Model Checking Memory Consistency and Cache Coherence

Hernan Palombo
University of South Florida
Department of Computer Science and Engineering
Tampa, FL
hpalombo@mail.usf.edu

***Abstract*—In this paper we study model checking techniques for memory consistency and cache coherence in multiprocessor systems with shared memory. We review the most common memory consistency models and we analyse the properties that define cache coherence by looking at different protocols. We describe formal verification methods and some of the complexities involved in model checking practice. We describe existing model checking tools that are normally used in the field and present some case studies of model checking verification applied to cache coherence and memory consistency using the tools available. Finally we discuss the challenges and opportunities for future development in this area.**

I. Introduction

Memory organization has been a principal topic in system architecture since the early years of computing [1]. The introduction of multiprocessor systems in the late seventies brought new challenges that motivated the development of memory consistency models and protocols to ensure cache coherence. Early papers demonstrate the consistent interest in the topic [10] [19]. In more recent years, the popularity gained by multi-core architectures and the possibility to scale the number of processors has generated a new set of challenging problems in cache communication protocols that still need to be solved. New models have emerged to cope with larger systems that implement more complex memory structures. In order to be able to keep up with the exponential growth in processing power that has been a trend in industry for decades, chip manufacturers have been increasing the number of cores on a single chip and, at the same time, more multi-core processors are being added to every system. Therefore, the problem of maintaining cache coherence and consistency using correct protocols becomes key for future development.

An equally important problem is establishing consistency models and coherence protocols correctness. The main reason is economical [29], the cost of fixing an error in hardware design increases exponentially as the development process advances to a later stage. For this reason, hardware companies are spending a greater percentage of the development budget in testing and verification. Although the use of simulation is still common in practice, is not sufficient as it is not as nearly as effective as verification for discovering errors that are deep in the structure of a system design. For this reason, verification in hardware design has been an active topic of research for many years [20]. Recent advancements in this area have overcame previous problems that in the past made verification impractical. Verification is a great example of how formal theory of computation can be applied to solve complex practical problems.

In this paper, we focus on the application of model checking techniques to verify memory consistency and cache coherence. First, we introduce some background knowledge on cache coherence and its relation to memory consistency models. Then we categorize general methods of verification, different types of model checking, and we explain how these are applied. We examine some case studies and point out interesting results. Finally, we discuss some of the challenges of cache coherence verification and current trends in industry and research.

A. Introduction to Cache Coherence and Memory Consistency

In shared memory systems, organization of memory modules can be categorized by physical location as being either centralized (also called symmetric) or distributed. In centralized models (Figure 1), memory units are at a single location, connected to an interconnection network, which communicates with each processor's level 1 cache. In distributed models, on the other hand, each processor may have its own memory module, which is then connected to the interconnection network. In all cases, memory access should be -at least to some degree-synchronized to ensure consistency. In this paper, we focus on centralized memory architectures because they are the most widely implemented architecture in practice.

There are two aspects that must be considered when studying multiprocessor architectures with shared memory. The first is what values should be returned by read operations, and the second aspect defines the order in which read and write operations execute. The first problem is addresses through the implementation of a cache coherence protocol. The second issue is resolved by establishing a memory consistency model.

It is the goal of systems with multiple processors to achieve better performance through parallelism. In order for this to happen, memory access time should minimized to avoid bottlenecks. It has been proven that, to avoid performance penalties related to memory operations, each processor must have its own level 1 cache. The main motivation behind implementing a cache coherence protocol is that multiple caches may contain different versions of the same data, that is to say, data may be incoherent. A cache coherence protocol ensures that data reflects accurate values. Special attention must be given to write operations, since these can cause the values at other caches to become stale and inaccurate. At the same time, when multiple read and write operations exist within a program, the system must ensure that these operations are executed in the expected order. This is usually known as maintaining memory consistency.

Memory consistency means that loads and stores should follow the order of execution specified in the original program. Naturally, this requirement is easy to implement in single processor architectures. On the other hand, multiprocessor architectures may not satisfy this requirement at all times. The resulting order of execution will depend on the memory consistency model.

First, let us examine the case of multiprocessor systems that implement a sequential consistency model. These systems aim to maintain the same order of execution as if the program was being executed in a single processor system. In this model, sequential program order is an invariant that must be satisfied throughout the different caches at all moments during program execution. As we have described earlier, achieving this requirement in single processor architectures is simple because, for the processor, there is only one view of memory at any given time. In multiprocessor architectures, however, the problem becomes of increased complexity since each processor has its own view of memory and each cache must have a way to communicate between each other to ensure both coherence and consistency.

If we generalize multiprocessor architectures as concurrent systems, we can define coherence and consistency as requirements of their communication protocol. For verification purposes, coherence and consistency can be expressed a set of correctness properties that need to be satisfied by a system model.

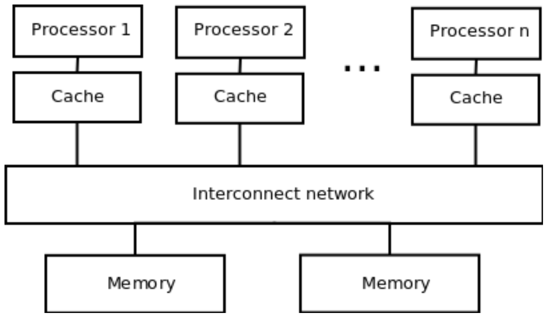

Fig. 1. High-level view of a multiprocessor with centralized memory

II. Cache Coherence Protocols

In multiprocessor systems, each processor's level 1 cache may have a different value of a program variable. For this reason, it is important that the system maintains data coherence between the local caches.

Cache coherence must satisfy two correctness properties [4]:

1) Only one cache must be able to write (also known as write serialization) and multiple caches must be able to read at any given time.
2) The data in each of the caches must be accurate and reflect global updated values (the consistency property).

At a high level view, coherent systems are composed of a memory controller and a set of cache controllers. System events can be formalized into a finite set of actions that each of the caches takes in reaction to other events. Furthermore, we can define a set of states for each of the memory blocks represented in the controllers. Sorin et at. [24] introduced the idea of representing a block as having one of three states:

- Exclusive (read/write access)
- Shared (read access)
- Invalid

This state representation can be used in both types of protocols (directory and snoopy). Figure 3 shows a finite state automaton representing the states of a memory block and the corresponding transitions in a cache coherent system.

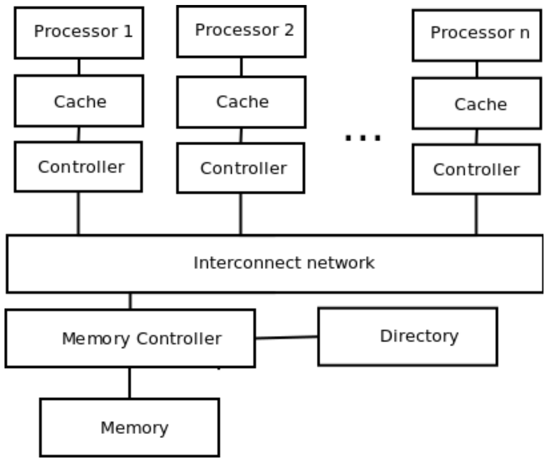

Fig. 2. Directory cache coherence protocol

There are two general mechanisms for ensuring cache coherence: directory and snoopy protocols. Both methods are based on maintaining a record of each cache block's state in the system. In the directory protocol, information is kept at a centralized location and all caches must communicate to the directory when there is an operation that requires system coherence enforcement. Distinctively, in snoopy protocols each cache keeps the information about its blocks, and maintains coherence by snooping in a shared bus to find out if information has changed in one of the other caches.

