atomic Inform and thus achieve better compression. On every memory operation, the IC tries to combine Informs to the same address or addresses within the same memory block. We can only combine two Informs if we can guarantee that they would have been processed from the VWB without any intervening Inform-Stores from other processors. For Informs of the same type (Load/Store) and size (e.g., 4-bytes) that access neighboring addresses in the same block, we can coalesce them into one Inform. For Informs to the same address (that are the same size), we can combine them in the following way:

- Load-Load: If the data for both accesses is the same, we can eliminate the second load.

4. If the Data/Inform network did not support point-to-point ordering, then we would need to use small buffers at the memory controller to order Informs from each processor before decompression. Otherwise, the memory controller would not be able to mirror the history table management at the processors.

- Store-Load: If the data of the load matches the store, the load is eliminated.
- Load-Store: If the data of the store matches the load, the store is eliminated.
- Store-Store: The first store is eliminated.

To ensure that combined Informs can be processed at the VWB without intervening Inform-Stores from other processors, the IC does not coalesce or combine Informs to a memory block across a cache coherence request to that block. The simplest way to implement this is to send all of the (packaged and combined) Informs in a log to the memory controller if a coherence request arrives for an address that *could* be in that log (i.e., the block address maps to that log). Thus no Inform-Stores by other processors to the same address can be interleaved with the combined Informs in the VWB. This is trivially true, because all combined accesses have the same logical time, and Informs are sorted by logical time in the VWB. Since there is no causal relationship between operations by different processors at the same logical time (by definition of logical time), the VWB arbitrarily orders them by arrival order. We also implement more sophisticated combining across coherence requests, which requires snooping of the Inform logs to ensure that an incoming coherence request is not for a block that is current in an Inform log. If a coherence request matches a block that is in a log, then the IC can no longer keep combining.

We will show in Section 5 that *Combining* achieves far greater compression than *Lossless*, but at the cost of some additional hardware at the ICs and memory controllers.

Lossy compression (*Lossy*). In this scheme, we are explicitly trading off an increased probability of false verification against lower bandwidth costs. Every byte of the original data word is mapped onto a fixed smaller number of bits using XOR hashing. After this step, the rest of the process is the same as in *Combining*. Due to *aliasing* (mapping two block values to the same hashed value), it is possible to introduce false negatives with lossy compression, which makes *Lossy* less appealing. *Lossy* does, however, enable us to hash values in the Verification Memory, as we discuss in Section 4.2.

We will see in Section 5 that *Lossy*'s ability to compress is not enough better than *Combining* to be worth the additional probability of false negatives.

4.2 Challenge #2: Verification Memory

We could naively implement Verification Memory by duplicating the memory arrays. However, our goals are to minimize storage and bandwidth, such that we can use a small enough Verification Memory Cache (VMC) that it fits on-chip and does not consume expensive off-chip bandwidth. The VMC is a memory-side cache that is accessible only by the memory controller. The VMC stores memory blocks which can be partially valid.

We can arbitrarily size the VMC by adding another aspect of probability to pDVSC. There are two orthogonal options. One option is to hash the values, which works well with *Lossy* compression of Informs. A hashed-value VMC has a non-zero probability of aliasing, which can potentially lead to false negatives. This probability can be adjusted by choosing the size of the hashed values and the hashing function.

A second option is to have fewer entries than a full Verification Memory (but with no backing store). An Inform-Store miss deletes a victim. An Inform-Load miss precludes verification of that Inform-Load, which is a potential false negative. The probability of an Inform-Load miss can be reduced by trading off VMC size and associativity.

Clouseau currently uses the first option in conjunction with *Lossy*, and it uses the second option in conjunction with *Lossless* and *Combining*. Our evaluation in Section 5 explores the tradeoff between VMC size and associativity and the corresponding VMC miss rate.

4.3 Implementation Details

While we have described the important issues in Clouseau, we have until now omitted a couple of the subtler implementation details. First, we must represent logical timestamps in a finite number of bits. The number of logical major times in existence at any point in time is roughly equal to the product of the number of outstanding checkpoints (equal to four in the configuration of SafetyNet we use with Clouseau) and the number of logical cycles per checkpoint (1000 in this SafetyNet configuration). We can thus represent this number of logical times with 12 bits. To avoid wraparound, we need an additional bit, which brings us to 13.

