Verification of Chip Multiprocessor Memory Systems Using A Relaxed Scoreboard

Ofer Shacham, Megan Wachs, Alex Solomatnikov, Amin Firoozshahian,
Stephen Richardson and Mark Horowitz
Department of Electrical Engineering, Stanford University, California
Email: {shacham, wachs, sols, aminf13, steveri, horowitz}@stanford.edu

Abstract

Verification of chip multiprocessor memory systems remains challenging. While formal methods have been used to validate protocols, simulation is still the dominant method used to validate memory system implementation. Having a memory scoreboard, a high-level model of the memory, greatly aids simulation based validation, but accurate scoreboards are complex to create since often they depend not only on the memory and consistency model but also on its specific implementation. This paper describes a methodology of using a relaxed scoreboard, which greatly reduces the complexity of creating these memory models. The relaxed scoreboard tracks the operations of the system to maintain a set of values that could possibly be valid for each memory location. By allowing multiple possible values, the model used in the scoreboard is only loosely coupled with the specific design, which decouples the construction of the checker from the implementation, allowing the checker to be used early in the design and to be built up incrementally, and greatly reduces the scoreboard design effort. We demonstrate the use of the relaxed scoreboard in verifying RTL implementations of two different memory models, Transactional Coherency and Consistency (TCC) and Relaxed Consistency, for up to 32 processors. The resulting checker has a performance slowdown of 19% for checking Relaxed Consistency, and less than 30% for TCC, allowing it to be used in all simulation runs.

1. Introduction

Validating the operation of a multiprocessor's memory system remains a challenging task. Most practical approaches to validation have focused on a framework based on either race-free diagnostics or pseudo random test suites combined with a golden model (also known as a scoreboard) of the memory system behavior. While this methodology is widely used and works well, constructing the scoreboard for a modern system is a difficult task. One of the factors that makes it complex is that memory consistency models such as sequential consistency (SC) specify rules and axioms that are easier to describe by a non-deterministic state machine—they do not completely specify the memory system's behavior. As a result, a given consistency model may have multiple correct implementations that perform differently, and thus need different golden models. This coupling causes two problems. First, one needs to create a new golden model for each implementation, and second, it is hard to keep the validation model completely separate from the implementation, which can lead to correlated errors. This problem is only getting worse with the requirement of some multiprocessor systems to support multiple consistency models, including a transactional memory model like Transactional Coherency and Consistency (TCC) [1].

Recent attempts, such as TSOtool [2], take a different approach to the problem of memory verification. Instead of dealing with the complexity of the implementation, TSOtool does a post-mortem analysis of the processors' traces. This approach checks that the observed trace values are logically correct with respect to the consistency model. Since it does not specify what the output should be at each cycle, or even what the ordering must be, it reduces the coupling between the verification model and the design details. The key insight is that this undesirable verification-design coupling can be broken by creating a checker that allows multiple output traces to be correct. We leveraged this insight from TSOtool to create a new approach toward validating memory system behavior, the Relaxed Scoreboard.

The Relaxed Scoreboard is a verification methodology that attempts to come as close as possible to verifying the temporal behavior of a CMP memory system implementation, while avoiding exponential complexity. Like a traditional scoreboard, the relaxed scoreboard is constructed

This research was partly sponsored by the Focus Center Research Program Center for Circuit and System Solutions (www.c2s2.org) under contract 2003-CT-888. Additionally, this research was partly sponsored by DARPA/IPTO under Air Force Research Laboratory (AFRL) contract FA8650-08-C-7829. Ofer Shacham is partly supported by the Sands Family Foundation. Megan Wachs is supported by a Sequoia Capital Stanford Graduate Fellowship.

to be an intuitive and simplified temporal model of the memory system, but like TSOtool, it is not tied to a specific implementation. The decoupling of the relaxed scoreboard from the implementation is done, similar to earlier work by Saha et al [3], by having a set of multiple possible values associated with each memory location. This set includes all values that could possibly be legally read from this address by any processor in the system.

We find that by using this relaxation (keeping a set of possibly correct answers), a relaxed scoreboard methodology introduces a number of traits that are important for efficient RTL design and verification. The construction of the scoreboard is derived directly from the relevant consistency model properties. Each of those properties can be considered separately, therefore enabling the verification environment to be developed incrementally along with the design, rather than requiring a complete model on day one. In contrast to static post-mortem trace analysis algorithms, the relaxed scoreboard is designed to be a dynamic on-the-fly checker, meaning that any error will be reported immediately, saving valuable human and compute time. Furthermore, the combination of dynamic runtime analysis plus the scoreboard's incremental construction enables the user, at later design phases, to incorporate key information from the design (such as exact arbitration time) into the scoreboard. While at the beginning of the design/verification cycle the number of acceptable results might be large, this set can become smaller as more sophisticated checks are added to the scoreboard, and can, if needed, turn into a tight, accurate model.

To understand where the relaxed scoreboard fits into a validation methodology, the next section quickly reviews modern validation approaches, and describes recent work upon which our solution is built. With this background, Section 3 formally describes two different memory consistency problems: verifying sequential consistency both with and without temporal information about the memory operations. Section 4 then describes the basic approach for creating a relaxed scoreboard, and demonstrates how it can be used to address the consistency problems described in the Section 3. Since validating memory consistency is NP complete and our scoreboard runs in polynomial time, Section 4 also describes the type of errors that can escape our checker. Section 5 extends the discussion to a real design example, where relaxed scoreboards were used to verify the implementation of an actual CMP design, the Stanford Smart Memories, for both Relaxed Consistency and Transactional Coherency and Consistency (TCC). Section 5 also provides information on both the effectiveness and overheads of using a relaxed scoreboard to validate the memory system.

2. Prior Work

Over the years, academia and industry have developed many RTL verification methodologies, which are today commonly employed among ASIC design groups. A number of books such as [4], [5] and [6] will guide the reader through the process of forming and implementing a verification plan. The verification goal is always to find as many errors as possible, as early in the design cycle as possible, and with as few resources spent as possible.

Functionally verifying an RTL design starts with a testbench which exercises the design under test (DUT) by stimulating its input signals. Industry tools such as OpenVera [7] or Specman [8] are used to generate, drive and monitor design interfaces, verifying the behavior of internal and output signals. Common approaches for increasing testbench effectiveness include employing assertions [9] [10], high level modeling, and comparison with "golden models" (or verification "scoreboards").