A. Directory Protocols

In directory protocols, every cache shares the state of its data to a centralized directory that keeps track of changes and either updates or invalidates data entries when a cache modifies values [10]. When a processor needs to load data from main memory, it sends a request to the directory, and the directory will take care of the necessary synchronization steps. The directory must not only keep track of the state of each block but also of which caches have a copy of the block. This is done by keeping a bit array of sharers of that block [8].

Let us examine the different actions that take place at each state of the system. As we described earlier, a cache block can have one of three states, exclusive, shared, or invalid. The exclusive state gives read/write access to the cache that owns the block, and the data may be changed at any time. For this reason, this state can also be referred as modified [10]. Furthermore, a block that is not cached would be considered invalid. The actions that can be sent from the cache controllers to the directory are either read or write misses, or a data write-back request.

We study the case of an invalid block first. The only two possible requests that such a block can make to the directory happen on a read or write miss. In both cases, the data is sent from memory, and the requester becomes a sharer of that block. In the case of a read, the block is set to shared. In the case of a write, the block is set to modified, meaning the block's write access is exclusive to the cache containing it.

When the directory receives a read or write request from a cache for a block that is in the shared state, similar actions occur. If the request is a read miss, the data is sent to the requesting cache and the requester is added to the sharers list for that block. In the case of a write miss, all the current sharers must be notified that the block now becomes invalid. All caches are removed from the sharers list and the new requester becomes the only sharer, and the block is set to exclusive at that particular cache.

When the directory receives a request for a block that is in the exclusive state, three things can happen. On a read miss, the owner of that block is notified so that the block can be set to shared at the residing cache and the data sent back to the directory. Then the directory sends the data to the requester cache and adds it to the list of sharers. Similarly, in the case of a write miss, the original owner of that block is removed from the sharers and notified forcing a write back and a change to invalid state at the residing cache. The requester becomes the new owner of the block's write access, so it is added to the sharers array and its block is set to exclusive. Finally, on a write back the owner of a cache block sends the data to the directory so that it can be written back to memory, and the cache is removed from the sharers array.

Our model of a directory protocol can be represented as a finite state automaton 3, which makes it apt for model checking and, at least intuitively, quite simple. In the verification section, however, we explore more in depth some of the issues that have been found when trying to verify cache coherence protocols.

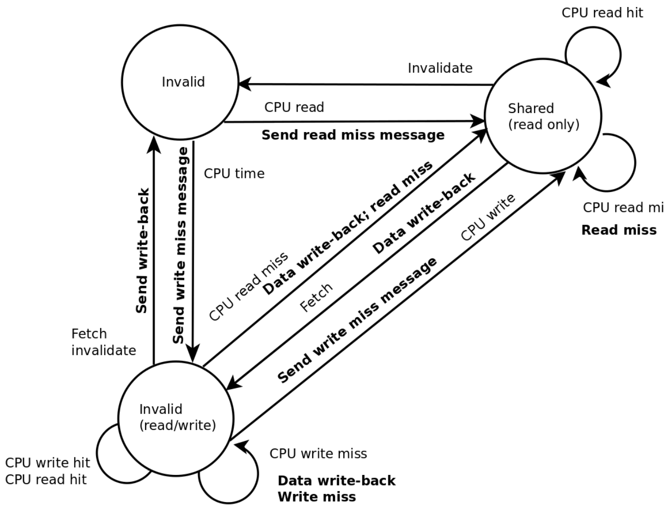

Fig. 3. MSI Cache coherence protocol [3]

B. Snoopy Protocols

Snooping protocols are based on the same concept of assigning a state to each of the cahe's block. However, they exhibit a different behaviour than directory protocols. The control logic is parallel and each cache is in charge of keeping track of the validity of its own copies of memory. When a cache modifies a value stored in memory, it sends a broadcast message with a reference to the information that needs to be replaced. All caches must be constantly snooping on a bus to maintain their copies up-to-date.

There are two kind of policies for maintaining data coherence, write-update and write-invalidate. The former one acts upon snooping caches by updating the values that were modified by the cache that owns the block. The latter one, and most common [23], simply informs the snooping caches that the value became obsolete by setting its state to invalidate.

For instance, let us examine a sample execution on a write-invalidate snoopy protocol represented on figure 4. In this example, (1) Processor 1 issues a write action on variable x. Processor 1's cache controller puts an invalidate request for variable x on the bus. (2) Processors 2,3...n and memory controller see the message and respond with an ACK. (3) Processor 2 issues a read on variable x. Local cache's value is dirty so processor 2's cache controller puts a request on the bus. (4) Value of x in memory is dirty. Therefore, memory controller issues a request for processor's 1 value. (5) Processor 1's cache controller sends the updated value of x to memory. (6) Memory controller sends x to processor 2's cache by placing it on the bus.

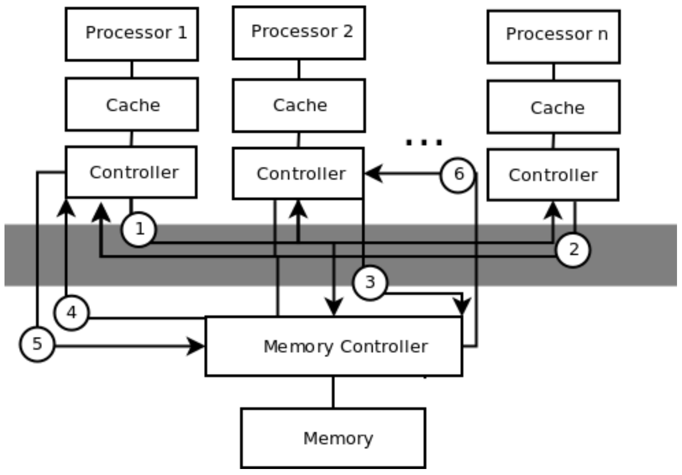

Fig. 4. Example execution of a snoopy protocol

With the advent of multi-core processors, additional complexity and new protocols have emerged. Combinations of snoopy and directory protocols have been designed. Usually, a directory protocol is used for on-chip interconnects, while snoopy protocols are in charge of communication between the different multi-core chips.New protocols based on maintaining a token ring have gained popularity [11]. Furthermore, systems that include multiple multi-core chips usually implement multi-layer cache coherence protocols. A major challenge is to be able to verify these protocols considering their increased complexity. Another challenge in verification of cache coherence protocols is being able to generalize methods for an unbounded number of processors.