Second, we have to deal with I/O accesses to shared memory. We assume that we do not have sufficient access to the I/O system to implement timestamping of its reads and writes. Thus, Clouseau does not incorporate I/O accesses into its verification scheme. Clouseau ignores I/O reads from memory. We assume that an I/O write must obtain coherence permission first, and we can thus timestamp the effect of the write (which we interpret as multiple stores if the write affects more than one memory block) with the logical time of the coherence request (just like a normal coherence request).

5 Evaluation

In this section, we seek to experimentally determine whether Clouseau is viable. We wish to show that Clouseau detects errors without incurring unreasonable costs, in terms of performance, bandwidth, or storage.

TABLE 1. Target System Parameters

Processor Core	4 billion instructions/sec (if caches were perfect), in-order
L1 Cache (I and D)	128 KB, 4-way set associative
L2 Cache	4 MB, 4-way set-associative
Memory	2 GB, 64 byte blocks
Miss From Memory	180 ns (uncontended)
Interconnection Network: Address	broadcast tree, totally ordered, 6.25 GBytes/s links
Interconnection Network: Data & Inform	2-dimensional torus, point-to-point order, 6.25 GBytes/s links
Checkpoint Log Buffer	512 kbytes total, 72 byte entries
Checkpoint Interval	1000 broadcast coherence requests

5.1 Methodology

We simulate an 8-node target system with the Simics full-system, multiprocessor, functional simulator [14], 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 SafetyNet backward error recovery, 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 memory hierarchy simulation.

Processor Model. We use Simics to model an in-order processor core that, *given a perfect memory system*, would execute four billion instructions per second (e.g., 2-wide at 2GHz) and generate blocking requests to the cache hierarchy and beyond. Due to cache misses, the processor does not achieve this maximum rate. We use this simple processor model to enable tractable simulation times for full-system simulation of commercial workloads. While an out-of-order processor model might enable certain fault models, as discussed in Section 3.6, it would not affect the results.

Memory Model. We have implemented a memory hierarchy simulator that captures all state transitions (including transient states) of our coherence protocol in the cache and memory controllers. We model the interconnection network and contention within it. In Table 1, we present the design parameters of our target memory system. With a checkpoint interval of 1000 broadcast coherence requests and four outstanding checkpoints, SafetyNet can tolerate long dynamic verification latencies.

SafetyNet. Our memory system simulator models the details of SafetyNet checkpoint/recovery. We use the same checkpoint log buffer size (512 kbytes) and other parameters as Sorin et al. [23],

TABLE 2. 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 transactions.
Java Server: SPECjbb2000 is a server-side java benchmark that models a 3-tier system with driver threads. We used Sun's HotSpot 1.4.0 Server JVM. Our experiments use 24 threads and 24 warehouses (~500 MB of data). We warm up for 100,000 transactions, and we run for 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 a 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 [25], with the 16K body input set. We measure from the start of the parallel phase to avoid measuring thread forking.

since this configuration was demonstrated to incur only negligible performance degradation due to SafetyNet overheads.

5.2 Workloads

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, for comparison, one scientific application. These workloads are described briefly in Table 2 and in more detail by Alameldeen et al. [1]. To handle the variability inherent in commercial workloads, we run each simulation 3-6 times with small pseudo-random perturbations. Our result graphs show the mean results as well as error bars that represent one standard deviation below and above the mean.

5.3 Results

We now present the results of our evaluation of Clouseau. While all of these results are for the broadcast snooping SMP described in Section 5.1, the results for a system with directory cache coherence (not shown) are qualitatively the same.

Error Detection. The goal of Clouseau is to detect all violations of SC. The only *fundamental* cause of false verification (i.e., not due to implementation optimizations) is the possibility of accidentally processing Informs out of logical time order because of finite VWBs. We experimented on an error-free system with *Combining* compression by varying the VWB size and detecting when an Inform arrives too late (in physical time) to be processed in order in logical time. These

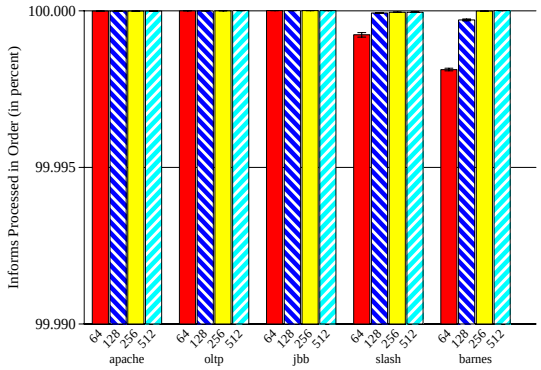


Figure 6. Probability of In-Order Inform Processing as Function of VWB Size (in entries)

situations may lead to false verifications, although the vast majority of reorderings would not (e.g., since reorderings of Informs for different memory blocks cannot cause false verifications). Figure 6 shows that the probability of out of order processing is still exceedingly small (less than 0.003%), even for VWBs with only 64 entries (at 64 bytes/entry, this is only 4 Kbytes).