A verification scoreboard is a component that behaves as a perfect reference model of the DUT, allowing an accurate comparison between the implementation's output and the expected functionality. Scoreboards are often written in higher level functional languages such as C, Vera or 'e'. The strength of a traditional scoreboard comes from its ability, as an end-to-end checker, to specify the correct output of the design at any time. Having such a reference model means that one can check the behavior of the system against any input pattern, which facilitates using random test generation in addition to self checking diagnostics. Random test input significantly improves the probability of hitting corner case errors in the design. Unfortunately, since in a shared memory system the detailed timing of the operations determines the output, creating a golden model and proving its correctness becomes an extremely difficult task.

Similarly to RTL scoreboards, Meixner and Sorin developed validation techniques for dynamic detection and recovery of consistency violations in a memory system [11] [12]. However, these studies rely on sufficient conditions for sequential consistency and therefore also produce false positives which are highly undesirable in an RTL verification environment. On the other side of the spectrum, much work has been done to formally prove that a hardware protocol really implements the memory consistency model for which it was designed. Methods such as a Cartesian product of Finite State Machines [13] [14], formal specification with Model Checking [15] [16] [17], and Lamport clocks [18] [19] are often used. However, the state space of an RTL implementation is very large, and formal methods can rarely be applied to it at system level. One way to address the additional complexity RTL introduces over its specification is to extract internal information from the RTL design. Serdar, Yuan and Brannon used a refinement map to transform simulation steps

in an implementation to state transitions at the specification level [20]. Lepak used exact time stamps of store-commit and of load-bound-to-value to verify sequential consistency in a software simulation environment [21]. The shortcomings of relying on the design's internal details are that it couples the checker with the implementation, and that in many RTL designs, in contrast to software simulators, extracting exact protocol state may not be practical.

In addition, protocols are often defined mathematically using non-deterministic automata. The inherent non-determinism implies that when implemented in RTL, multiple correct designs may exist, each of which may produce different traces of execution. Consequently, a traditional golden model for one implementation cannot be used for a different implementation. Therefore, a more robust solution is needed, one that encapsulates the non-determinism of the protocol, and hence can be used to verify different implementations of the protocol. One good example is the TSOtool [2], a suite of post-mortem algorithms for trace analysis, which checks that a CMP implementation complies with the Total Store Ordering consistency model. Similar work has been done to verify TCC implementations [22].

We realize we can build on the work of [2] and [22], to create a new approach to scoreboard design that makes the construction of a memory scoreboard easy. Thus one gets the advantages of the ability to verify portions of the model early-on, and dynamically add, adjust and remove protocol restrictions (or axioms) later in the process, as well as performing the checking at real time, without generating long trace files.

Most importantly, in comparison to a TSOtool-like checker which strives to verify existence of a **logical order** of operations based on the consistency model, the relaxed scoreboard strives to verify existence of a **temporal order** of operations. The latter is a stronger condition because all temporal orders are also logical order, but not vice-versa, as we show in the next section.

The idea of a memory system checker that holds "*valid sets*" for the purpose of decoupling the checker from the implementation was first introduced by Saha et al [3]. We extend this idea to a complete framework, the relaxed scoreboard, and use it to verify multiple consistency models on a real CMP RTL implementation. We also show both an analytical analysis and experimental results regarding the effectiveness and overhead of using this methodology. Finally, while the concept of valid sets as presented by [3] proves useful, it can not catch all errors since the problem of verifying an execution trace of a multiprocessor is NP-Complete (see Section 3). Therefore, this paper also adds an analysis of the completeness of the checker and suggests improvements.

3. Problem Definition

Verification of a shared memory system is in essence the attempt to prove that the hardware complies with the mathematical definition of the coherence and consistency model from a programmer standpoint. For example, deciding whether a set of processor execution traces complies with sequential consistency is known as *Verifying Sequential Consistency* (VSC) [23]. Similar definitions apply for other consistency models [24]. When dealing with RTL/architectural verification, as opposed to post silicon verification, an attractive verification approach is to leverage not only the values observed on the system's ports, but also their temporal information—the time at which they were observed. By using temporal information, a checker can also flag errors that obey the consistency model but should not occur in real hardware. The temporal version of VSC is known as *Verifying Linearizability* (VL) or *Atomic Consistency* [25].

The following are the formal definitions of Verifying Sequential Consistency and Verifying Linearizability, as described by Gibbons and Korach [25]. *Verifying Sequential Consistency* is based on respecting program orders and the read/write semantics, while ignoring start-of-interval and end-of-interval times.

> INSTANCE: Variable set A, value set D, finite collection of sequences $S1, ..., Sp$, each consisting of a finite set of memory operations of the form "$read(a, d)$" or "$write(a, d)$", where $a \in A$ and $d \in D$.
>
> QUESTION: Is there a sequence S, an interleaving of $S1, ..., Sp$ such that for each $read(a, d)$ in S there is a preceding $write(a, d)$ in S with no other $write(a, d')$ between the two?

Verifying Linearizability adds the further constraint that the schedule *S* must respect the time intervals for the operations.

> INSTANCE: Variable set A, value set D, finite collection of sequences $S1, ..., Sp$, each consisting of a finite set of memory operations of the form "$read(a, d, t_1, t_2)$" or "$write(a, d, t_1, t_2)$", where $a \in A$ and $d \in D$, and t_1 and t_2 are positive rationals, $t_1 < t_2$, defining an interval of time such that all intervals in an individual sequence are pairwise disjoint, and t_1 and t_2 are unique rationals in the overall instance.
>
> QUESTION: Is there an assignment of a distinct time to each operation such that 1) each time is within the interval associated with the operation; 2) for each $read(a, d, \tau_1, \tau_2)$, there is a $write(a, d, t_1, t_2)$ assigned an earlier time, with no other $write(a, d', t'_1, t'_2)$ assigned a time between the two?

Similar definitions can be applied to other consistency models such as Total Store Order, Weak Ordering, etc. One should note that in practice, most CMP architectures would allow non-blocking writes, and even multiple outstanding reads. This means that a black-box verification environment will not have the end-of-interval time stamp for $write$ operations,

and that all operations may not be pairwise disjoint. The practical implication is that a checker may need to keep track of more than P concurrent accesses, where P is the number of processors in the system. One such architecture is the Stanford Smart Memories [26], on which the relaxed scoreboard was evaluated.