III. Memory Consistency

Memory consistency models are usually categorized by the degree of sequentiality that they enforce. Accordingly, there are three main groups of memory consistency models: sequential consistency, total consistency order, and relaxed consistency models. Sequential consistency enforces that all read and write operations must be atomic and execute in succeeding order. Its behaviour is intuitive and its design the simplest to implement and verify. To avoid write stalls, a modification was introduced to the sequential consistency model to allow programs to continue with execution while a value is being written, and this is called total order consistency. This model improves performance by using write buffers with bypassing enabled. A relaxed consistency model is similar to total store order, although it allows more instruction reorderings to exploit data parallelism even further.

A. Sequential Consistency

As first formalized by Lamport [6], a sequential consistency model ensures that a program execution on multiple processors exhibits the same result as if the program was executed on a single processor in the sequential order specified by the programmer. For this to happen operations must be atomic. This means that a memory operation must not be able to start executing until the previous operation has completed. In this model, memory operations reordering is not allowed. This rule must be enforced for all possible combinations of consecutive memory operations: load after load, load after store, store after load, and store after store. Weaver and Germond [9] formalized sequential consistency safety properties as follows:

1) All local loads and stores must respect their program order. Let O be the set of operations and $<_p$ be the program order, the following must be true for all load and store operations $l, s \in O$:
   a) $l_i <_p l_j$
   b) $l_i <_p s_j$
   c) $s_i <_p l_j$
   d) $s_i <_p s_j$
2) All loads must get their values from the last store in global memory order. Let $<_m$ be the global memory order, then the following must also be true for all load and store operations $l, s \in O$:
   a) $l_i <_m l_j$
   b) $l_i <_m s_j$
   c) $s_i <_m l_j$
   d) $s_i <_m s_j$

Liveness properties may also be defined to prove that sequential execution completes for every program without livelock or starvation.

In multi-threading systems, a program can be thought of as a program thread or task, and the global memory order would be the order of the whole program, consisting of multiple threads. An interesting observation is that, in this context, sequential consistency limits the advantages of simultaneous multi-threading because stalls would have to be inserted to preserve memory orderings.

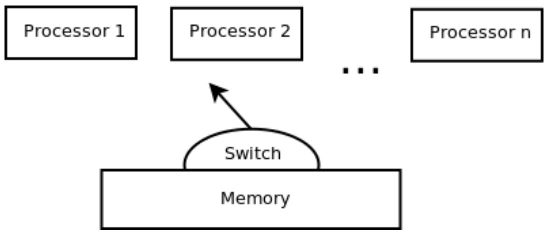

Fig. 5. Sequential consistency can be modelled as a switch [4]

A sequential consistency model can be visualized as a switch 5, in which memory access is given to only one core at a time [4].

The biggest advantage of sequential consistency systems is that it always follows a sequential order of execution at all times, which makes it easier to understand and implement. However, although sequential consistency model is simple, it does not take advantage of independent instructions that could potentially be executed in parallel.

As an example, let us examine the execution sequence presented in Table I. Since processors execution can be interleaved in any arbitrary order, the set of possible execution orders is all the permutations of the set of instructions that start with the initial instruction at any of the processors, and continues in an interleaved fashion until completion. In this example, execution can start with either the store(i) on processor X or store(j) on processor Y. Accordingly, the next two instructions can occur in the same or in a different order (processor X followed by processor Y, or processor Y followed by processor X). To maintain sequential consistency, the system must enforce the two invariants described earlier. First, program order must be preserved, that is to say store(i) must execute before load(i) on processor X, and store(j) must execute before load(i) on processor Y. The second constraint tells us that store(i) on processor X must complete before any other instruction can start execution.

TABLE I. LOAD AFTER STORE EXAMPLE

Processor X	Processor Y
store(i)	store(j)
load(i)	load(i)

B. Total Store Order

As we can see from the previous example, the greatest drawback of the sequential consistency model is that given a store operation, the processor has to wait for the store to complete before it can start executing the next operation. The total store order consistency model has been designed to deal with this limitation. Total store order architectures use first-in-first-out (FIFO) write buffers to allow the processor to continue execution while the store operation takes place. By allowing this relaxation of the sequential consistency model, two considerations must be taken into account. First, the store after store set of operations presents no hazard if the write buffer is of type FIFO, since the first store operation will always complete before the second one. Second, special attention must be given to load after store instructions. This sequence of operations could be a hazard and should be prevented. For instance, in the example given at Table I, processor X issued a store of variable i on memory location a, followed by a read of i. And processor Y also issued a load of i. If we allowed processor X to load the new value from the write buffer as it would normally occur on a single processor architecture, the value of i loaded on processor X and the value of i loaded on processor Y could differ. This could happen because processor Y would get the value of i from main memory (or level 2 cache), which would be obsolete. Instead, processor X would get the updated value of x from the write buffer. Therefore, the use of write buffers in multiprocessor systems require enabling bypassing. This would force processor X to get the value of variable i from main memory. Thus, consistency would be conserved.

We can summarize total store order memory consistency model as a variation of the sequential consistency model in which there is a store-load relaxation in program order:

1) All local loads and stores must respect their program order. Let O be the set of operations and $<_p$ be the program order, the following must be true for all load and store operations $l, s \in O$:
   a) $l_i <_p l_j$
   b) $l_i <_p s_j$
   c) $s_i <_p s_j$
2) All loads must get their values from the last store in global memory order. Let $<_m$ be the global memory order, then the following must also be true for all load and store operations $l, s \in O$:
   a) $l_i <_m l_j$
   b) $l_i <_m s_j$
   c) $s_i <_m s_j$

The synchronization of the store $<$ load sequence is left to be enforced in software, and the compiler must insert a FENCE instruction to force a specific order of execution.

Total store order has been used in SPARC implementations [9] [29] and x86 architectures [4] [33]. Despite its popularity, more recent architectures have implemented further relaxations of the consistency properties to allow greater performance through parallelism [34].

The main advantage of the total store order consistency model is that it allows for some performance improvement (by allowing load execution to continue before a write has completed), without adding significant complexity to the compiler. The disadvantage is, on the other hand, that out of order execution is not fully exploited to its potential and stalls will still occur.

C. Relaxed Models

To allow further optimizations, total store order models have evolved into more relaxed consistency models. Relaxed models exploit load/store reordering to achieve improved performance at the cost of increased programming complexity and less portability [7].

The relaxation of some or all of the consistency requirements determine the degree of a system's relaxation. The partial store order model extends the total store order by allowing write after write operations. By eliminating the write after write consistency requirement, hardware that implements this type of relaxed model can use non-FIFO coalescing buffers.

Weak consistency models extend this idea by allowing overlapping read operations. In this kind of fully relaxed model, all operations may execute out of order. Therefore, none of the consistency properties are enforced. Hence, any operations that require to be executed in a certain order must be specified in software through the use of FENCE instructions. Hardware only enforces consistency for operations specified between FENCE instructions. The cost of the performance improvement is an increased complexity to compiler implementations and verification efforts [7].

IV. Formal Verification

Formal verification of systems has been a growing area of research in recent years. The main purpose of formal verification is to prove a system's correctness properties. There are different ways this can be achieved. In general, the process of verifying a system starts by constructing a model of the current system and defining a set of correctness properties (a model of the specification) that must apply to the system implemented.

Formal verification can be classified in two general categories: theorem proving and model checking. Accordingly, the latter one can also be subdivided and categorized. A short introduction to each method follows.

