

MCjammer: An Adaptive Verification Tool for Multi-core and Multi-processor Designs

Ilya Wagner

Valeria Bertacco

Advanced Computer Architecture Lab
The University of Michigan – Ann Arbor, MI
{iwagner,valeria}@umich.edu

ABSTRACT

The challenge of verification of multi-core and multi-processor designs grows dramatically with each new generation of systems produced today. Validation of memory coherence and memory consistency of the entire system, which includes multiple levels of cache and complex protocols, remains a major fraction of this difficult task. Unfortunately, current tools are incapable of addressing these new challenges, leading to an unacceptably high risk that critical bugs could slip into designs, and make software behave unpredictably or cause wrong computation results.

In this work we present a scalable approach to the verification of memory coherence and consistency protocols in large multi-core and multi-processor systems. We accomplish this task with multiple cooperating agents, which feed the cores or processors with stimuli, attempting to both achieve their own verification goals and support other agents on their. The agents can dynamically change the stimuli that they generate based on coverage and pressure observed during the validation. Since each agent has a minimal knowledge of the entire system, their communication and decision process is greatly simplified. Moreover, since the agents' view of the system is independent of the number of nodes in it, our approach can be efficiently scaled to target large multi-processor systems. Our experimental results on two common memory coherence protocols demonstrate that this technique can reach 100% coverage of the individual agents' verification goals and of the system-level coherence protocol FSM much faster than a constrained-random approach.

1. INTRODUCTION

Multi-processor systems have been the foundation of high-performance computing for several decades. Supercomputers developed by companies such as Cray, IBM and SGI, featuring hundreds and thousands of processors all working in parallel, allowed several critical scientific problems to be solved in a timely manner and with great precision. Systems with fewer processors, on the other hand, were traditionally used by companies as databases and web servers. Recently, however, multi-processor and multi-core systems started to permeate the consumer market due to the inability of single-processor systems to support the programming trends of the market with just frequency scaling and microarchitectural improvements. Processors such as the IBM Cell [5], the Sun T1 (Niagara) [6] and the Intel 80-core 1.28 TFLOP processor [13] feature multiple cores which themselves are relatively simple, when compared to high-end processors that target single thread performance. Moreover, these parallel

systems are attractive due to lower power dissipation and higher reliability, in addition to their high performance on multi-threaded applications. Nonetheless, the complexity of these systems is increasing exponentially, with the number of cores or processors presenting a growing challenge to verification engineers.

Formal verification tools, such as SAT solvers and theorem provers, use mathematical reasoning to check if a design adheres to the specification. Unfortunately, often the capabilities of formal verification tools fall short of the complexity of today's high-end single processor systems, let alone multi-processors. Although these tools can prove fundamental properties in coherence protocols, such as absence of illegal transitions, they cannot handle the actual implementations of multi-processor systems and require much of the design to be abstracted away. On the other hand, constrained-random simulation-based approaches are quite scalable and can be used for verification at various levels of abstraction: high-level protocol model, low-level RTL, or even gate-level implementations. However, the shortcoming of these tools is their non-exhaustive nature: they can only guarantee the correct behavior of the execution scenarios that they investigate. Often the notion of coverage, or verification thoroughness, is used to assert the effectiveness of constrained-random methods and enable an engineer to design new tests to validate uncovered scenarios. Because of this, human involvement in simulation-based verification remains a major bottleneck of the validation process in today's industry, while insufficiently tested designs such as processor cache controllers are still being manufactured.

1.1 Contributions

In this work we present MCjammer - an adaptive verification tool for **Multi-Core** designs that uses closed-loop feedback to dynamically adjust its simulations to effectively test corner cases of design behavior. MCjammer is specifically designed for the verification of the memory subsystem, namely cache controllers, memory controllers and interconnect. MCjammer uses multiple cooperating agents that create and correlate test sequences trying to satisfy their coverage goals. Instead of fully understanding the system-level FSM of a coherence protocol, each agent uses a simplified view of the system to formulate its goals and produce sequences of memory accesses to achieve them. In addition, coverage and frequency of conflicting memory requests are analyzed dynamically by the agents, so that they can track progress on their goals, produce test sequences with large amount of "stress" on the system, and try to expose design errors. Finally, the data that agents supply to the design

under test is uniquely tagged and can be used to detect a variety of errors, including violations of memory coherence or even faults in the interconnect. Both simplicity of the system and data tagging enables us to easily scale and adapt MCjammer to large multi-processor designs. In addition it allows us to use MCjammer with a variety of coherence/consistency protocols and with different representations of designs, from high-level simulation of the protocol level FSM to RTL code implementing processors, memory and caches. Our experiments with two common cache coherence protocols demonstrate that MCjammer achieves better coverage with lower effort than an open-loop constrained-random simulation approach.