Gibbons and Korach proved that VSC is NP-Complete [25]. Cantin, Lipasti and Smith extended the proof to all other commonly used consistency models [24]. Moreover, the problem is NPC even when any of address-range/number-of-processors/accesses-per-processor factors is bounded. Although VL was also shown to be an NPC problem, the limiting factor for VL is the number of processors that are accessing a shared address, rather than the length of the trace.

To conclude the formal definitions of VSC and VL, we argue that for the purpose of hardware verification, VL is a stronger correctness criteria since any set of traces that would violate a checker for VSC will also violate a checker for VL. Furthermore, there can be traces with errors that violate VL but do not violate VSC. The following set of temporal sequences demonstrates such a scenario.

Time	S1	S2	S3
10	A:RD(a,5,10,11)	D:WR(a,1,10,11)	G:WR(a,2,10,11)
20	B:RD(a,5,20,21)	E:WR(a,3,20,21)	H:WR(a,4,20,21)
30	C:RD(a,5,30,31)	F:WR(a,5,30,31)	I:WR(a,6,30,31)

* Assume MEM[a]=0 at time 0
** This example illustrates a 'stuck-at' error case

In this counterexample one can easily find a global ordering that complies with SC (E.g.,{G,H,I,D,E,F,A,B,C}) . However, no ordering can be found that complies with VL. This implies that a checker with temporal information will find more error cases than a post-mortem checker without temporal information. The latter might not find errors that violate causality as in this example. Section 4.1 will show another example of a property violation (write atomicity) that can only be found by a checker that leverages temporal information.

4. Relaxed Scoreboard

To verify the linearizability of a CMP implementation, the Relaxed Scoreboard is built as a global online design checker. As noted before, the relaxed scoreboard does not compare an observed value to a single known correct result. Instead, it keeps a set of possible results and ensures that the observed value is within this bounded set. It is an oracle that can answer the following question: "Does the value *Val*, that was observed at interface *ifc* of the design, exist in the group of allowed values for that interface, at that given time?" In this sense, the relaxed scoreboard is simply a data structure of groups, which can answer queries and receive updates.

In order to understand why a single value might be hard to predict but a bounded set is not, one can think about a simple

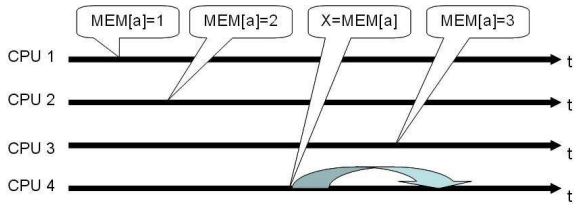

Figure 1. Non-determinism of a CMP memory system.

4-input round-robin arbiter. In order to predict who would win the arbitration, a golden model would have to "know" or imitate the internal state of the arbiter. In contrast, in order to determine the set of possible winners, a relaxed scoreboard only needs to know which inputs are asserting a request and which inputs have already been granted (in previous rounds). It would then perform simple accounting operations to keep the set of possible winners as tight as possible. Note that in addition to simply checking the DUT outputs, the scoreboard uses those outputs to reduce the allowable future outputs.

Verification of a memory system in a shared memory multiprocessor is particularly challenging because of the inherently non-deterministic and timing-dependent execution of memory accesses, which inevitably produces race conditions. The sequence of memory accesses from each processor might depend on how the previous race condition was resolved. Figure 1 shows a simple example of a race condition in a four-processor system. Processors 1, 2 and 3 are initiating writes to address a while processor 4 is reading the value saved at that address. The data is returned some time after the load started, as noted by the curved arrow. Depending on the exact arbitration mechanism, each of the values 1, 2 or 3 might be returned as the result of the load instruction.

In order for the relaxed scoreboard to be an efficient verification tool, we define two requirements:

1) Bounded Uncertainty: The set of possibly correct answers must be kept bounded. If this set grows in time, then the testbench as a whole loses efficiency. This is the most important requirement of the relaxed scoreboard.
2) On-The-Fly: Errors should be detected on-the-fly, as close as possible (in time) to their origin. Since the relaxed scoreboard is designed with big, complex designs in mind, it must recognize the need of both designers and verification engineers for a fast turnaround time.

Though not a requirement, it is recommended that one use a *black-box* approach, at least in the early part of the design-verification cycle. The reason is twofold: (a) The more information that the scoreboard is using from the design, the more likely it is to repeat (and therefore mask) the same errors. (b) The more internal signals and states that the scoreboard is using from the design, the more dependent it becomes on the specific implementation. To be a black

box checker, the relaxed scoreboard should be connected to the interface of the DUT through monitors that observe the signals from the implementation and send higher level abstraction messages to the checker. Once the design matures, and gross errors are flushed out, it may be desirable to incorporate key information (such as arbitration time or snoop messages) to tighten the matching between the scoreboard and the actual design.

To meet the above requirements efficiently, the relaxed scoreboard implementation uses an internal, easily searchable data structure. The main purpose of the scoreboard's internal data structure is to keep a set of possibly correct values, sorted by address and chronological time. Each entry in the scoreboard's data structure is associated with a sender ID, value, and start and end time stamps, as observed during runtime on the relevant processor interface. In addition, each entry contains a set of expiration times, one for each possible interface on which this transaction may be observed. Upon arrival of a new transaction from a monitor, the scoreboard performs a series of *checks* followed by a series of *updates*. The checks produce the scoreboard's oracle answer, based on the values stored in the data structure, of whether that operation is possible. The updates use the DUT output to update the internal data structure with the new information, reducing the set of answers that the scoreboard considers as correct for future operations.

Updates reduce the set of possible values held in the data structure, but the scoreboard can perform simple checks even before any updates are added. For example, the most useful check is a check of causality (a value that is read from an address must have been previously written to that address). In this very loose scoreboard, any stored value would be considered as correct. Under such a simple scoreboard, the check would quickly become ineffective and the data structure would explode in size. Thus, updates are the completing and crucial part of the scoreboard. Updates use the rules of the implemented protocol to reduce the set of possible values, keeping it up to date with the values exercised by the DUT.

Updates and checks can operate independently, so new checks and updates can be added to verify different aspects of the specification, or existing checks and updates can be made more effective by considering more details of the implemented system. This characteristic allows the verification effort to concentrate on the simplest and easiest to detect errors first, and gradually move towards more sophisticated design problems. This implies that while at the beginning of the design and verification cycle the number of acceptable results might be large, this set later becomes smaller, evolving towards a tight, accurate model.