A. Theorem Proving

Theorem proving involves defining formulas that represent a system and its specification axiomatically, and then using deductive methods of mathematics to prove the system representation's correctness. The problem with theorem proving is its complexity and some limitation to be automated, which makes it impractical in certain cases and not very popular in industry.

B. Model Checking

The term model checking has been coined in the early eighties by Emerson and Clarke [38]. In model checking, a system's implementation (or design) is first modelled in a structured language and then checked against a set of properties formally defined based on the system's specification. There are two types of propositions that can be specified, safety properties and liveness properties. A system is verified to be correct if and only if a model satisfies all the properties defined for that system. For completeness, all possible executions should be evaluated.

1) Explicit Model Checking: Explicit model checking involves generating a complete state space graph, and traversing such graph representation exhaustively. The model checker tries to prove correctness by finding a counterexample. That is, it traverses the graph trying to find a state in which at least some of the required properties are not met. In terms of performance, a correct system represents the worst case, when the model checker needs to explore the complete graph to prove that the condition is satisfied. Unfortunately, as the system's model grows in complexity, the state space also grows exponentially, and this issue is commonly known as the state explosion problem. To solve this problem in executable time, many techniques have been developed. Partial order reduction, symmetry reduction and on-the-fly state graph generation are only some of the most common approaches [20].

2) Symbolic model checking: An alternative to explicit model checking is symbolic model checking. In symbolic model checking [21], instead of building a state graph, the transition system is represented as a set binary decision diagrams (BDDs), and then evaluated against the conditional properties. The advantage of symbolic model checking is that it only needs to represent the states and its relationships, the transitions, and therefore can have smaller time and space requirements than explicit model checking. However, studies show [23] that as complexity of the system being evaluated grows, the complexity of the BDDs also grow. Therefore, the state explosion problem is also present in this type of model checkers.

3) Bounded model checking: Another form of verification is bounded model checking. This technique was first introduced by Biere et al. in 1999 [12]. In bounded model checking, a system is modelled as a set of logic formulas that are then solved using a satisfiability modulo solver. This method makes the verification process simpler in terms of memory and computation complexity. However, it adds some overhead to the modelling phase and provides an incomplete proof of correctness (only to a user-defined bound).

C. Model checking tools

In this section we will explore some of the most popular model checking tools that have been used or could be potentially used for verification of memory consistency and cache coherence protocols. In our case studies section we explain how these tools have been used to verify cache coherence and consistency, and explain some of the issues that remain to be addressed.

1) SPIN: SPIN is an explicit model checking tool that has been widely used to verify concurrent systems [13]. In SPIN, a system is modelled using the Promela language (Process Meta Language). This is an imperative language with a c-like syntax, usually used to model distributed systems that abstract the details of implementation. Synchronization between processes is achieved by the use of shared variables or communication channels. Process execution is interleaved non-deterministically, so no assumptions can be made about timing properties. System properties are specified using linear temporal logic (LTL). SPIN has two modes, simulation and verification. The former one being used mainly as a design tool and for system refinement. The verification mode takes advantage of several optimization techniques used to reduce time and space requirements. The verification process executes a depth-first search trying to find a counter-example that would violate one of the properties specified in linear temporal logic.

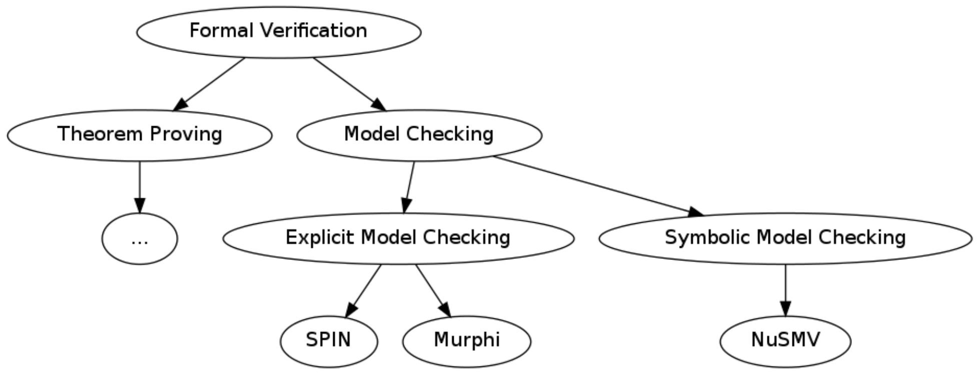

Fig. 6. Taxonomy of Verification Techniques and Tools

When a violation is found, SPIN outputs a trace file that can be used to reproduce the execution that lead to the error.

2) Murϕ: Murϕ is an explicit model checking tool originally developed at Stanford and supported by Intel [14] that works by enumerating all possible states and then doing a depth-first search to find possible traces that do not satisfy a set of boolean propositions. Modelling is done by enumerating a set of guard-action rules that are then executed iteratively by the model checker. Murϕ has been extended over the years, and a parallel version has been written that can take advantage of a distributed network to solve harder problems that could not be solved in a single machine. It has been used in industry to check a cache coherence protocol with approximately 30 billion states [15]. The Murϕ system provides semi-automatic symmetry reduction: if the user identifies symmetric structures by using special data types, the verifier automatically reduces the state space with respect to that symmetry [30].

3) NuSMV: NuSMV is a symbolic model checker developed by Carnegie Mellon University, the University of Genova, and the University of Trento [16]. NuSMV uses binary decision diagrams (BDD) and satisfiability (SAT) solvers to verify system properties. Version 2 of the system is modular and open source, and features an interaction shell, a graphical interface and extended model partitioning techniques. In the past, it has been used to verify cache coherence protocols and several errors in the design have been found [17] [18].

V. Verification of Cache Coherence Protocols and Memory Consistency

Early work in cache coherence protocols used simulation to test the behaviour of systems [10]. This methodology gained some popularity in the 1980s, and was still the main testing methodology applied by processor manufacturers in the mid-90s [26] [27]. However, many results have shown that this method was inconvenient and not sufficient to ensure systems correctness [28] [22]. Studies of formal verification applied to cache coherence protocols showed that it is a hard subject [5] [22]. A common approach [23] has been to model coherence protocols as finite state machines. The advantage of such approach is that existing model checking tools can be utilized, simplifying the verification task in comparison with other techniques such as theorem proving. Even though model checking has become a more popular technique for verification of coherence protocols, research community has taken different approaches. Explicit model checking has been used widely because it is simpler to understand than other methods of model checking that require higher level of abstraction. On the other hand, symbolic model checking has been gaining popularity as new tools have become available that showed better performance [16].

A. Modelling

In order to perform verification of cache coherence, we must first model the protocol being studied in a language that the verifier can understand and evaluate. In general, we want to be very precise to capture a model that truly reflects the actual design or implementation. In fact, as we will see later, a model that does not represent the exact protocol behaviour can lead to incorrect verification results. Accordingly, the specification must also be formalized using a precise language. On the other hand, it is important that our model abstracts from the actual system to be able to deal with the state explosion problem.