The rest of the paper is organized as follows: First we briefly overview the structure of multi-processor / multi-core systems and the challenges of their verification in Section 2, as well as related prior work in this field in Section 3. We then go over the structure of the MCjammer tool in Section 4 and explain in detail its feedback and consistency check mechanisms in Section 5. Section 6 presents the experimental results, and Section 7 concludes the paper.

2. BACKGROUND

In a shared-memory multi-core/multi-processor system several processors communicate via an interconnect structure (bus, network, *etc.*) to the main memory or with each other, as shown in Figure 1. Unfortunately, the latency of a memory access in such a system can be significantly higher than in a single-processor machine since memory is physically located much further away. A processor's request often must go through a network interface and make multiple hops to reach the memory controller and then return back with data. Therefore, caches, which reside at each core/processor and amortize the access time, become vital for performance. Unfortunately, this also complicates the interaction between processors since some of them might have more recent data in their caches than what main memory has.

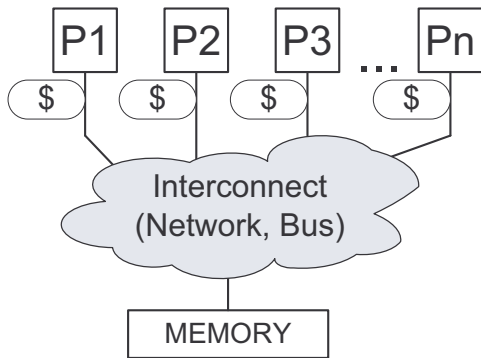


Figure 1: Structure of a multi-core/multi-processor system. Multiple cores/processors P_1 through P_n have separate caches, but communicate with each other and the shared main memory via interconnect.

To make sure that all processors have a coherent view of each memory location, and all data changes are propagated through the entire system with the best possible performance, a variety of *cache coherence* protocols are used. Figure 2 presents a model of *MESI* coherence protocol, where from the point of view of each cache controller a particular

memory location can be in one of the four states: ‘Modified’, ‘Exclusive’, ‘Shared’ or ‘Invalid’. ‘Invalid’ means that location is not present in the cache and to load it, the processor would have to send a request to the main memory. If the location is in ‘Modified’ at one of the caches, then the data was fetched by this processor from the main memory, and then modified by a store. However, the updated data is not visible to the rest of the system at this time. The processor can perform any operation on this data: loads, stores and evictions from the cache. All other processors at this point must have the same memory location marked as ‘Invalid’, so if they try to load the data, they must request it from the main memory, which in turn will retrieve it from the processor that modified it. If the memory location is marked ‘Shared’ it is guaranteed to be consistent with data in the main memory, and there may be other processors that have the same location in ‘Shared’ state as well. A processor can perform loads to a ‘Shared’ memory location or evict it without notifying other processors or the memory. ‘Exclusive’ state means that the processor is the one who owns the right to modify the location. Note that, if one processor has a location in either ‘Exclusive’ or ‘Modified’ state, others must have it in ‘Invalid’. A more thorough description of the MESI protocol, as well as other cache coherence protocols can be found in [3].

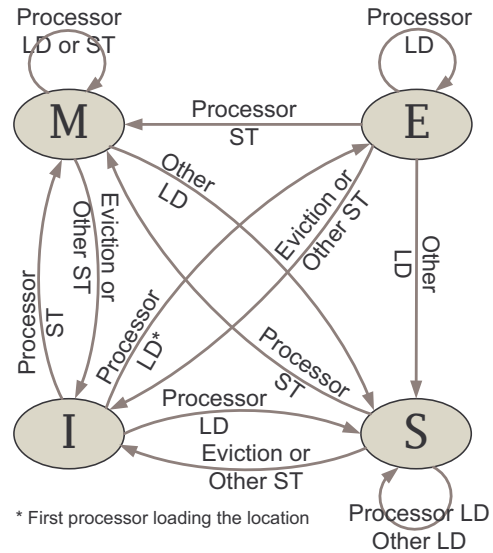


Figure 2: MESI cache coherence protocol. Each memory location can be in one of the following states in each cache controller: ‘Modified’, ‘Exclusive’, ‘Shared’ or ‘Invalid’. ‘I’ when the location is not available in the cache, ‘E’ when only the corresponding processor can modify the data, and ‘M’ after the value has been updated in the cache. The controller is in the state ‘S’ when the data loaded in the cache but cannot be modified.