Overall, the relaxed scoreboard is essentially a set of global protocol-level assertions. The assertions are constructed to verify that certain rules of the protocol are followed at a high level, without actually relying on or examining the implementation details.

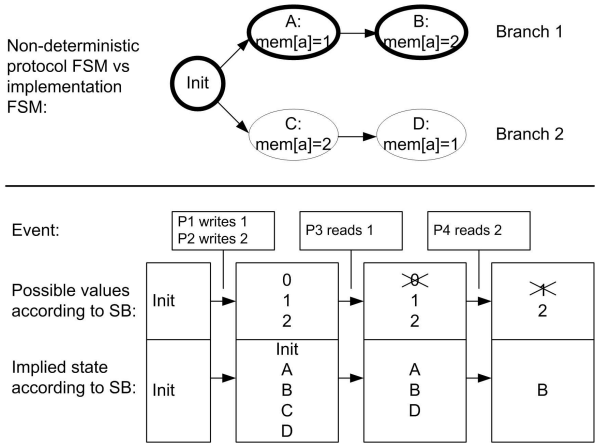

Figure 2. Write atomicity example.

Sections 4.1 - 4.3 show a few examples to demonstrate the use of the relaxed scoreboard in verifying implementations of shared-memory protocols. In the examples, *write(a,d)* and *read(a,d)* are used respectively to denote a write or a read of data *d* from address *a*. To make the consistency problems easier to see in these examples, reads are assumed to complete in less than 10 cycles, and *nop*s indicate operations that are not relevant for the property in question.

4.1. Write Atomicity

As a run-time checker, the scoreboard needs to determine whether a given operation corresponds to a legal transition in the machine state. As our first example, let us examine how a relaxed scoreboard would check for write atomicity, which is part of many consistency models [27]. In order to detect write atomicity violations, we convert the protocol property into a check and update, and add them to the relaxed scoreboard:

> PROTOCOL PROPERTY: All processors see writes to the same location in the same order: single serialization point. If there is a store that is observed by one processor, it should be observed as completed by all processors.
> UPDATE: When one processors loads a value from a certain location, mark all previously committed stores to this location as invalid after the load time.
> CHECK: A load can only return a value which was valid at some point between its start-of-interval and end-of-interval

The following is an observable write atomicity violation:

Time	P1	P2	P3	P4	P5
10	WR(a,1)	WR(a,2)	nop	nop	nop
20	nop	nop	RD(a,1)	nop	nop
30	nop	nop	nop	RD(a,2)	nop
40	nop	nop	nop	nop	RD(a,1)

* Assume MEM[a]=0 at time 0
** Assume read operations complete in less than 10 cycles

In this example, Processors 3, 4, and 5 disagree on the ordering of the store to address *a*. Figure 2 illustrates the operation of the relaxed scoreboard for the above code sequence. The top part shows portions of the specification's non-deterministic state machine that correspond to the code above. It shows that one of two sequences can exist: either P1 wrote 1 and then P2 wrote 2 or vice versa—write atomicity is maintained in both cases. The portion of the figure that is drawn in bold illustrates the corresponding implementation's state machine (which is deterministic). The bottom part of the figure shows the state of the relaxed scoreboard with respect to the same code: The scoreboard identifies two writes and marks both values as possible answers. This corresponds to the state machine being in either state $Init$, A, B, C or D. When the first read is reported to the scoreboard, the scoreboard deduces that the design can be in either state A, B or D, and when the second read is reported, the uncertainty window collapses to a single allowed value. In this example, the scoreboard will immediately identify the read by processor 5 as an error since the returned value is no longer in its list of allowed values for that address. It is important to note that a trace analysis checker that does not use temporal information will not identify this set of traces as erroneous. Without the time information, the read by P5 could be assumed to have taken place before the read by P4.

4.2. Transaction Isolation in TCC Memory Model

To provide another example of how to convert protocol rules into checks and updates we examine Transactional Memory. *Transactional coherence and consistency* (TCC) is a memory model that has been proposed to simplify the development of parallel software [1]. In TCC, the programmer annotates the code with start/end transaction commands, and the hardware executes all instructions between these commands as a single atomic operation. If a transaction conflict is detected, such as one transaction updating a memory word read by another, the latter transaction is considered as violating and must be re-executed.

In order to track the state of a TCC system, the relaxed scoreboard must be able to determine when transactions begin, commit or abort. Fortunately, as part of the software-hardware contract, the timing of these events is determined by a TCC runtime system using special marker instructions to signal the state of a particular processor and transaction. The relaxed scoreboard can use this information for its own operation. The scoreboard's data structure also needs the ability to keep track of multiple transactions simultaneously; all written values must be kept and associated with their initiating processor until the time of commit. Similar to the example discussed in Section 4.1, the scoreboard does not need to know the exact timing of the events in order to check for end-to-end properties. For example, the scoreboard does

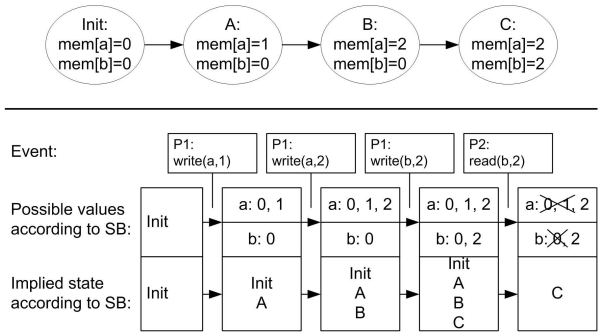

Figure 3. Total Store Order example.

not know when a store becomes visible to other processors; it assumes that it happens sometime during commit.

The *Transaction Isolation* property means that when a processor is executing a transaction, the transaction's intermediate state is hidden from other processors in the system. They can only observe the transaction's state before a transaction start or after a transaction commit. As in the previous example, in order for the relaxed scoreboard to check for transaction isolation, we convert the protocol property into an update:

> PROTOCOL PROPERTY: A transactional store cannot be observed by other processors unless the transaction commits successfully.
> UPDATE: When a store is issued during a TCC transaction, leave it as invalid for all but the issuing processor. Upon observing transaction commit, validate it for other processors as of commit time.
> CHECK: A load can only return a value which was valid at some point between its start-of-interval and end-of-interval

The following sequence is an example of a violation of transaction isolation property:

Time	P1	P2
10	START_TRANSACTION	START_TRANSACTION
20	WR(a,1)	nop
30	nop	RD(a,1)
40	START_COMMIT	

* Assume MEM[a]=0 at time 0
** Assume read operations complete in less than 10 cycles

In this example, the scoreboard would not allow the value 1 for P2's load, as it has not observed the commit event from P1. Thus on P2's load it would raise an error.