As we have previously stated, the main goal in the modelling stage is to develop an accurate representation that is as close as possible to implementation and at the same time its verification can be done by the model checker in computable time. Refinement is an interesting technique used to aid in the modelling process, in which a high level abstraction of the system is first constructed and verified. In the next step, more states and transitions are added to the system. Formal methods are used to prove that the new system is derived from the initial system and, therefore, it also satisfies the original set of requirements. The process continues iteratively, until a desired level of detail is obtained.

The relationship between the model and the design can be categorized according to the level of abstraction. Let us consider three levels [23]:

1) the behavioural level,
2) the behavioural level with aggregated properties, and
3) the implementation level.

The behavioural level captures the main functionality with high degree of abstraction. On the other end is the implementation level, which is equivalent to the actual code. Level 2 would be a middle ground between levels one and three. In general, we can only deal with the first or second level because of the state explosion problem. This is a limitation that current model checking techniques have, in order to keep verification in the computable order. Some studies [31], however, report that SMV has been used to verify cache coherence and, in combination with refinement techniques, the model verified has been extended down to Register Transfer Level (RTL) language successfully.

In particular for cache coherence protocols, we should take into consideration basic characteristics that are particular to these protocols. When we look at cache blocks, we should consider the following properties of state categories [19]:

- Validity
- Dirtiness
- Exclusivity
- Ownership

For example, let us examine the relationship between the properties of the states and the current state of a block. Recall that a block state can be either modified, shared or invalid. The modified state occurs when a valid block's state is exclusive to only one owner. If the block is dirty (data has changed), the block must be written back to memory before going into a shared state or being replaced at the local cache.

As we can see from the previous example, a verifiable model must accurately represent the system specifications and consider all possible behaviours. In explicit model checking, the later is achieved by reachability analysis, an exhaustive exploration of states. This leads to a well known issue in model checking, the state explosion problem.

The state explosion problem means that the complexity of the verification process will grow exponentially as the number of states in the system increase. This is due to the combinatorial nature of explicitly checking all possible states [5]. Furthermore, protocols that require modelling multiple system components also increase the state space significantly. Several techniques have been developed to deal with this issue. In the next section, we discuss different approaches to deal with the well-known state explosion problem.

B. Reducing the state explosion problem

Some early approaches limit the specification to perform a partial verification, or add abstraction layers to the model, which limits the number of states explored [5]. Despite the benefits of a smaller state space, these simplifications also reduce the effectiveness of the overall verification process.

More elaborated techniques exists, however, that preserve the semantics of the system. If we examine closer the characteristics of the system being modelled, we can take advantage of equivalence relations for state pruning. For instance, in the verification process of a snoopy protocol, the model of each of the cache controllers will add significant complexity to the overall system as number of caches increases. However, they exhibit similar behaviour that will result in unnecessary redundancy that can be removed.

Another approach for dealing with complex systems is to apply compositional verification. This method is based on splitting the system into various components that can be verified independently. After verification of each individual component, the components are composed back into a larger system [25].

State enumeration requires the model checker to traverse the graph searching for states that violate the requirements. In general, existing tools keep the system states in a stack. Some tools like SPIN and Murϕ optimize the search with on-the-fly stack traversal. A depth first expansion only keeps states on the current expansion path whose length is bounded by the depth of the state space graph [23].

An interesting approach to verification of larger systems has been paralleling model checking across multiple computing nodes. A successful implementation of this technique is PReach [32], a distributed version of the Murϕ model checker.

1) Symmetry reduction: Symmetry reduction is a technique to reduce redundant steps in the verification process. It is based on the observation that most complex systems show regularity. Therefore, states can be compressed by extracting common features. In the scope of cache coherence and memory consistency, this technique focuses on identifying equivalence relations in sequences of instructions that affect memory in a particular way.

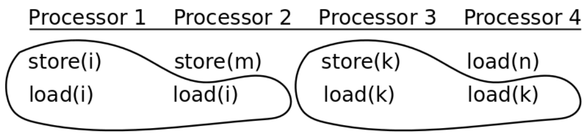

Fig. 7. Symmetry reduction

For example, Figure 7 shows an execution trace on a system with four processors in which an equivalence relation has been identified. The relationships are then grouped into classes. The verification process is simplified because only one element from each of the classes needs to be checked as the equivalence relation holds for all other elements of the class. Furthermore, from previous studies we know that given a protocol model with n processors, the maximum reduction is $n!$ [23].

2) Partial order reduction: Partial order reduction is a technique that exploits the redundancy of independent sequences of operations that lead to the same resulting state. Two operations are independent of each other if altering the order in which they are executed does not affect the the outcome of the sequence.

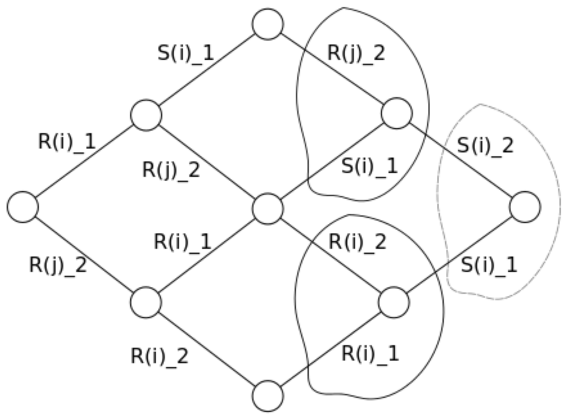

Fig. 8. Partial order reduction

For example, figure 8 shows the state graph for the previous program that we have examined that executed read and write operations on memory addresses i and j and ran on two processors. In our example, instruction $S(i)_1$ (a read of memory address i by processor 1) is independent of $S(i)_2$ (a read of memory address i by processor 2) because alternating the order of execution of these two instructions has the same effect on the system. This means the sequence of instructions $S(i)_2$ followed by $S(i)_1$ can be removed from the graph because the resulting state is the same that an execution of the sequence $S(i)_1$ followed by $S(i)_2$. Furthermore, the figure shows two other sequences of independent instructions that could be safely trimmed from the graph using the same process of partial order reduction just described.

3) State caching and on-the-fly stack search: Explicit model checkers use a depth-first search algorithm to exhaustively explore a transition system's graph looking a counter example that violates one of the specification properties. In complex systems, memory requirements can grow exponentially as the state space that needs to be explored increases. A method called on-the-fly stack search is used to guarantee that memory usage is limited during exploration of states. This optimization works because the search algorithm maintains a stack of states that have already been visited in the current path. The stack allows the algorithm to backtrack using information that is already known, and also to be able to provide an execution trace in case a counter-example is found.

C. Properties specification

There are two kinds of properties that can be specified for verification of cache coherence protocols. First, properties that prevent a system from going into an unwanted state. Those properties are called safety properties and the most common example is the mutual exclusion property, i.e. avoiding deadlock. The conflict would occur any time two caches try to obtain write access the same shared memory address. Only one cache must be able to write to main memory at any given time (write serialization). The other type of properties that must be verified are called liveness properties. Those ensure that eventually some expected behaviour will happen, or, in other words, that there is progress in the system, i.e. avoiding livelock and starvation [22]. When two caches try to obtain write access for a memory block, eventually each of them must get access and continue with execution.

