MCjammer: Adaptive Verification for Multi-core Designs

Ilya Wagner
iwagner@umich.edu

Valeria Bertacco
valeria@umich.edu

Advanced Computer Architecture Lab
University of Michigan, Ann Arbor, MI 48109

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 of such systems, which include multiple levels of cache and complex protocols, constitutes a major fraction of this task. Unfortunately, current tools are incapable of addressing these challenges, allowing bugs, which cause unpredictable software behavior and wrong computation results, to slip into hardware.

In this work we present a scalable approach to the verification of memory coherence protocols in large multi-core and multi-processor systems. We accomplish this task through a distributed network of cooperating agents, which feed the processors with stimuli, each agent attempting to accomplish its own verification goals and support other agents on theirs as well. The agents can dynamically change the stimuli based on coverage and pressure observed during simulation. 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 linear in the number of nodes in it, our approach can be efficiently scaled to target large multi-core systems. Experimental results on two common coherence protocols and a range of multi-core configurations demonstrate that our technique can reach high levels of coverage of the system-level protocol much faster than a constrained-random generator.

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, allowed critical scientific problems to be solved in a timely manner and with great precision. Recently, multi-processor and multi-core systems started to permeate the consumer market due to the inability of single-processor systems to support the performance trends with frequency scaling and microarchitectural improvements. Processors such as the IBM Cell [10], the Sun T1 (Niagara) [9] and the Intel 80-core Polaris processor [15] feature multiple cores of a relatively simple design; nonetheless, the complexity of these systems grows exponentially with the number of cores, presenting a growing challenge to verification engineers.

Formal verification tools, such as model checkers and theorem provers, use mathematical reasoning to check if a design adheres to the specification. Unfortunately, often the abilities 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 states, they operate on significantly abstracted designs and cannot handle actual implementations. On the other hand, constrained-random simulation-based approaches are quite scalable, but have a non-exhaustive nature: they can only guarantee the correct behavior of the scenarios that they investigate. Often the notion of coverage, or verification thoroughness, is used to evaluate the effectiveness of verification methods and to design new tests targeting uncovered scenarios. Because of the need of ad-hoc tests to boost validation coverage, human involvement remains a major bottleneck of the validation process in today's industry, while insufficiently tested designs are still being manufactured.

1.1 Contributions of this work

In this work we present MCjammer - an adaptive verification tool for **M**ulti-**C**ore designs that uses closed-loop feedback to dynamically adjust simulation to effectively test corner cases of design behavior. MCjammer is a novel scalable approach to stimulus generation and coverage analysis, which is specifically designed for the verification of the shared memory subsystem, namely cache controllers, memory controllers and interconnect. MCjammer relies upon multiple cooperating agents, each of them containing a simplified model of the system that is linear in the number of processors/cores in it. Agents create and correlate with each other sequences of memory accesses, attempting to maximize coverage of transitions in their respective simplified system models. The use of a distributed simplified model allows our approach to be more scalable, than techniques that are based on the full description of the system coherency protocol.

In addition, coverage and frequency of conflicting memory requests are analyzed 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, they allow us to use MCjammer with a variety of coherence/consistency protocols and with different design representations, including RTL code or high-level C simulators. Since individual agents in MCjammer do not require information about the entire system, we believe that this approach can be also extended to high-speed post-silicon validation.

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. Then, Section 3 overviews prior work in multi-core system validation. We then go over the structure of MCjammer in Section 4 and present in detail its feedback and error check mechanisms in Section 5. Section 6 presents the platform we used for MCjammer evaluation as well as results of our experiments, and Section 7 concludes the paper.

2. BACKGROUND

In a shared-memory multi-core or 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

978-3-9810801-3-1/DATE08 © 2008 EDAA

reach the memory controller and then return back with data. Therefore, caches, which reside within each core/processor and amortize the access time, become vital for performance. This also complicates the interaction between processors since some of them might have more recent data in their caches than what main memory has.

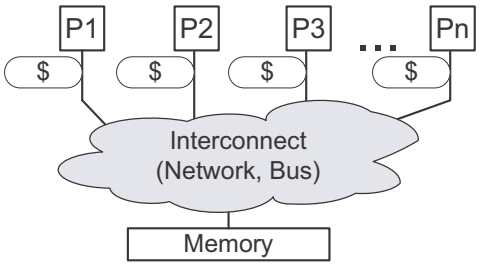

Figure 1: Structure of a multi-core/multi-processor system. Multiple cores/processors $P1$ through Pn have separate caches, but communicate with each other and the shared main memory via interconnect.