4.3. Store Ordering in a TSO Memory System

A design that implements Total Store Ordering (TSO) ensures that all stores from a processor complete in program order, as seen by all processors (this is different from weak consistency, which makes no guarantees about the ordering

of stores to different addresses). To facilitate verification of store ordering, we add another update to the scoreboard:

> PROTOCOL PROPERTY: All stores by a processor must commit in issuing order, as seen by all processors
>
> UPDATE: When a store by processor Pi is observed by a processor's load, all previous stores by Pi to all addresses should be marked as committed. If more than one store by Pi exists for a given address, mark the latest as commited, and invalidate older ones.

To analyze the scoreboard actions when used for TSO, consider the following example of a TSO violation:

Time	P1	P2
10	WR(a,1)	nop
20	WR(a,2)	nop
30	WR(b,2)	nop
40	nop	RD(b,2)
50	nop	RD(a,1)

* Assume MEM[a]=MEM[b]=0 at time 0
** Assume read operations complete in less than 10 cycles

Figure 3 illustrates the operation of the relaxed scoreboard for the above code sequence. The top part shows the change in state of the machine given the TSO memory model. The relaxed scoreboard cannot deduce the exact state, but it can deduce a set of possibly correct values, as illustrated in the bottom part of the figure. By observing the three writes one can deduce that the state is $Init$, A, B or C. When the first read is reported to the scoreboard, the design can only be in state C, therefore the uncertainty window collapses to a single allowed value. Finally, as the second read by processor 2 is observed, the scoreboard will immediately identify it as an error since the returned value is no longer in its list of allowed values.

The three previous examples showed how it is possible to write simple updates and checks for the scoreboard to verify different aspects of memory implementations[1]. In the following two sections, we analyze the complexity and completeness of this methodology.

4.4. Algorithm Complexity

In addition to the simulation length and the number of active processors, two main factors determine the computational complexity of the scoreboard. The first is the number of addresses used during simulation. A smaller address range can increase the chance of race conditions (increases the uncertainty), and on the other hand, a wide address range may increase the amount of relevant dependent operations. The second factor is the latency of operations. Long latency stalls (e.g. due to cache misses) can increase the uncertainty regarding the possible returned values for a given load operation. While sparse and bounded address ranges can not co-exist, long latency stalls constitute the majority of the complexity. In fact, these are the "interesting" cases in the system since these cases imply that the system is performing some arbitration and synchronization.

Assume that a specific run contains N instructions from P processors thus resulting in $O(N \cdot P)$ calls to the scoreboard. Each loaded value must be compared to $O(C \cdot P)$ values (*check* phase) and then update $O(C \cdot P)$ values (*update* phase), where C is a constant property of the system and represents an upper bound for the number of possible outstanding operations per processors. In the worst case, the property C does not exist (i.e. $C == N$), and thus the total complexity of the algorithm described is $O(N^2 \cdot P^2)$. However, in practical systems, one can put an upper bound on outstanding operations, or at least an upper bound on the time it takes to complete an operation. Using this upper bound as one of the update rules, the scoreboard can collapse old unused data sets, thus significantly reducing the runtime and memory footprint. The complexity is reduced to $O(N \cdot C \cdot P^2)$.

4.5. Algorithm Completeness

The problem of Verifying Linearizability is an NPC problem with respect to the number of processors in the system P (i.e., there is an algorithm which is polynomial to the address range and to the length of the trace, but not to P). However, the relaxed scoreboard as described thus far, is a polynomial time checker, implying that it is not complete. A checker of a consistency model is considered *complete* if it can completely prove or disprove any set of traces to comply with the relevant consistency model. For example, the first version of the TSOtool was an incomplete checker, since it could not identify all erroneous traces [2]. A later study by the authors improved the algorithm to be complete [28].

The reason the relaxed scoreboard is not complete is it never revisits a previous decision it has made. This can be a problem when, as happens in real systems, reads can overlap and complete in varying amount of time. We give an example to demonstrate this case, in which loads are no longer assumed to complete in less than 10 cycles: P1 reads the value 2, which implies that 2 was committed, and should be visible to all processors at cycle 40. P3 starts a read at cycle 50, and the value returned is 3, which implies that the write of value 3 committed after the write of 2. This, in turn, implies that the read by P2 (which happens after the store of value 3 to maintain program order by P2) must not return the value 1 (it should return 3). Yet, since the start time-stamp of P2's read is set at a time for which the value 1 may have been valid, a relaxed scoreboard will not flag it as error.

1. The interested reader can find more implementation details at http://www-vlsi.stanford.edu/smart_memories/RSB/index.html

Time	P1	P2	P3
10	WR(a,1)	WR(a,3)	nop
20	WR(a,2)	RD(a)	nop
30	RD(a)	stalled	nop
40	→2	stalled	nop
50	nop	stalled	RD(a)
60	nop	stalled	→3
70	nop	stalled	nop
80	nop	→1	nop

* Assume MEM[a]=0 at time 0
** Assume the system maintains write atomicity

The example demonstrates a case in which the information at cycle 60 (the returned value 3 to P3) has implications on the temporal information and order of events that were seen in previous cycles (i.e., the write of value 3 must have committed after the beginning of the read by P1, and consequently so did the read by P2).

In order to fully account for cases in which later information impacts the validity of already verified loads, the scoreboard would have to roll back and rerun portions of the checks and updates. The number of reruns may (in worst case) be an exponential function of the number of processors that interact. However, empirical results, based on random test patterns, show that less then 0.05% of transactions are hazards for this type of rollback scenario[2].