To test Clouseau's error detection coverage, we periodically injected single errors into the memory system. These errors included dropped messages, reordered broadcast requests, incorrect processing of coherence requests, and errors in Clouseau itself (e.g., dropped Informs). For VWB sizes greater than 32 entries, Clouseau detected all errors.

Performance. Since Clouseau can pipeline its latency for detecting potential errors and SafetyNet was shown to impose negligible performance overhead as long as error detection latencies can be pipelined [23, 22], the performance of an error-free system with Clouseau (and unlimited available interconnection network bandwidth) is negligibly worse than a system without Clouseau. In Figure 7, we plot the runtime (inverse of performance) for: a system without pDVSC, Clouseau with *Combining* with 50 GB/s Data/Inform network links (effectively unlimited bandwidth), and Clouseau with *Combining* with 6.25 GB/s links. We observe that Clouseau's performance does not degrade from 50 GB/s down to 6.25 GB/s links. We also see that, statistically, Clouseau does not perform worse than the non-pDVSC system except for in the oltp workload, for which performances are close.[5]

Inform Bandwidth. We experiment with a system with unlimited available link bandwidth on its Data/Inform network and we observe how much bandwidth Clouseau uses on its most highly contended link. In Figure 8, we study the bandwidth usages of five systems: without pDVSC, Clouseau with no compression of Informs, and Clouseau with the three types of compression discussed

5. The results for slash show that *mean* performance for Clouseau is better than for a system without pDVSC, but the error bars reveal that this performance differential is not statistically meaningful.

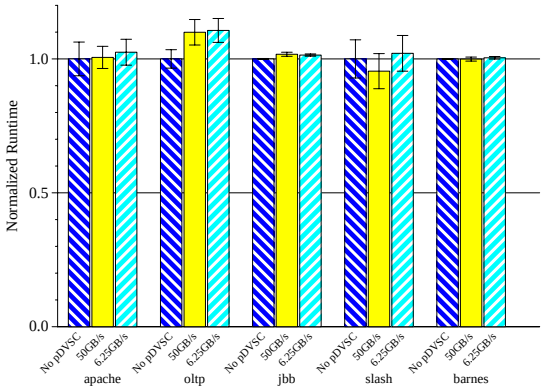
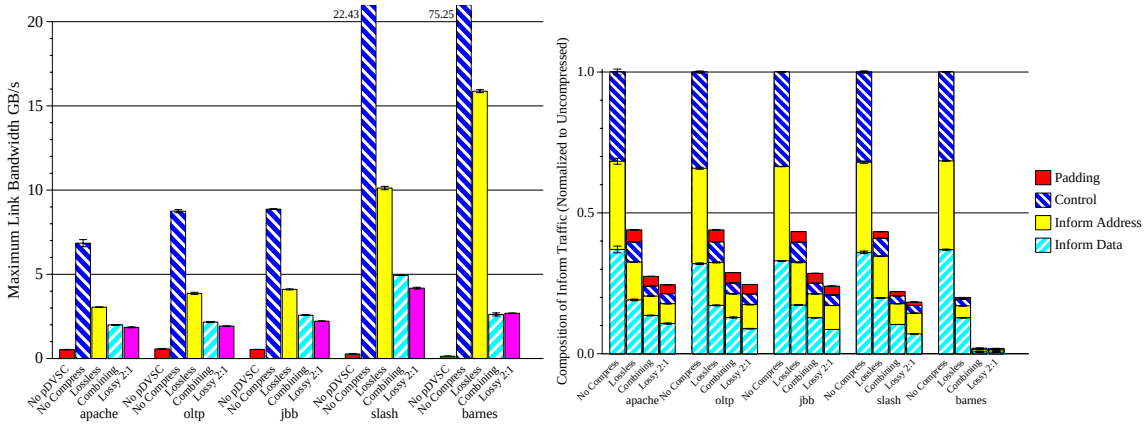
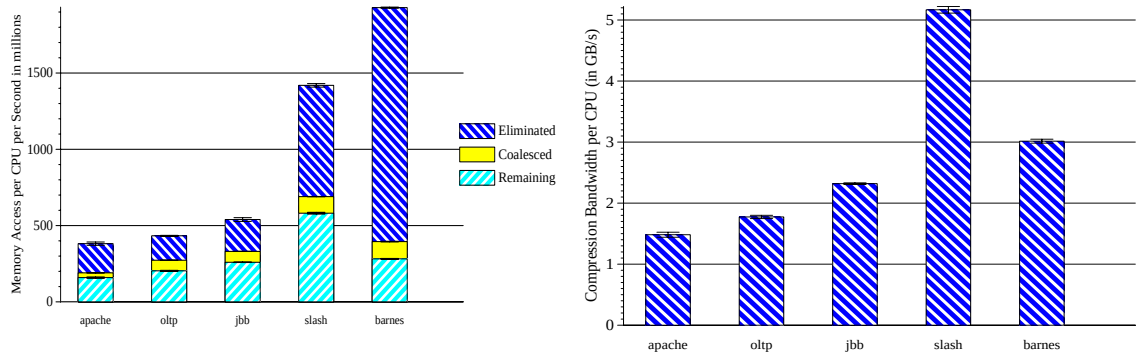