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 have been proposed. Figure 2 presents a model of the *MESI* invalidation coherence protocol (described in [4]), where (from the point of view of each cache controller) a particular memory location can be in one of four states: 'Modified', 'Exclusive', 'Shared' or 'Invalid'. For example, if processor P2 has a memory location 0x1000 in its cache in state 'S', the value in P2's cache is the same as in the main memory and the same memory location is potentially in the shared state in other processors' caches.

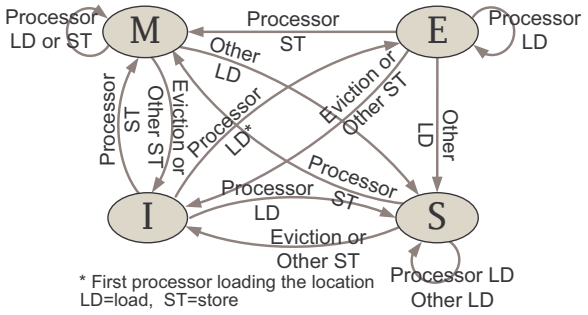

Figure 2: MESI cache coherence protocol. Each memory location can be in one of the following states in each cache controller: 'Modified', 'Exclusive', 'Shared' or 'Invalid'. 'I' when the location is not available in the cache, 'E' when only the corresponding processor can modify the data, and 'M' after the value has been updated in the local cache. The cache line is in state 'S' when the data is in the cache, but it may also be present in caches of other processors.

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 logic for the full system protocol 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. A product FSM for a MESI-based system with three nodes is shown in Figure 3. For example, a scenario where processors P1 and P2 have a particular memory location in state 'S', while P3 does not have it in the cache ('Invalid' state), corresponds to state 'SSI'.

Verification of the memory coherence in a multi-core/multi-processor system includes verification of this type of system-level FSMs, which encode all possible interactions of the nodes with respect to one memory location. The main aspects to verify in this case are absence of invalid transitions and invalid states (for example, states where several processors have the same memory location marked 'Modified' simultaneously).

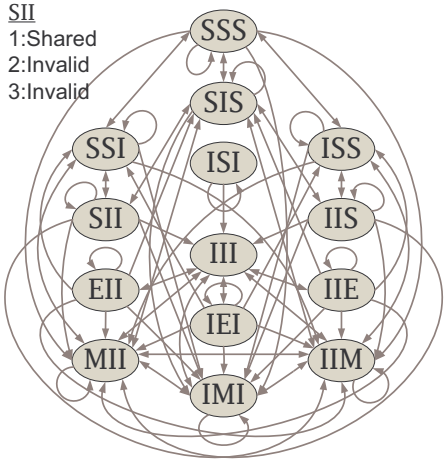

Figure 3: Finite state machine for the full system cache coherence protocol for a three processor MESI-based system. Each processor follows the MESI protocol (Figure 2).

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.* [16] used random test generation to verify cache coherence of the shared-memory SPUR machine. Compared to this type of testing, our approach offers the same scalability and improved coverage due to closed loop feedback and cooperations between stimulus generators for each of the nodes.

Formal techniques are also recognized as a valuable tool at early stages of multi-core/multi-processor design. For example, Murφ [5] was developed specifically for protocol specification and formal analysis. Chen *et al.* [3] use Murφ description of the machine to reason about system's coherence properties, however their approach still required human assistance for counter-example refinement. Murφ also was used in an industrial setting at IBM to verify ASCI Blue memory protocol [7]. However, due to the large size of the system, the authors still had to resort to several manual formal proofs. It is important to note that in this work the authors employed formal methods at the abstract protocol specification level, since the tools could not be applied to the logic-level implementation. Moreover, even with employment of this extensive formal verification flow, there were still escaped errors in components checked by Murφ. Similar experiences with the TLA+ language were reported by Joshi, *et al.* in [8].

Model checking approaches were also employed in several academic cache verification projects [14, 6]. The work by Pnueli, *et al.* [14] however, concentrates only on the formal description of the parameterized (in terms of number of nodes) protocol and may not be scalable to large systems due to usage of Binary Decision Diagrams (BDDs).

In [6] the authors conduct model checking on an abstract history graph, which reflects the state of a single processor and abstracted state of the rest of the system. An abstract

history graph is somewhat similar to the *Dichotomic Finite State Machine* (DFSM) used in this work, however, instead of formal reasoning, we use DFSMs to postulate coverage goals and record simulation coverage. Moreover, since we employ multiple agents, each with its own DFSM, our approach allows for better coverage of the protocol's state machine.