To completely verify linearizability of a system, three possible extensions are suggested:

1) Enable rollbacks: Enable the scoreboard to rollback, and redo portions of checks and updates based on new information. Potentially, this may slow down the simulation. While the worst case is NP with respect to the number of processors, for practical systems it is unlikely to be this bad.
2) Complementary post mortem check: Use the relaxed scoreboard as a partial checker, but complement it with a complete post-mortem (exponential time) checker such as the one described by Gibbons and Korach [25]. However, this would require dumping the huge traces and their temporal information.
3) Turn to "white box": Since the scoreboard enables incremental additions, once the design is mature enough, key information can be used to better determine the time in which instructions committed.

One should also note that the relaxed scoreboard can be used to verify a subset of the consistency model properties (thus obviously incomplete). Our experience shows that even in those cases where not all axioms are translated into checks and updates, it was useful in finding elusive bugs that could not have been found using race-free or self-checking test vectors. The next section describes the results of using the relaxed scoreboard to verify several memory models.

2. The scenario depicted here is the only type of undetected error that we could find in order to prove incompleteness. However, there may be other scenarios for which a relaxed scoreboard may fail to identify an error.

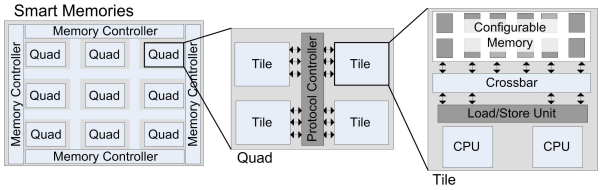

Figure 4. Smart Memories Configurable Architecture.

5. Evaluation

We applied the relaxed scoreboard to the verification of the Stanford Smart Memories chip multiprocessor architecture [26] and evaluated its effectiveness based on the number and complexity of the design errors that it could reveal, in addition to the impact that it had on the overall verification/simulation environment.

5.1. Scoreboard Design for Smart Memories

Smart Memories is a modular, reconfigurable multiprocessor architecture that supports multiple programming models, including conventional shared memory with cache coherency (CC), stream programming [29] and transactional memory (TCC) [1]. The system consists of Tiles, each with two simple VLIW cores, a number of reconfigurable memory blocks, and a crossbar that connects them to each other and to the outside world. Tiles are grouped together to form Quads. Each Quad has a configurable protocol controller with a generic network interface for communicating with the rest of the system. Quads are connected via a mesh-like network, which also connects them to memory controllers and off-chip memories. Figure 4 shows a block diagram of the architecture.

When configured as a shared memory CMP, the Smart Memories system implements a hierarchical version of the MESI protocol. The shared memory consistency model [27] is a variation of Weak Ordering (WO), which maintains write atomicity. Processors are allowed to issue non-blocking writes to memory, which can be overlapped and completed out of program order. On the other hand, reads are treated as blocking operations and stall the processor until the data is returned. When configured as a transactional memory CMP, the Smart Memories system implements the TCC protocol [1], briefly described in Section 4.2.

A relaxed scoreboard was used to help verify the design. It was implemented as an object oriented programming class using OpenVera. Vera's Aspect Oriented Programming (AOP) capability was leveraged to connect the scoreboard into the existing verification environment, which already included code to monitor key interfaces. For cache coherency, the scoreboard's data structure contained an associative array of queues, one queue per writable address in the system.

For TCC, the scoreboard also maintained a queue for each processor, containing pointers to the transaction's read and write sets, and flags to indicate the state of the current transaction. Checks and updates, a superset of those shown in Sections 4.1 and 4.2, were written based on the protocol rules of [27] and [1] respectively[3].

Due to the scoreboard's black-box nature, it was usable without modification across 25 different cache-coherent configurations and 15 different TCC configurations. Moreover, it was applied to both single-quad (8 processors) and four-quad (32 processors) testing environments, where the only difference was the number of monitor instances that fed the scoreboard with information.

In order to further demonstrate the flexibility of the system, we also configured the scoreboard to check for Total Store Ordering. This required writing a single new update for the scoreboard, only 100 lines of Vera code. While this initial version was correct, the data structure was rather inefficient, so we also introduced a second lightweight data structure to track outstanding stores and a short update to populate it.

5.2. Quality of Results

We evaluate different verification methodologies by considering the complexity of the design errors each can reveal, as well as the cost of applying them. This cost can be measured in terms of both the overheads incurred when simulating the DUT, and the difficulty of integrating the methodology into the verification environment. Table 1 compares and contrasts the relaxed scoreboard with other memory system verification schemes across a number of dimensions. For comparison purposes, all schemes were considered as if implemented for sequential consistency, and bold text is used to indicate the most desired value.

To demonstrate error cases that can be revealed by using a relaxed scoreboard, Table 2 summarizes classes of errors that the relaxed scoreboard found in the Stanford Smart Memories design. One should note that the errors described in Table 2 were found after multiple runs of self checking diagnostic tests were used to identify "simple" errors.

Given the complexity of the errors described in Table 2, and the fact that some of the errors would have evaded previous verification methodologies, we found the relaxed scoreboard to be a useful debugging tool and another weapon in the arsenal against memory system errors. Using the scoreboard simplified test generation, as tests no longer had to be entirely self-checking. This allowed the test suite to include truly random tests, where before it was limited to tests with only false sharing or strictly synchronized accesses between processors. The random tests revealed subtle bugs, such as erroneous ordering of back-to-back stores to the same address. Moreover, the efficiency of the self-checking tests increased because the relaxed scoreboard was able to detect intermediate errors, even when the final output was correct.

For Cache Coherence we were able to detect many problems related to memory ordering and corruption. These errors existed in many parts of the design: the Cache Controller, Load Store Unit, Memory Controller, and in configuration of our programmable hardware. In the TCC mode, the scoreboard was able to detect errors such as faulty commits and violations. These included a very subtle case where a transaction violated unnecessarily due to an error in the manipulation of state bits. Since this was a performance error, it would have otherwise remained undetected by any self checking test. Another type of error that the scoreboard detected in TCC mode was runtime sequencing problems (software library errors). These included cases in which two start transaction markers were observed without an end/violate transaction marker in between.

Some transactional memory proposals include more advanced features which are not supported by the Smart Memories hardware used to benchmark the relaxed scoreboard. Future work in this area can include extending the relaxed scoreboard checks/updates to test commit handlers, nested transactions, etc.

As was mentioned in Section 4, we were pleased that the same basic approach served to validate very different memory system models, as well as different implementations of each model. This generality was enabled by the flexible and incremental addition of checks and updates

5.3. Performance and Overheads

Several hundred diagnostic, application and constraint random vector generators were used to generate memory transactions for the verification of the Smart Memories design. For the analysis shown here we concentrate on a subset shown in Tables 3 and 4, which comprise over half a billion memory accesses. All simulations were performed on the same machine specification: A 3.40GHz, 8GB RAM, Dual Processor Intel Xeon, running CENTOS4.