Verification of memory consistency requires a slightly different consideration. To ensure consistency, we must evaluate the ordering of accesses to memory operations, both locally (at each cache) and globally (as a system). On a sequential consistency model, we know that the order in which processors execute is interleaved, but we cannot make assumptions on how the different processors will be interleaved. Therefore, we must treat the ordering choice non-deterministically. Moreover, we must verify that all read accesses return the latest value written to global memory. This requirement can be translated into a safety requirement that is an invariant throughout execution.

VI. Case Studies

In the past, different verification methods have been applied to cache coherence protocols 9. From theoretical proofs to explicit model checkers, there is no standard but a collection of diverse methods that have lead to different results. Theorem proving was the main verification method used in early research [44]. However, the development of new model checking tools made this technique more popular in the 1990s [40] [25] [23]. Furthermore, in the last 10 years systems grew larger and model checking techniques were found to be limited due to the state explosion problem. In most cases, to be able to verify such systems effectively, a combination of model checking techniques and inductive proofs were applied [48] [35].

A. Model checking the Alpha and Itanium processors memory consistency and coherence using Murphi

In 2002, Chatterhee [34] demonstrated how to model and verify weak consistency models found in two industrial processors (Alpha and Itanium). The author's focus was on finding an automated way to model the system for verification. In the study, a common convention [7] was followed to represent a memory consistency model as a finite state deterministic automata. The author defined every event in the system as a tuple (p, l, o, a, d), where the system was represented as follows:

p - the processor

l - a label for the instruction

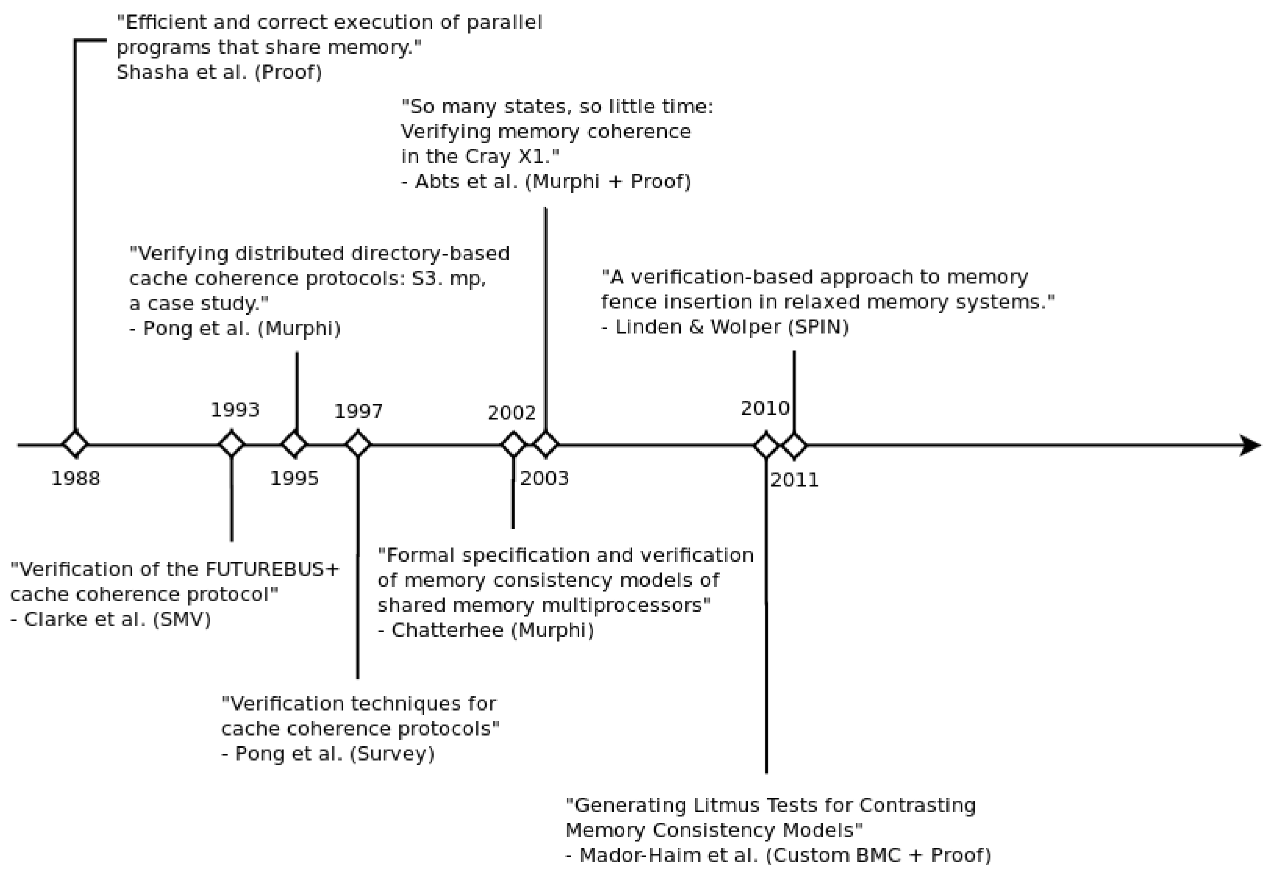
"Efficient and correct execution of parallel programs that share memory." Shasha et al. (Proof)
"So many states, so little time: Verifying memory coherence in the Cray X1." - Abts et al. (Murphi + Proof)
"Verifying distributed directory-based cache coherence protocols: S3. mp, a case study." - Pong et al. (Murphi)
"A verification-based approach to memory fence insertion in relaxed memory systems." - Linden & Wolper (SPIN)
1988
1993
1995
1997
2002
2003
2010
2011
"Verification of the FUTUREBUS+ cache coherence protocol" - Clarke et al. (SMV)
"Formal specification and verification of memory consistency models of shared memory multiprocessors" - Chatterhee (Murphi)
"Verification techniques for cache coherence protocols" - Pong et al. (Survey)
"Generating Litmus Tests for Contrasting Memory Consistency Models" - Mador-Haim et al. (Custom BMC + Proof)

Fig. 9. Timeline showing the most relevant published studies about verification of cache coherence and memory consistency

o - an event type
a - a memory address
d - the data

Next, a set of memory ordering rules that defined a sequential consistency model was described:

1) Read Value: The value of a load instruction $instr_{load}$ as acquired from the most recent store instruction $instr_{store}$ to the same address in the global memory order $<_m$. For example,
   a) $instr_{store} <_m instr_{load} \rightarrow a(instr_{load}) = a(instr_{store})$ and
   b) There is no store instruction $instr_i$ such that $a(instr_{load}) = a(instr_i)$ and $instr_{store} < instr_i < instr_{load}$
2) Per Process Order: The local order for each process $<_p$ was given by a subset of 5 rules. Let t_1 and t_2 be two instructions in the same process such that $t_1 <_p t_2$. Then:
   a) load $<_m$ load: $o(t_1) = load, o(t_2) = load$
   b) load $<_m$ store: $o(t_1) = load, o(t_2) = store$
   c) store $<_m$ load: $o(t_1) = store, o(t_2) = load$
   d) store $<_m$ store: $o(t_1) = store, o(t_2) = store$

The author developed a tool that took an assembly language program as input and generated all possible execution outcomes that would be allowed for a particular memory consistency model. This was achieved by translating the input program to promela language first, and then ran in SPIN to generate all possible execution outcomes.