Several other notable approaches to multi-processor verification investigated in industry include the work done at Cray Corp. [1] and IBM [12]. In particular, the work of Malik, *et al.* [12] used product state machine coverage for autonomous test generation, however, it is unclear how efficient this system would be in a large multi-core design. Another 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 to fine-tune the simulation and is dependent on the target system configuration.

4. MCJAMMER TOOL

MCjammer is a novel simulation-based verification tool designed specifically to target multi-core and multi-processor systems. MCjammer requires only high-level knowledge of the coherence protocol, and it is easily portable to multiple representations of the same multi-core design: C model, protocol-based RTL or full RTL implementation. We also believe that the distributed nature of our approach can be adapted to high-speed post-silicon validation, allowing high-quality tests to be generated by software running on real cores or processors. Our tool employs multiple agents that generate concurrent and colliding memory access patterns, trying to expose incoherent cache states or errors in data or address manipulation. At the same time the agents support each other in achieving individual coverage goals, thus attempting to maximize coverage of the full protocol state machine. This section gives an overview of the tool.

4.1 Overall Structure

MCjammer instantiates a collection of cooperating adaptive agents, generating test sequences for each of the processors of the design (see Figure 4). To allow for scalability and efficiency of our approach, we designed each agent to have a novel simplified view of the system under verification, which we call *dichotomic finite-state machine (DFSM)*. DFSM allows an agent to distinguish only between its own actions and the actions of the "environment", *i.e.* all other agents. The DFSM in each agent is used for an internal representation of coverage. Agents keep track of DFSM transitions traversed in the past and use this information to direct testing towards unexplored scenarios of execution. At the beginning of each simulation run, each agent selects an insufficiently verified transaction in its DFSM, and generates load/store instructions to cover it. We also enabled collaboration between agents, allowing them to generate stimuli in a coordinated fashion to quickly achieve common coverage goals. At the end of the run, the agent checks the coverage report of the simulation and adjusts its actions so that *i*) it increases the likelihood of observing the desired transition in its DFSM, and *ii*) it increases the pressure on the memory system to maximize the number of collisions and expose possible design errors.

4.2 Dichotomic Finite-State Machine

As was pointed out in the introduction of this work, the number of processors and individual cores used in today's chips is increasing rapidly every new generation. Therefore, the number of possible states and state transitions in the global coherence protocol increases exponentially. As a result, tools that evaluate coverage on the global system level become unscalable. In designing MCjammer, we decided to divide the large problem of validating all possible transitions in the full system's coherence protocol (*full system FSM*) into a set of smaller problems, each with a simplified FSM. Instead of one agent formulating test sequences for the multi-processor system based on coverage or collision metrics, we chose to create a set of simpler agents cooperating with each other. However, for simplicity reasons, agents don't have an understanding of the full system FSM and instead use a *dichotomic finite-state machine (DFSM)* to represent their perspective of the protocol and coverage. States in a dichotomic finite-state machine are only comprised of the states for the local node and the state of the "environment", *i.e.* all the other nodes.

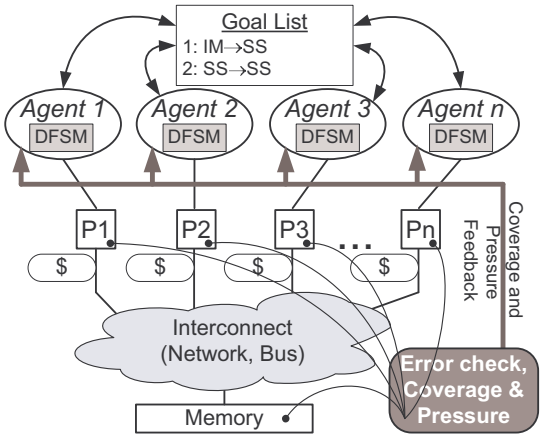

Figure 4: Structure of MCjammer. Each core in the system is assigned an agent. Agents formulate their goals in terms of transitions in the *dichotomic finite-state machine (DFSM)* of the memory coherence protocol specified by the user. During each run, agents choose if they want to attempt to achieve their own goals, provide support to another agent, or execute a random transition.