Note that, the finite state machine of the protocol shown in Figure 2 only reflects the view of a single processor on the state of the memory location. Therefore, the full system protocol FSM for a single memory location is a *product* of n finite state machines (FSM), where n is the number of processor nodes in the system. An example of a product FSM for a MESI-based system with three nodes is shown in Figure 3. Thus, verification of the memory coherence in

a multi-core / multi-processor system includes verification of this system-level FSM. The main aspect to verify is the absence of invalid transitions and invalid states, for example, a state where several processors have the same memory location marked as ‘Modified’ in their caches.

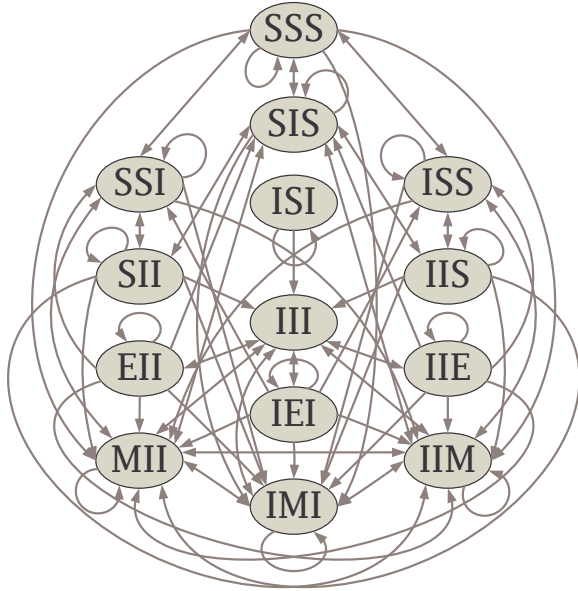


Figure 3: Full system FSM of three processor MESI-based system. Each processor follows the MESI protocol presented in Figure 2. A unique memory location in the system can be in one of the shown fourteen states.

Another crucial aspect of multi-core computing is memory consistency, which defines the order of memory accesses that are legal in a particular machine. The issue of consistency arises from the fact that scalable interconnects in multi-processors may re-order request messages, thus different processors may see the global sequence of loads and stores in different orders. For example, *Strict Consistency* demands that memory operations occur in the order they were issued. On the other hand, *Sequential Consistency* [7], requires that the same order is observed by all processors, although it may differ from the order in which accesses are issued. Other models, such as *Processor Consistency* and *Total Store Ordering*, pose even fewer restrictions on the order in which actions may occur. However, their implementations require special memory ordering commands so software developers can have precise, although expensive in term of performance, control over memory accesses. Memory consistency is typically hard to verify since implementable models tend to have a large range of possible behaviors that need to be tested. Failure to do this validation may result in unexpected software behavior.

3. PRIOR WORK

Verification of multi-processor systems was a strong focus of academic and industrial communities for a long time. Since the introduction of massively parallel supercomputers this effort mostly relied either on direct tests (programs) or constrained-random simulation-based techniques. Wood *et al.* [14] used random test generation to verify coherence of the cache controller in a shared-memory SPUR machine

designed and built at UC Berkeley. Recently Sorin *et al.* worked on several approaches to dynamically verify cache coherence [12] and sequential consistency [10] in multi-processors. Note that these dynamic verification techniques require additional hardware for on-the-fly error detection and correction. In contrast, our approach aims at verifying the multiprocessor at design time and preventing bugs from escaping into the final design.

There are also several formal verification tools available that specifically target multi-processor designs, including Mur φ [4] developed by Dill. Mur φ includes a special language for cache coherence protocol specification and a verification tool for protocol formal analysis. Our approach is different from this work in that we do not require a new language to specify the protocol and do not conduct verification via state enumeration. This makes our approach more scalable and applicable to simulation-based verification flows used heavily in today’s industry.

In addition, industry investigated several approaches to multi-processor verification, including work done at Cray Corp. [1] and IBM [11, 8]. Notably, the work of Malik *et al.* [8] used product state machine coverage for autonomous test generation, however, it is unclear how efficient this system would be in a large multi-core/multi-processor design. Another notable work from IBM includes the Genesys Pro test generator [2] that can be used to generate colliding memory accesses based on a set of templates given by the user. Unlike our solution, this tool requires significant user input for fine-tuning the simulation and is dependent on the configuration of the target system.