To model coherence, a directory protocol was used. Because a split-transaction bus was used, coherence communication messages were not atomic. It was assumed that each block became owned right after an exclusive request was sent on the bus. Furthermore, a distinction was made between the address and data states of each cache's block. The address state was kept at the bus controller, while the data controller kept the state of the data, which may have lagged behind. The bus controller kept the states for each cache's block in one of: exclusive, shared, or invalid. To model the non-atomicity of the channel, each coherence action was divided into an acquired and released sub-states. Moreover, Scheurich optimizations were used. Scheurich optimizations allow processors to queue invalidations, send acknowledgements, and then perform the invalidations as long as the processor is kept in isolation from communication with the rest of the caches [34]. The code was implemented using Murϕ model checker and was included at the end of the research publication. To run the actual verification, the author used a parallel version of Murϕ model checker. Notably, it was reported that the process completed in between 54 and 240 minutes on 16 processors, visiting up to 250 million states.

B. Memory consistency relaxation in the x86 architecture

A more recent study on memory consistency was presented by Linden and Wolper [33]. In the study, the authors use SPIN model checker to verify a total store order (TSO) memory consistency model implemented in the x86 architecture. The x86-TSO extends the use of store buffers and allows synchronization and lock operations. The authors describe a process in which a sequential consistency program was converted into an equivalent relaxed consistency program.

The authors described the process as follows:

1) Initially, a program was modelled and checked against a sequential consistency specification.
2) Next, relaxed memory consistency properties were modelled according to the vendor's description of the behaviour of the system.
3) The program was then checked against a total store order consistency model.
4) To convert the program into a TSO-valid program, an algorithm was applied that inserted fence operations until correctness was obtained.
5) Finally, the resulting program was checked against the relaxed consistency properties showing that the program satisfied the memory consistency requirements.

Although several other studies have shown fence insertion algorithms in the past [36] [37], the particularity of this approach is that it can analyse cyclic programs. Furthermore, explicit state enumeration was used in contrast to other studies that used symbolic model checking techniques. Interestingly, the authors have shown that by using partial order reduction techniques, the complexity of verifying relaxed consistency can be similar to verification of sequential consistency, making explicit state verification an interesting alternative to symbolic model checking.

C. Verification of a cache coherence protocol using NuSMV

In a study published in 2008 [17], a MSI directory-based and a MESI snoopy bus-based coherence schemes were verified using NuSMV symbolic model checker. Four groups of properties were verified:

a) correct responses and correct transitions between states,
b) mutual exclusion: only one cache at a time can have a particular block in exclusive state,
c) staleness: responses should return up-to-date data, and
d) liveness: instructions should complete.

Several abstractions had to be made to keep the state space under reasonable runtime requirements; reads and writes were chosen non-deterministically; data operations was performed on a single variable (assuming this is principal source of conflict); a system with only three processors was modelled. Although this study reports successful verification of the coherence protocols, it is not clear if the abstractions applied to the model may diverge the verification results.

VII. Discussion and future work

Multiprocessor architectures became popular in the nineties and remain widely used in the present. In this context, cache coherence protocols and memory consistency models have been developed to deal with shared-memory issues. Studies have shown these models can be verified. In the last 15 years, the trend in microprocessor architectures has been to develop multi-core multiprocessor systems. As architects increase the number of cores in a system, cache coherence and memory consistency continue to be important problems. Some studies have tried to verify larger systems using a combination of model checking techniques and inductive proofs. However, the question remains if model checking can be used as a the only tool when systems scale the number of processors.

A recurring issue cited in model-checking experiments has been the state explosion problem [**?**] [23]. Numerous techniques have been developed to deal with this problem [33] [23]. As we have seen in our case studies section, these techniques have been applied to verify cache coherence and memory consistency successfully. In most cases, errors were found and designs were corrected accordingly [43] [25]. This confirms our idea that model checking is a useful verification method that can improve the quality of hardware design. Furthermore, the availability of tools make this technique easier to implement than more formal theorem proving techniques. Axiomatic verification techniques still require remarkable human insight and effort, which makes them less popular than model checking. It remains unknown if these methods could become more practical in the future. On the other hand, we recognize that there may be a need for development of alternatives to model checking because using only model checking tools some larger systems are still hard to compute in practical time. We recognize this issue is a disadvantage of model checking in comparison to other verification methods. New and improved tool that can deal with larger problems may become available, and model checking using SAT solvers may become more popular. We predict that the future of model checking is still open and will depend on future research and development.

VIII. Conclusion

This paper serves as a survey of verification of cache coherence and memory consistency models in multiprocessor systems with shared memory. First, we gained insight on why the coherency and consistency problems are critical in multiprocessor architectures.

After reviewing the most common coherence protocols and memory consistency models, we found that states of cache blocks in both directory and snoopy protocols can be represented as finite state machines and used as input for a model checker. We reviewed current developments in model checking tools, and standard techniques to reduce the state explosion problem.

We studied different approaches to verify memory consistency. From the case studies, we learned that several algorithms exist that can convert from sequential consistency to a more relaxed memory model. Due to the fact that humans think sequentially it is easier to write programs that follow a sequential order. For this reason, we believe that optimizing these algorithms (to only insert fence instructions when needed) and building some tool that implements them could be an interesting area for future development.

We reviewed current trends in memory organization and verification studies of coherence and consistency, and we realized that as new interconnect architectures are being designed and built, verification methods will have to be adjusted to deal with larger and more complex systems. We have not found studies that use just model checking for verifying such larger multi-core multiprocessor systems. Instead, a combination of model checking and induction proving seems to be a more appropriate approach to be able to generalize coherence and consistency in larger systems. This opens up the possibility of future research in combining model checking techniques with theorem proving verification.

Furthermore, by studying different verification experiments, we concluded that verifying systems is always beneficial and, in most cases, errors in protocol design are found and corrected. If that was not the case, verification is still useful as a refinement and debugging tool for protocol design.

Finally, as we were analysing how to model the different protocols, we identified that modelling is one of the most challenging parts of verification. Even if all the other parts of the process were correct, an error in the model could make the whole verification effort incorrect. For this reason, we believe that effective modelling and automation would be another important topic that could have significant impact in the application of verification to memory organization and other fields.

References

[1] L. Freed and S. Ishida, *History of Computers.* Ziff-Davis PUblishing Co., Hightstown, NJ, USA.1995.

[2] G. Moore, ”Cramming more components onto integrated circuits.” 1965.

[3] D. Patterson and J. Hennessy. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.1990.

[4] D. Sorin et al., *A Primer on Memory Consistency and Cache Coherence*, 3rd ed. Harlow, England: Addison-Wesley,2011.

[5] G.V. Bochmann and C.A. Sunshine ”Formal Methods in Communication Protocol Design”, *IEEE Transactions on Communications*, Vol. COM-28, No. 4, pp. 624-631.April 1980

[6] L. Lamport. ”How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”, *IEEE Transactions on Computers, C-28(9):69091,*Sept. 1979.