An example of a DFSM for MESI protocol is shown in Figure 5. A state in this figure represents the protocol for a single memory location at the agent's cache (first letter) and at some other agent's cache (second letter). For instance, in the state 'SI' the agent has the cache line marked as 'Shared', while some other agent has it as 'Invalid'. Transitions between states correspond to actions of the agent itself (subscript *s - self*) or other agents (subscript *o - other*) and include such actions as load (LD), store (ST) or cache eviction (E). A transition between states in DFSM represents a change in the state of the system, for example, transition from 'II' to 'IM' to 'SS' corresponds to a scenario where first some other node wrote to the location and then the agent itself loaded the location in the cache. Moreover, if in a four-node system a memory location transitioned from state 'SSSS' (all nodes have location as 'Shared') to 'SSSI', then in the first agent's DFSM both the edge from 'SS' to 'SI' and a loop from 'SS' to 'SS' are marked as visited. This is because in state 'SSSI' relative to the first agent there is another node that has the location as 'Shared' (either 2 or 3) and there is another node that has the location as 'Invalid' (node 4). Note that in a dual-processor/dual-core system, the DFSM corresponds exactly to the full system protocol, since there are only two nodes present. However, if the number of processors increases, the full system protocol FSM grows (recall Figure 3), while individual agents retain the same DFSM structure.

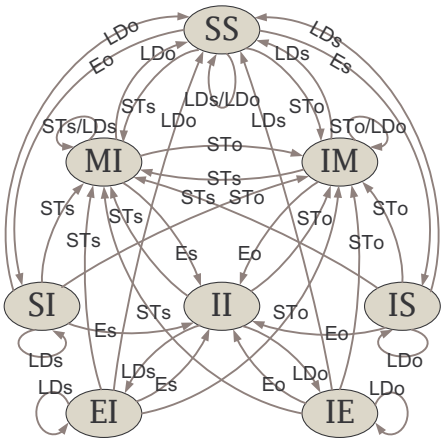

Figure 5: Dichotomic Finite State Machine for MESI protocol. DFSM represents a simplified view of the global states that each memory location in the system might have. An agent using this DFSM only distinguishes between actions of its own and actions of the "environment" or the rest of the agents. Transitions in the DFSM are labeled with the corresponding action that the agent (subscript s) or one of other agents(subscript o) must take. Actions include load (LD), store (ST), and eviction (E).

For more precise feedback and better cooperation between the agents, we augmented each DFSM with additional information: each edge in the Dichotomic Finite State Machine is associated with a *coverage vector* of n entries, where n is the number of cores in the system. An ith element of the vector is equal to the number of times a given edge was traversed by cooperation of the current agent and agent i. Therefore, we preserve pairwise edge coverage of the full system FSM. The coverage vector allows us to bias the probability distribution in the agent's algorithm to make the pairwise coverage of the DFSMs more even. Notice that although this additional information makes the size of the DFSM linear, in terms of the number of nodes in the system, it provides much more precise coverage information, while retaining the simple goal and action formulation algorithm of the DFSM.

The division of the complex protocol state machine into simple DFSMs allows us to retain the simplicity of individual agents and their interactions regardless of the number of agents in the system. On the other hand, if a single agent had precise knowledge of the entire system, *i.e.* the full system FSM, the complexity of its decision and communication processes would not be scalable beyond just a few cores. For example, a MESI system containing just four processors would have 211 edges. If each agent had a full view of the system, the collaboration between agents and the decision making algorithm would be extremely complicated. In the same case, MCjammer, on the other hand, has partial but overlapping DFSMs, each with only 37 edges, leading to a total of 148 edges. By dividing the problem into a set of smaller problems, we create a solution of manageable complexity, while, hopefully, maintaining high full system FSM coverage.

4.3 Agents' Goals

Each agent selects its verification goals by indicating which transition it would like to cover in the node's DFSM. For example, in Figure 4, agent 1 had chosen transition 'IM'→'SS' as its goal. Since transitions in the 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 covering all the transitions of all DFSMs is not equivalent to covering all the transitions in the full system FSM. To enhance the coverage in the full FSM (which we do not and cannot even represent), we force MCjammer to observe each transition within each DFSM several times, partially compensating the effect of DFSM's abstraction of the entire system.