We assessed the overhead related to the scoreboard's implementation as well as the overhead in simulation time when activating it. As the most basic metric, we compared the amount of code associated with the scoreboard to that of the rest of the verification environment. It turned out to be surprisingly small: the entire scoreboard code is about 3000 lines, which is less than 5% of the total of the Smart Memories verification environment. Measuring simulation time and memory footprint yield the results presented in Tables 3 and 4. The numbers in the Time and Mem columns represent the ratio of CPU runtime and memory footprint (respectively) between the run with scoreboard to a run

3. The source code of the relaxed scoreboard implementation for Smart Memories can be found at http://www-vlsi.stanford.edu/smart_memories/RSB/index.html

Table 1. Qualitative comparison of the relaxed scoreboard methodology to other memory system validation schemes

Attribute	Self checking tests	Gold model	TSOtool-like (base) [2]	TSOtool-like (complete) [28]	Relaxed Scoreboard
Correctness Criteria	VL (races avoided)	**VL**	VSC	VSC	**VL**
Completeness	No	**Yes**	No	**Yes**	No
Algorithm Complexity	NA	NA	**Polynomial**	Exp. (worst case)	**Polynomial**
On-The-Fly Capable	No	**Yes**	No	No	**Yes**
Post-Mortem Capable	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Black Box / Reusablity	No	No	**Yes**	**Yes**	**Yes**
Incremental Additions	**Yes**	No	No	No	**Yes**

Table 2. Classes of errors found by the relaxed scoreboard (LSU=Load-Store Unit; PC=Protocol Controller; MC=Memory Controller; Net=Network)

Error Description	Error Location
Cache Coherency and Consistency	
Program order violation: processor writes new value, then reads the old value from the same address	LSU
Program order violation: later store value is overwritten by older store from the same processor	LSU
Soft processor stall doesn't work	LSU
Back-to-back loads from the same processor return different values	LSU
Coherence violation: load from a different processor returns zero instead of valid data	PC
Instruction fetch returns incorrect value	LSU
Instruction fetch returns incorrect value	Boot sequence
Load returns a no-longer valid data	MC
Load returns incorrect data	LSU
Load returns incorrect data	MC
Load returns incorrect data	PC
Load returns incorrect data	Compiler
load returns X during boot/setup	Boot sequence
Synchronized load returns incorrect data	MC
Synchronized load returns incorrect data	PC and PC config
Transactional Coherency and Consistency	
START_TRANSACTION command called twice	TCC runtime
Missing COMMIT command	TCC runtime
Unnecessary TCC violation	PC
TCC coherent load returns stale data	Net Config
TCC committed value is lost	Net Config
TCC transaction is not violated, missed dependence	PC
TCC coherent load returns wrong data	PC

Table 3. Test Benches and Runtime Statistics on an 8 and 32-Processor system with CC.

Test Bench	8 CPUs #	8 CPUs Time ratio	8 CPUs Mem ratio	32 CPUs #	32 CPUs Time ratio	32 CPUs Mem ratio
Fast Fourier Transform	10.5M	1.29	1.86	52.5M	1.30	2.20
LU matrix decomposition	30.7M	1.49	4.87	79.5M	1.19	3.00
Matrix multiplication	18.3M	1.29	3.24	20.5M	1.33	2.56
Merge sort	6.2M	1.15	3.24	8.3M	1.08	2.19
Bitonic sort	19.6M	0.82	4.91	21.9M	1.24	2.58
Random* loads and stores	0.25M	1.019	2.62	1M	1.03	5.33

* Random test and restricted address range to induce race conditions

Table 4. Test Benches and Runtime Statistics on an 8 and 32-Processor system with TCC.

Test Bench	8 CPUs #	8 CPUs Time ratio	8 CPUs Mem ratio	32 CPUs #	32 CPUs Time ratio	32 CPUs Mem ratio
Fast Fourier Transform	6.4M	1.24	4.04	20.2M	1.09	2.53
LU matrix decomposition	19M	1.64	8.69	31.7M	1.53	4.97
mp3d	63.8M	1.2	1	109M	1.14	3.3
Flipper*	29M	1.21	1.86	31.2M	1.18	1.56

* Random test and restricted address range to induce race conditions

without the scoreboard. The # column denotes the number of memory accesses, including all reads and writes that were performed in each test, in byte granularity. The average overhead measured in verifying cache coherency (Table 3) was 19% in runtime and 201% in memory footprint. For TCC (Table 4), the average overheads were 28% and 250% respectively. These results show that the scoreboard runtime overhead is reasonable, especially in light of the fact that simulations are most likely to be deployed on multiple servers as a regression suite, and should an error be found, the relaxed scoreboard would immediately halt the simulation and report it. The memory overhead is a consequence of the relaxed scoreboard's data structure, which holds a list of timestamps and other information for every memory location that the test uses. This overhead was hardly ever an issue, and therefore no significant effort was made to improve it. One can decrease the execution overheads by memory management techniques, or by constructing the scoreboard using a lower level programming language in place of Vera, such as C or C++.

Finally, as mentioned in Section 4, a useful measure of the effectiveness of the relaxed scoreboard is the size of the set of possibly correct values (referred to as the uncertainty window). Figure 5 shows the size of the uncertainty window for several test-runs on 32 processors. Each subfigure shows a histogram of the number of possibly allowed values for every simulated load in the test. Figure 5.a shows that for a random test, limited to 10 addresses for all 32 processors, the average uncertainty is 35 values and never exceeds 81. Figures 5.b and 5.c show that as there is less contention for

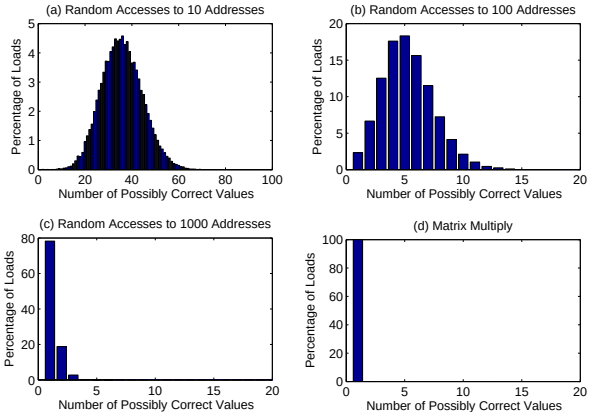

Figure 5. Histograms of uncertainty for applications running on 32 processors.

the addresses there is also less uncertainty. Figure 5.d shows the results for a real application, in which there is very little uncertainty, and the scoreboard only occasionally needs to maintain more than one value.