[7] K. Gharachorloo. ”Memory consistency models for shared-memory multiprocessors”, *Diss. Stanford University,*1995.

[8] Lenoski, Daniel, et al. ”The directory-based cache coherence protocol for the DASH multiprocessor”, Vol. 18. No. 3a. ACM, 1990.

[9] D. L. Weaver and T. Germond. *SPARC Architecture Manual (Version 9).* PTR Prentice Hall,1994.

[10] A. Agarwal et al., ”An evaluation of directory schemes for cache coherence”, *Computer Architecture. Conference Proceedings. 15th Annual International Symposium on , vol., no., pp.280,289,*30 May-2 Jun 1988

[11] Marty, M.R.; Hill, M.D., ”Coherence Ordering for Ring-based Chip Multiprocessors”, *Microarchitecture. MICRO-39. 39th Annual IEEE/ACM International Symposium on , vol., no., pp.309,320,*9-13 Dec. 2006

[12] A. Biere et al., ”Symbolic model checking without bdds”, *in Tools and Algorithms for Construction and Analysis of Systems, In TACAS99,*1999.

[13] S. Mador-Haim et al., ”The Model Checker SPIN”, *in Proceedings of the 22nd International Conference on Computer Aided Verification,*1997.

[14] D. Dill et al., ”Protocol Verification as a Hardware Design Aid”, *IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525.*1992.

[15] U. Stern and D. Dill, ”Parallelizing the Murphi Verifier”, *9th International Conference on Computer Aided Verification,*1997.

[16] A. Cimatti, et al. ”NuSMV 2: An OpenSource Tool for Symbolic Model Checking”, *in Proceeding of International Conference on Computer-Aided Verification (CAV 2002).* Copenhagen, Denmark, July 27-31, 2002

[17] K. Aisopos, et al. ”Extending Open Core Protocol to Support System-level Cache Coherence”, *in Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (CODES+ISSS ’08). ACM, New York, NY, USA, 167-172.*2008.

[18] Biere, Armin, et al. ”Bounded model checking”, *Advances in computers 58: 117-148.*2003.

[19] P. Sweazey and A. J. Smith. ”A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus”, *in Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 414423,*June 1986.

[20] S. Mador-Haim, et al., ”Verification techniques for cache coherence protocols”, *in Proceedings of the 22nd International Conference on Computer Aided Verification,*July, 2010.

[21] K.L. McMillan, ”Symbolic model checking”, *in Kluwer Academic Publ.,*1993.

[22] F. Pong, and M. Dubois. ”The verification of cache coherence protocols.” *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures.* ACM,1993.

[23] F. Pong, et al., ”Verification techniques for cache coherence protocols”, *ACM Computing Surveys (CSUR) 29.1: 82-126.*1997.

[24] D. J. Sorin, et al., ”Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol”, IEEE Transactions on Parallel and Distributed Systems, 13(6):556578,June 2002

[25] F. Pong, et al. ”Verifying distributed directory-based cache coherence protocols: S3. mp, a case study.” *EURO-PAR’95 Parallel Processing. Springer Berlin Heidelberg, 287-300.*1995.

[26] D. Lenosky, et al., ”The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor”, *Proc. of the 17th Intl Symposium on Computer Architecture, pp. 148-159.*June 1990

[27] M. Galles, and E. Williams. "Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor." *System Sciences. Proceedings of the Twenty-Seventh Hawaii International Conference on. Vol. 1. IEEE,*1994.

[28] K. L. McMillan, and J. Schwalbe. "Formal verification of the gigamax cache consistency protocol." *Proceedings of the International Symposium on Shared Memory Multiprocessing.*1992.

[29] A. J. Hu, et al. "Formal verification of the HAL S1 system cache coherence protocol." *Computer Design: VLSI in Computers and Processors, 1997. ICCD'97. Proceedings., 1997 IEEE International Conference on. IEEE,*1997.

[30] C. N. Ip and D. L. Dill. "Efficient verification of symmetric concurrent systems." *In International Conference on Computer Design, pages 230234. IEEE,*October 1993.

[31] Eiriksson, T. Asgeir "Integrating formal verification methods with a conventional project design flow." *Design Automation Conference Proceedings 1996, 33rd. IEEE,*1996.

[32] B. Bingham, et al. "Industrial strength distributed explicit state model checking." *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on. IEEE,*2010.

[33] A. Linden and P. Wolper. "A verification-based approach to memory fence insertion in relaxed memory systems." *Model Checking Software. Springer Berlin Heidelberg, 144-160.*2011.

[34] P. Chatterhee, "Formal specification and verification of memory consistency models of shared memory multiprocessors", Diss. The University of Utah, 2002.

[35] S. Mador-Haim, et al., "Generating Litmus Tests for Contrasting Memory Consistency Models", *in Proceedings of the 22nd International Conference on Computer Aided Verification,*July, 2010.

[36] M. Kuperstein, et al. *"Automatic inference of memory fences." Formal Methods in Computer-Aided Design (FMCAD), 2010. IEEE,*2010.

[37] T.Q. Huynh and A. Roychoudhury. *"A memory model sensitive checker for C sharp." FM 2006: Formal Methods. Springer Berlin Heidelberg, 476-491.*2006.

[38] E. A. Emerson, and E. M. Clarke. *"Using branching time temporal logic to synthesize synchronization skeletons." Science of Computer programming 2.3: 241-266.*1982

[39] A. Cimatti, et al. "NuSMV: a new symbolic model verifier", *Proceeding of International Conference on Computer-Aided Verification (CAV'99). In Lecture Notes in Computer Science, number 1633, pages 495-499, Trento, Italy,* July 1999.

[40] E. Clarke, et al. "Verification of the FUTUREBUS+ cache coherence protocol", *In 11th CHDL,*1993.

[41] K. L. McMillan and J. Schwalbe. "Formal verification of the encore gigamax cache consistency protocol", *In Proceedings of the International Symposium on Shared Memory Multiprocessors, pages 242-251. (sponsored by Information Processing Society, Tokyo, Japan),* 1991.

[42] M. Marty. *Cache coherence techniques for multicore processors. ProQuest,*2008.

[43] F. Pong and M. Dubois. *"Correctness of a directory-based cache coherence protocol: Early experience." Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on. IEEE,*1993.

[44] D. Shasha and M. Snir. *"Efficient and correct execution of parallel programs that share memory." ACM Transactions on Programming Languages and Systems (TOPLAS) 10.2: 282-312.*1988

[45] M. Zhang et al. *"Fractal coherence: Scalably verifiable cache coherence." Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on. IEEE,*2010.

[46] U. Wiener. *"Modeling and Analysis of a Cache Coherent Interconnect." Diss. Eindhoven University of Technology,*2012.

[47] F. Cao and Z. Liu. *"Snooping and ordering ring-an efficient cache coherence protocol for ring connected cmp." Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on. IEEE,*2009.

[48] D. Abts et al. *"So many states, so little time: Verifying memory coherence in the Cray X1." Parallel and Distributed Processing Symposium, 2003. Proceedings. International. IEEE,*2003.

[49] M. M. K. Martin et al. *"Why on-chip cache coherence is here to stay." Communications of the ACM 55.7: 78-89.*2012.