4. MCJAMMER TOOL

MCjammer is a simulation-based verification tool designed specifically to target multi-core/multi-processor systems with various interconnects and memory hierarchies. The tool considers the system at a very high level, which brings the advantage of being portable between multiple representations of the same system: abstract FSM, C simulator, protocol-based RTL or full RTL implementation. It is also easily tuned to work with different cache coherence protocols, memory hierarchies and memory consistency models. The tool employs multiple agents that generate concurrent and colliding memory access patterns, while at the same time helping each other to achieve postulated goals. This section gives an overview of the tool.

4.1 Overall Structure

MCjammer institutes a collection of cooperating adaptive agents attached to each of the processors or cores of the design (see Figure 4). An agent is responsible for identifying verification goals as well as actions taken by the individual processors during the validation process. The agents also check the coverage reports obtained after each *attempt* to reach the desired goals and adjust their actions so they *i.*) are more likely to see the desired goal and *ii.*) increase the pressure on the memory system to maximize the number of collisions and expose possible design errors. Both goals and the overall coverage are calculated in terms of transitions in a *dichotomic finite-state machine (DFSM)*, which depends on the coherence protocol of the system. A DFSM is a simplified view of the system, where an agent only distinguishes between its own actions and the actions of the “environment”, *i.e.* all other agents combined.

munication processes would not be scalable beyond just a few cores/processors. For example, the system FSM for a 16 processor MESI system would have 65568 states and an order of magnitude more transactions, making the collaboration between agents extremely complicated. The MCjammer for such system, on the other hand, would have partial overlapping DFSM with 8 nodes per processor, resulting in just 128 nodes in total for the 16 processor system. By dividing the problem into a smaller set of problems we can create a tool with manageable complexity.

4.3 Agents' Goals

Goals of individual agents in our framework are formulated as individual transitions in the DFSM that individual agents would want to verify in a particular *round*. For example, in Figure 4, agent 1 had chosen transition $IM \rightarrow SS$ as its goal. Since transitions in DFSM are labeled with actions that the agent and/or other agents need to perform for the transition to occur, generating actions to test a particular goal is straightforward. Note, however, that due to the many-to-many relation, we must add additional constraints to the coverage so the full system FSM is thoroughly verified. In our framework a transition in a DFSM is considered verified only if it has been observed several times. This increases the probability that all transitions in the protocol are verified.

Since each agent has a simplified view dictated by the DFSM, the algorithm governing its activity, shown in the pseudocode of Figure 6, is straightforward. The activity is divided into *rounds* during which individual agents formulate the goals using function *ChooseUnverifiedTransition()*. Each round consists of several *attempts* where agents decide if they will pursue their own goal, help another agent, or execute a random transaction (function *ChooseAction()*). Actions are then merged by function *MergeActions* into a program that is fed into a simulator. Results of the executions are then analyzed for presence of bugs (*CheckConsistency*) and coverage.

If, during a particular attempt, an agent chooses to work for itself, there is no guarantee that another agent will help it or that a transition of interest occurs. However, if an agent did not observe the desired transition, it would attempt to change the timing between its stimuli and request any of the helping agents to do the same. The process of delay reduction is based on the pressure metric discussed in Section 5.1. If the coverage report indicates that a transition of interest did occur a sufficient number of times during an attempt, an agent chooses another action based on the list of unreached goals and signals all agents who helped to do the same. There are several ways a *round* can terminate: when all goals are verified, or the specified number of *attempts* was reached but not all of the goal transitions are sufficiently verified.

4.4 Implementation Insights

In designing MCjammer, we decided to eliminate strict partnership, where agents deterministically choose partners to test various transitions. First of all, this simplifies the complexity of the agent algorithm. Secondly, we do not eliminate the possibility of executing the right sequence of tests to verify a transition, while at the same time creating various interaction scenarios between agents, stressing the consistency protocol.

```

for round = 1:N_ROUNDS
begin
  foreach Agent
    ChooseUnverifiedTransition(DFSM);

    for attempt = 1:N_ATTEMPTS
    begin
      foreach Agent
        ChooseAction( Goal, Coverage, Pressure );
      MergeActions;
      RunSimulator;
      CheckConsistency;
      Coverage, Pressure = AnalyzeResults;
      if ( AllGoalsVerified )
        break;
      end
    end
  end
end

```

Figure 6: Agent algorithm. In each *round* each agent formulates its goal to verify a particular transition in the DFSM. Then during multiple *attempts* each agent chooses either to pursue its own goal, help another agent or execute a random transition. After the simulation consistency is checked and coverage report and pressure metric are computed. Actions taken in the subsequent *attempts* by individual agents are based on coverage and pressure observed.