One conclusion we draw from Figure 5 is that the size of the set of possibly correct values is always bounded, even when the test focuses on a very small address range. In fact, for a self-checking diagnostic such as matrix multiplication (Figure 5.d), a relaxed scoreboard behaves almost identically to a golden reference model. In addition, it is important to note how much more stressful a random test with true sharing is (Figure 5.a) in comparison to a deterministic self-checking diagnostic (Figure 5.d). The latter rarely induced any races. This emphasizes the initial motivation for creating an end-to-end reference model that can be used with random tests, for a more efficient verification environment.

6. Conclusions

The advantages of having a scoreboard or a golden model for validation are well known as are the difficulties of creating this model. One approach to mitigate these difficulties is to allow a degree of flexibility in the reference model. Leveraging this flexibility and non-determinism to construct a *relaxed scoreboard*—a reference model that tracks a tight set of possible "right" answers and is therefore decoupled from implementation decisions—greatly simplifies the construction of the model. Since the possible set is almost always small, it does not change its effectiveness in finding errors in the design implementation.

This relaxed scoreboard creates the perfect framework for building chip multiprocessor memory checkers, since one can incrementally and almost directly convert memory ordering or protocol properties into update rules for the scoreboard. This allows one to create a scoreboard early in the design, with little effort, and refine it as the design progresses. We used it to validate a number of different memory models, including TCC, in the Smart Memory verification effort, and it was very effective in detecting errors in our design.

Compared to other memory validation methods, the relaxed scoreboard combines the ease of use and the ability to leverage timing information as in the traditional scoreboard, with the implementation independence and ease of creation of TSOTool [2]. This approach enables users to easily create a memory system checker that is customized for their memory model, runs concurrently with simulation, and yet remains decoupled from the implementation details. Moreover, the resulting checker can be made tighter than a checker for logical correctness, since it can verify both logical (VSC) and temporal (VL) correctness.

Acknowledgment

The authors would like to thank the rest of the Stanford Smart Memories faculty and team, including Don Stark, Kyle Kelley, Zain Asgar, Wajahat Qadeer and Rehan Hameed, for their assistance and invaluable comments. In addition, a special thanks is extended to Professor David Dill for his help in understanding the formal aspects of the Relaxed Scoreboard, and for offering fresh perspectives.

References

[1] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *International Symposium on Computer Architecture (ISCA '04)*, 2004, p. 102.

[2] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "Tsotool: A program for verifying memory systems using the memory consistency model," in *ISCA '04*, 2004, p. 114.

[3] A. Saha, N. Malik, B. O'Krafka, J. Lin, R. Raghavan, and U. Shamsi, "A simulation-based approach to architectural verification of multiprocessor systems," *Computers and Communications, 1995. Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on*, pp. 34–37, 28-31 Mar 1995.

[4] J. Bergeron, *Writing testbenches: functional verification of HDL models*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[5] T. Fitzpatrick, A. Salz, D. Rich, and S. Sutherland, *System Verilog for Verification*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[6] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip verification: methodology and techniques*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[7] *OpenVera*, Synopsys, http://www.open-vera.com/.

[8] *Specman Elite - Testbench Automation*, Cadence, http://www.verisity.com/products/specman.html.

[9] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for System Verilog Assertions*. New York, NY, USA: Springer Science+Business Media, Inc., 2005.

[10] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "Focs: Automatic generation of simulation checkers from formal specifications," in *International Conference on Computer Aided Verifi cation*, 2000, pp. 538–542.

[11] A. Meixner and D. Sorin, "Dynamic verification of sequential consistency," *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pp. 482–493, June 2005.

[12] ——, "Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures," *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pp. 73–82, 2006.

[13] F. Pong and M. Dubois, "The verification of cache coherence protocols," in *Symposium on Parallel Algorithms and Architectures*, 1993, pp. 11–20.

[14] ——, "A new approach for the verification of cache coherence protocols," *Transactions on Parallel and Distributed Systems*, vol. 6, no. 8, pp. 773–787, 1995.

[15] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *International Conference on Computer Design*, 1992, pp. 522–525.

[16] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu, "Checking cache-coherence protocols with tla+," *Formal Methods in System Design*, vol. 22, no. 2, pp. 125–131, 2003.

[17] S. Qadeer, "Verifying sequential consistency on shared-memory multiprocessors by model checking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 8, pp. 730–741, 2003.

[18] M. Plakal, D. Sorin, A. Condon, and M. Hill, "Lamport clocks: verifying a directory cache-coherence protocol," *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pp. 67–76, 1998.

[19] A. Condon, M. Hill, M. Plakal, and D. Sorin, "Using lamport clocks to reason about relaxed memory models," *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, p. 270, 1999.

[20] S. Tasiran, Y. Yu, and B. Batson, "Linking simulation with formal verification at a higher level," *IEEE Des. Test*, vol. 21, no. 6, pp. 472–482, 2004.

[21] K. Lepak, "Exploring, defining, and exploiting recent store value locality," Ph.D. dissertation, UNIVERSITY OF WISCONSIN, 2003.

[22] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun, "Testing implementations of transactional memory," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2006, pp. 134–143.

[23] E. Gibbons, P.B.; Korach, "The complexity of sequential consistency," *Symposium on Parallel and Distributed Processing*, pp. 317–325, 1992.

[24] J. Cantin, M. Lipasti, and J. Smith, "The complexity of verifying memory coherence and consistency," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 7, pp. 663–671, 2005.

[25] P. Gibbons and E. Korach, "Testing Shared Memories," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 1208–1244, 1997.

[26] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *ISCA '00*, 2000, pp. 161–171.

[27] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[28] C. Manovit and S. Hangal, "Completely verifying memory consistency of test program executions," *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pp. 166–175, 11-15 Feb. 2006.

[29] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media processing with streams," *IEEE Micro*, vol. 21, no. 2, pp. 35–46, 2001.