Figure 7. Clouseau Performance



(a) Relative bandwidth usage

(b) Composition of Inform bandwidth

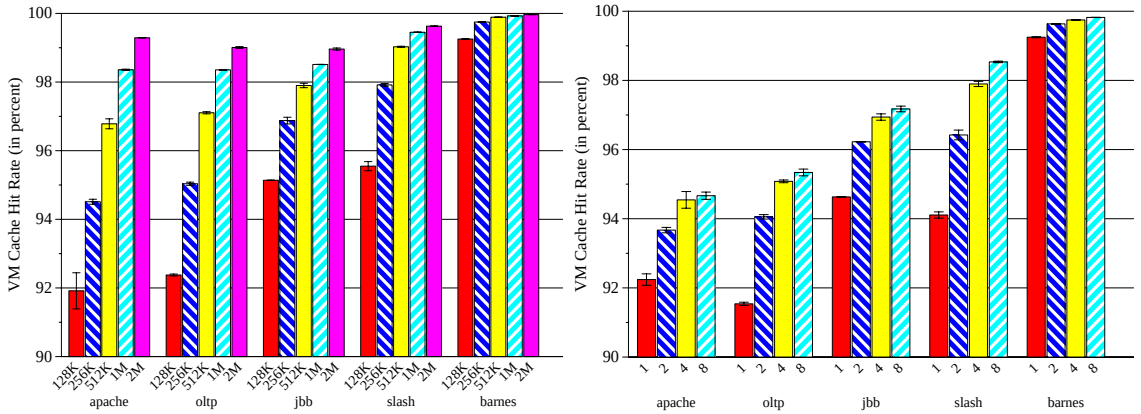


(c) Compression bandwidth (in accesses/sec)

(d) Compression bandwidth (in bytes/s)

Figure 8. Clouseau's Interconnection Network Bandwidth and Compression

in Section 4.1 (*Lossy* uses 2:1 data compression). Figure 8(a) shows that uncompressed Clouseau is quite costly, but that compression reduces the bandwidth usage significantly. In fact, *Combining* bandwidth usage is well less than that provided by the Alpha 21364 [17]. Among the compression schemes, *Combining* compression achieves good compression without having to resort to lossy techniques. Figure 8(b) shows the composition of Inform messages. We observe that control (which includes timestamps in the uncompressed case), addresses, and data are all compressed sig-



(a) Hit rate vs. size (4-way set-associative) (b) Hit rate vs. associativity (256 KByte size)

Figure 9. VMC Load Hit Rate

nificantly. *Lossy*, compared to *Combining*, does not help much to compress data (even for 8:1 compression—results not shown), which is why *Lossy* does not do much better than *Combini*ng. Thus, while Clouseau does incur a non-negligible interconnect bandwidth cost, total bandwidth usage is not prohibitive, especially for CMPs with their greater available bandwidths. One other concern is being able to implement the compression hardware that can keep up with the requests from the processors. In Figure 8c, we plot the rate of memory accesses per processor per second (in millions). We observe that *Combining* compression can greatly ease the pressure on the compression hardware by eliminating and coalescing Informs that would otherwise have to be compressed. Even with *Combining*, Figure 8d shows that compression hardware must still compress between 1.5-5.5 GB/s of Informs. Fortunately, much of this compression is done in parallel, including compression of Informs in different logs. Compression of Inform addresses and data can also be done in parallel.