Also, it should be pointed out that our implementation of MCjammer is simulator-independent and is connected to a particular environment via the *MergeActions* function. Given actions associated with edges of the DFSM the function combines them and translates them into a format acceptable by the simulator. As we show in Section 6.2, in our experiments for this work we used the Wisconsin Multifacet [9] simulator with minor modification. One distinctive feature of the memory tester that is generated by this simulation environment is that it requires a single file with load and store instructions for multiple processors. Therefore, the translation method used in MCjammer had to combine actions of multiple agents into a single list of loads and stores. Alternatively, for simulation environments where separate program files need to be specified for distinct processor/cores, this method can be changed to produce these programs. This detachment of the verification intelligence (agents) from the actual simulator allows the tool to be highly portable not only between different simulators, but between various levels of simulation as well. For example, no changes, besides those to a *MergeActions* function are required to go from high-level C simulation to an RTL simulation of the system. Ultimately, our environment can be used even in gate-level simulations if verification of the implementation is required.

5. FEEDBACK AND CORRECTNESS

This section introduces the coverage model and coverage-directed feedback in MCjammer. We also discuss pressure, which is a metric of “stressfulness” of a simulation run. Finally, we show how MCjammer uses pressure as additional feedback parameter to increase the test quality.

5.1 Coverage and Pressure

Coverage is the measure of thoroughness of the verification process and assurance that all of the design behaviors are tested. Often coverage is only reported by verification tools to the engineer, who then designs the next test based on unverified regions of operation of the design. Unfortunately, this human intervention becomes a very high-latency and

high-cost part of the verification process. In MCjammer we chose to automate this process and close the loop with coverage and pressure feedback from simulation results to test sequence generator.

A natural coverage metric for a cache coherence and memory consistency protocol is the coverage of the states or transitions in the full system state machine of the protocol, since it is independent on its particular implementation. However, as was shown above, the number of states in this FSM grows enormously when number of processors increases. Therefore, using protocol state machine coverage for automated reasoning about the test thoroughness is a prohibitively complex task. In MCjammer, however, we chose to use coverage of DFSM transitions as the main metric for agents to reason about the effectiveness of the test and postulate new goals.

After each run, the agent identifies DFSM transitions were explored and records the information. Note that since the agent does not distinguish among the other agents, there exist a mapping from one DFSM transition to several transitions in the full system state machine. Therefore, a single traversal of an edge in DFSM is insufficient to guarantee that corresponding DFSM transitions were verified. So MCjammer agents are required to verify each DFSM transition multiple times before marking it as covered. In our tests we considered a transition to be covered when it was seen $2 * n$ times, where n is the number of processors in the system. Since agents do not have a global view of the entire system, it is possible that multiple transitions with the same pair of nodes occur and yet not all of the possible interactions between some pairs of nodes are tested. Nevertheless, we believe that threshold of $2 * n$ allows for substantial diversity in the stream of generated transactions while keeping the number of repeated transitions fairly low.

Note that since agents record all transitions observed, even a non-goal transition can become covered during the simulation. Once a transition is seen the set number of times by a particular agent, it is appropriately marked and could not be chosen as the agent's goal. The goals of the agents in MCjammer are adjusted dynamically with every coverage report obtained from each *attempt* of each *round*. This fine granularity of feedback allows us to efficiently direct tests towards unverified areas of design operation and requires no human effort or oversight.

In addition to coverage feedback, MCjammer uses a measure of pressure on the memory system to adjust the generated tests between consecutive *attempts*. In designing MCjammer we assumed that the most crucial for verification design functions involve "stressfull" operation of the system, when actions of different processors interfere with each other. An example of a collision in the MESI protocol shown in Figure 5 would be a situation where one processor tries to load a previously un-cached location from the memory and is in the process of going from 'Invalid' to 'Exclusive' state, and another processor tries to store to this location in the middle of this transition. Pressure in MCjammer is computed as a mean time between such colliding events at the caches and memory controller and is used to maximize the "stress" on the system proportionally. If the pressure is low function *ChooseAction()* (see Figure 6) will reduce the timing between actions done by the agents. Namely, the largest delay in the sequence of actions performed by an individual agent is reduced by a factor proportional to the pressure value. This increases the chance that a con-

flicting request arrives during an ongoing transaction and, therefore, increases collision possibility in future *attempts*. Moreover, since each agent changes its delays independently of the others this may cause a change in the order of issued actions to produce the transaction of interest. Higher pressure also causes a small reduction of some randomly selected delays between agents' actions in the attempt to change their order and investigate other execution scenarios. As our experiments demonstrate, pressure and coverage feedback help MCjammer to quickly create high-quality test sequences and achieve better coverage than an open-loop system.