Verification Memory. Our initial VMC design is a set-associative cache. Figure 9 plots the VMC load hit rate as a function of VMC size and associativity, for Clouseau with *Lossless* compression. From Figure 9b it appears that 4-way associativity is the sweet spot. For a 4-way VMC that is only 1 MB, all benchmarks achieve hit rates greater than 98%, as shown in Figure 9a. These encouraging results even for small VMCs reflect the locality in the access streams.

Conclusions. Clouseau detects injected errors, does not appreciably degrade performance, and it can be designed to avoid false verifications. Nevertheless, it does not come without costs. Most notably, the bandwidth for Informs and storage for Verification Memory could be expensive if implemented naively. We show, though, that an optimized design can greatly reduce these costs and make Clouseau a feasible approach for improving system availability.

6 Related Work

Related work exists in dynamic verification and logical time ordering of events. We do not discuss prior research in consistency speculation, since it assumes error-free execution.

6.1 Dynamic Verification

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]. For this system model, DIVA is excellent, but it is unclear how to scale its basic idea to multithreaded memory systems. At the multithreaded level, Sorin et al. dynamically verify end-to-end invariants of the cache coherence protocol and interconnection network [22]. 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, and it does not provide a way to integrate it with a recovery mechanism. Cain and Lipasti propose an algorithm based on vector clocks for DVSC, but they leave for future work the issues of implementation and integration of the algorithm with a BER mechanism [3].

6.2 Logical Time Ordering of Events

The original work on using logical time to order events in distributed systems was developed by Lamport [12]. Static verification with Lamport clocks [24, 18] uses logical time to order all possible events and show that a system design can never create orderings that violate the consistency model. Isotach networks [19] use logical time to specify the ordering of message arrivals in the network. Timestamp Snooping [15] uses logical time to create a total order of cache coherence requests on a physically unordered interconnection network. In software, parallel discrete event simulation [10] uses logical time to coordinate the simulation of a parallel target machine on a parallel host.

7 Conclusions and Future Work

We have demonstrated how to improve system availability with probabilistic dynamic verification of sequential consistency (pDVSC) for multithreaded memory systems. Our first implementation of pDVSC, called Clouseau, pipelines verification with the active execution, in order to hide its latency. Our evaluation demonstrates that we can trade bandwidth and storage to detect violations of SC with an arbitrarily high probability. Future work will address more optimized implementations, as well as other memory consistency models.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CCR-0309164 and EIA-9972879, IBM, and a Duke Warren Faculty Scholarship. Any opinions, findings and conclusions or recomendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF). We would like to thank Alvy Lebeck, the Duke Architecture Group, Anne Condon, Mark Hill, Alan Hu, David Wood, Jesse Bingham, and Raimund Seidel for their helpful discussions about this work. We thank David Becker, Tong Li, Carl Mauer, and Min Xu for their gracious help with simulator issues.

References

[1] A. R. Alameldeen, M. M. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. 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 *Proceedings 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. In conjunction with ISCA.

[5] A. Condon and A. J. Hu. Automatable Verification of Sequential Consistency. In *Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 113–121, July 2001.

[6] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4), July-August 2003.

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

[8] M. Farrens and A. Park. Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[9] P. Fenwick. Punctured Elias Codes for Variable-Length Coding of the Integers. Technical Report 137, University of Auckland, Dec. 1996.

[10] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

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

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

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

[14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[15] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore,

M. Plakal, M. D. Hill, and D. A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, Nov. 2000.

[16] S.-W. Moon, K. G. Shin, and J. Rexford. Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 203–212, June 1997.

[17] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 Network Architecture. In *Proceedings of 9th Hot Interconnects Symposium*, Aug. 2001.

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

[19] P. F. Reynolds, Jr., C. Williams, and R. R. Wagner, Jr. Isotach Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):337–348, Apr. 1997.

[20] 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.

[21] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.

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

[23] 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 *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.

[24] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. Martin, and D. A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.