5.2 Consistency Check

To check memory consistency and detect bugs in data or address manipulations we employ a data tagging technique. The data for each store in the system contains the unique ID of the agent issuing the store, the unique ID of the store operation at that processor, and a subset of the address bits of the store. Therefore, consistency and other invariants can be defined as a set of rules based on the store tags. Given the result of a load we can quickly identify which store is observed by this load operation, check the timing of the store and decide if a violation had occurred by consulting the consistency rules.

For example, in a system implementing Sequential Consistency all processors must see all store operations in the same order. By checking the tags of the data loaded by each processor, we can quickly establish if this rule was preserved during the simulation. Analogously, if Processor Consistency model is used in the system, these rules can be relaxed to allow writes from different processors be perceived in any order. However, writes from a single processor still must be seen in the same order by all of the other processors. Data tagging allows identifying if such order was preserved. Therefore, the consistency checker doesn't need a fully-specified memory reference model to establish that a violation has occurred, and only the axioms of load/store ordering are required. We believe this is a very powerful technique for multiprocessor validation, since most of the memory consistency models are defined in terms of such invariants. Although this framework does not directly allow to specify invariants such as absence of deadlock and races, such violations can also cause an inconsistency in tags retrieved by loads, and the system will alert the verification engineer. For example, if a memory access doesn't complete successfully until the end of the *attempt* due to a deadlock, MCjammer issues a "missing access" warning.

It is also important to note that the data tagging allows us to quickly identify problems with data and address transmission and other manipulations (storage, retrieval, buffering). If either data or address is corrupted while traversing the network or in caches/memories we can quickly detect this error by checking the tag. For example, if an address gets corrupted in a request message and a load returns data from a wrong memory location, the address bits encoded in the data may reveal the problem. Alternatively, the problem may appear as a consistency error when store ID or agent ID bits of the data get corrupted. In any case this will attract attention of the verification engineer who will investigate it more thoroughly to establish the cause of misbehavior.

Note that our system also keeps track of the state of each memory location, so coherence problems can also be recognized. For example, if a memory location is reported to

be in ‘Modified’ state in one cache and in ‘Shared’ state in another the checker will report the state as incoherent and alert the user.

6. EXPERIMENTAL RESULTS

In this section, we first introduce our experimental evaluation framework and the protocols and coverage metrics that we used to analyze the performance of our approach.

6.1 Experimental Framework

To analyze the performance of MCjammer we conducted several experiments on two multi-processor protocols using the Multifacet GEMS simulator [9]. In particular, we used the Ruby Simulator to model the interconnect, caches and memory and coherence controllers. We augmented the tester program included in Ruby to allow multiple nodes in the system to initiate overlapping memory operations. Memory operations such as loads and stores, as well as timing between their initiations, were supplied as a file to the Ruby tester, which also was modified to output additional information relevant to load and store data. Two protocols that we used in these experiments were MOSI, (*MOSI_SMP_Bcast_1level* in the Ruby model), and MESI, (*MESI_SMP_LogTM_directory*). Both of these benchmarks closely resemble real-life multi-processors and include complex FSMs with protocol and intermediate states, multiple queues for messages of different types (*i.e.* data and coherence messages), variable interconnect latencies, *etc.* The systems were configured to have four nodes, each containing only two banks of a fully-associative L1 cache. Consecutively, eviction from the cache was modeled as two back-to-back loads to specific memory locations.

MCjammer was implemented as a tester input file generator (written in C) and parser (written in Perl) that took the output of the Ruby tester, checked consistency axioms, and reported coverage and pressure back to the generator. Descriptions of DFSMs for both MESI and MOSI designs were created manually based on the protocol specifications. For performance evaluation during the simulation we monitored the average number of DFSM edges covered per node as well as state coverage of the full system FSM. In the experiments, the performance of MCjammer was compared to that of a constrained random memory access generator that did not use pressure feedback from the simulator and cooperating agents. The number of rounds in each of the simulation was increased from 2 to 128, while number of attempts was kept constant at 64. Therefore, for the random generator each round consisted of exactly 64 simulations with randomly generated patterns, while MCjammer, which featured early termination, could have fewer than 64 attempts in each round. The random generator also included DFSMs for each of the processor nodes identical to the ones in MCjammer to allow for DFSM coverage measurements.

6.2 Results

In the first experiment we used the number of DFSM transitions as coverage metric to gauge the performance of both MCjammer and a constrained-random generator (*RAND*). In Figures 7 and 8 we plot the average number of transitions covered by each of the agents in the four-core system against the number of rounds, *i.e.* the verification effort. As it is seen from the figures, MCjammer starts off with better coverage even for the smallest number of rounds and reaches full cov-

erage faster, unlike *RAND*, which never reaches 100% coverage. It is also important to notice that MCjammer usually requires fewer than 64 simulation runs (attempts) per round, while a round of *RAND* execution contained exactly 64 attempts. Therefore, the graph depicts the worst case performance for MCjammer, when none of the early termination techniques were used throughout the entire simulation.

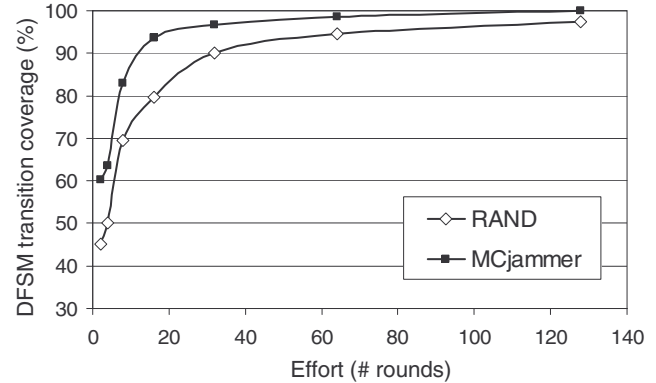


Figure 7: MOSI DFSM transition coverage. On average, *MCjammer* is able to cover more transitions in DFSMs at each node with of a four-node MOSI system with less effort than *RAND*.

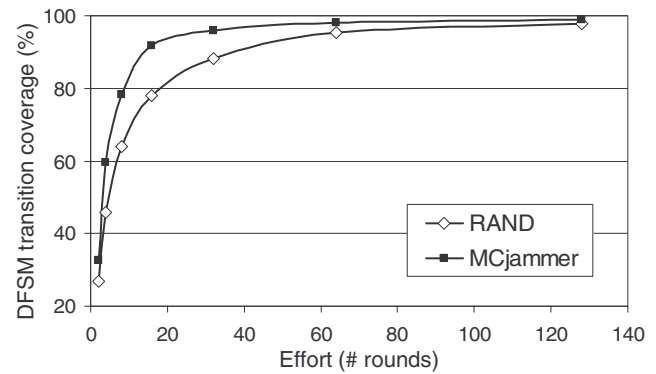


Figure 8: MESI DFSM transition coverage. On average, *MCjammer* is able to cover more transitions in DFSMs at each node with of a four-node MESI system with less effort than *RAND*.

In the second experiment we analyzed the state coverage of the full system FSM for a single memory location. Although DFSM coverage is an easy-to-measure metric, coverage of the system FSM is a more concrete measure of the thoroughness of the verification process. Note that MCjammer agents do not observe the full system FSM and, moreover, for a large multiprocessor the FSM would be impossible to represent explicitly. The purpose of this experiment is, therefore, to gauge the effectiveness of MCjammer with its distributed agents and DFSM coverage and compare our technique to constrained-random simulation.

The results of the experiments are shown in Figures 9 and 10, for MOSI and MESI designs, respectively. As you can see, the constrained-random simulator achieves significantly lower coverage of the entire FSM than MCjammer. We believe this is due to the lack of coordination between the memory accesses performed by the individual nodes in *RAND*, as well as low probability of generating colliding memory accesses. MCjammer, on the other hand, tries to

orchestrate accesses so not only DFSM coverage is improved, but various global states are explored. In addition, pressure feedback provides a useful metric to create multiple colliding memory access patterns.

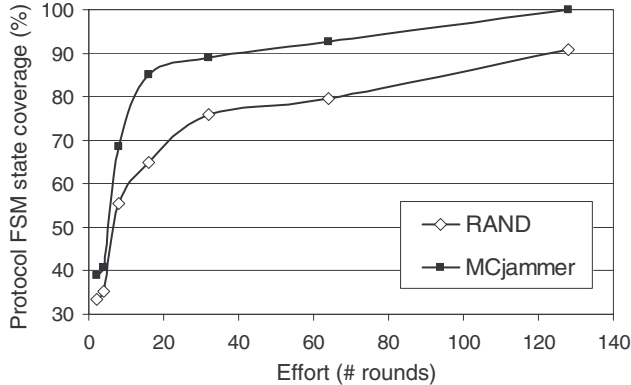


Figure 9: MOSI full system FSM state coverage. *MCjammer* is able to cover more states of the four-node MOSI protocol FSM with less effort than *RAND*.

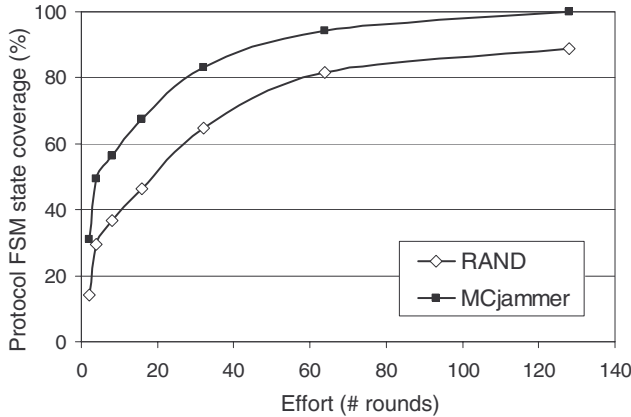


Figure 10: MESI full system FSM state coverage. *MCjammer* is able to cover more states of the four-node MOSI protocol FSM with less effort than *RAND*.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented *MCjammer*, a novel scalable tool designed specifically for verification of memory coherence and consistency in multi-core/multi-processor systems. *MCjammer* uses multiple adaptive agents that are attached to individual processors in the system and that work together to generate concurrent, and often conflicting, memory accesses. This coordination allows *MCjammer* to thoroughly cover the behavior of the design under test while also gradually increasing pressure on it to test “stressful” operation of the design. To set verification goals and measure coverage, each agent has a simplified view of the full system FSM of the coherence protocol that is independent of the number of processors/cores in the system. Additionally, *MCjammer* features unique data tagging that allows it to quickly verify if memory consistency rules were violated in the design or even detect errors in the interconnect. These features make *MCjammer* highly portable among designs with different protocols, number of nodes and cache hierarchies. Our experiments with two four-processor systems

indicate that *MCjammer* achieves 100% coverage with lower effort than an open-loop constrained random simulation.

In future work we plan to extend our tool to work with RTL simulations and systems with a large number of nodes, and more complex protocols and memory hierarchies. Additionally, we intend to establish a more detailed relation between the DFSM coverage and the actual protocol coverage metrics, including transition and path coverage.

8. REFERENCES

- [1] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: Verifying memory coherence in the cray x1. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 11.2, Washington, DC, USA, 2003. IEEE Computer Society.
- [2] A. Adir and G. Shurek. Generating concurrent test-programs with collisions for multi-processor verification. In *HLDVT '02: Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop (HLDVT'02)*, pages 77–82, Washington, DC, USA, 2002. IEEE Computer Society.
- [3] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, Aug. 1998.
- [4] 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*, pages 522–525, 1992.
- [5] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, pages 589–604, 2005.
- [6] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, pages 21 – 29, Mar. 2005.
- [7] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1997.
- [8] N. Malik, S. Roberts, A. Pita, and R. Dobson. Automaton: An autonomous coverage-based multiprocessor system verification environment. In *RSP '97: Proceedings of the 8th International Workshop on Rapid System Prototyping (RSP '97) Shortening the Path from Specification to Prototype*, pages 168–172, Washington, DC, USA, 1997. IEEE Computer Society.
- [9] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.
- [10] A. Meixner and D. J. Sorin. Dynamic verification of sequential consistency. *SIGARCH Comput. Archit. News*, 33(2):482–493, 2005.
- [11] A. Saha, N. Malik, B. O’Krafka, J. Lin, R. Raghavan, and U. Shamsi. A simulation-based approach to architectural verification of multiprocessor systems. In *Computers and Communications, 1995. Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on*, pages 34–37, Mar. 1995.
- [12] D. J. Sorin, M. D. Hill, and D. A. Wood. Dynamic verification of end-to-end multiprocessor invariants. *International Conference on Dependable Systems and Networks (DSN'03)*, pages 281 – 290, 2003.
- [13] S. Vangal, J. Howard, G. Ruhl, S. Digne, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28tflops network-on-chip in 65nm cmos. In *International Solid State Circuit Conference*, pages 5–7, 2007.
- [14] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Des. Test*, 7(4):13–25, 1990.