Since each agent relies on a simplified view dictated by the DFSM, the algorithm governing its activity is also fairly straightforward. Prior to each simulation run, individual MCjammer agents formulate their goals by choosing insufficiently verified transitions in their DFSM's. Then all agents exchange their goals and make a probabilistic decision to either pursue their own goal, or generate a stream of instructions to allow another agent to achieve its goal, or execute a random stream of memory accesses. In our agent algorithm the probability of a random stream is constant, while probability of helping another agent is inversely proportional to the number of own goals that an agent had reached. In other words, agents that reach their own goals early are more likely to help other agents that have fewer goals covered. Actions of individual agents are then translated into streams of loads and stores timed in such a way that those of two or more collaborating agents have little overlap with other memory accesses. This is done by partitioning the set of agents into collaboration groups and allocating a time-frame for each group, so that interference between groups is reduced. After MCjammer sets up this list of timed loads and stores, the simulation starts with this distributed "program".

If, during a particular simulation run, an agent chooses to work on its own goal, there is no guarantee that another agent will help, or that a transition of interest will occur. However, if an agent does not observe the desired transition, it will attempt to change the timing between its load/store instruction and request any of the helping agents to do the same. The process of delay reduction is based on the pressure metric discussed below. If the coverage report indicates that a transition of interest occurred sufficiently often, the agent chooses another action based on the list of unreached goals and signals all helping agents that the goal has been accomplished. After a preset number of simulation runs, all agents are required to change their goals and repeat the procedure. In addition to coverage and pressure analysis, simulation results are analyzed for errors that may be detected with our unique data tagging technique discussed in Section 5.2.

In designing MCjammer, we decided to avoid strict partnership, where agents deterministically choose partners to test various transitions. This simplifies the complexity of the agent algorithm. and allows the possibility to generate unforseen interactions, which strengthen the verification of the coherence protocol. Moreover, the implementation of MCjammer is simulator-independent and can be easily ported between simulators of different level (architecture, RTL, and even gate-level). As we show in Section 6, in our experiments for this work we used the Wisconsin Multifacet [13] simulator with some minor modifications. Moreover, by partitioning and abstracting the full protocol's FSM into a collection of DFSMs, we make the agent's algorithm independent from any specific protocol, which enables MCjammer to be easily portable to other cache coherence protocols. In fact, the portability to other verification environments is executed by simply modifying the DFSM description, with no other modification. To test this aspect in our experimental setup, we seamlessly deployed MCjammer in a range of systems based on both MESI and MOSI protocols.

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 quality of generated tests.

5.1 Coverage and Pressure

Coverage is the measure of thoroughness of the verification process and full coverage is an assurance that all monitored design behaviors have been 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.

A natural coverage metric for a cache coherence is the coverage of the states and transitions in the full system state machine. However, as was shown in Section 2, the number of states in this FSM grows exponentially when the number of nodes increases. Therefore, using protocol state machine coverage to automatically evaluate the test thoroughness is a prohibitively complex task. To overcome this issue in MCjammer, agents evaluate coverage on their individual DFSMs. After a run, each agent identifies which DFSM transitions were explored and records the information. As we discussed above, a single traversal of an edge in the DFSM is generally insufficient to gain high coverage of all the corresponding full system. To boost the full system coverage, MCjammer agents are required to cover each DFSM transition several times before marking it as verified, so this transition cannot be chosen as the agent's goal again. The goals of the agents in MCjammer are adjusted dynamically with every coverage report obtained from each simulation run. This fine granularity of feedback allows us to efficiently direct tests towards insufficiently verified 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 simulation runs. In designing MCjammer we strove to create "stressful" activity on the system by having actions from different nodes interfere with each other. This allows MCjammer to exercise a rich set of unexpected interactions between the nodes for improved coverage. Moreover, empirical evidence suggests that complex bugs are often exposed by such situations. With reference to Figure 5, an example of such situation arises when one processor attempts to load a previously un-cached location from the memory and is in the process of going from 'Invalid' to 'Exclusive' state, meanwhile another processor tries to store to the same location at the same time. Pressure in MCjammer is computed as a mean time between colliding events at the caches and memory controller and it is used to maximize the "stress" on the system. If the pressure is low, the delay between actions initiated by the agents is reduced, hence increasing probability of collision in future simulation runs.As our experiments demonstrate, pressure and coverage feedback help MCjammer to quickly create high-quality tests and achieve thorough coverage of complex multi-processor protocols.

5.2 Error Check

To detect bugs in data or address manipulations and check for potential errors with memory consistency 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. Given the result of a load we can quickly identify which agent issued the last store that wrote to this location. For example, if the first agent is issuing a store with unique ID=2 to address $0x00AB3456$, the data written to this location will be $0x01023456$. The most significant byte in this case carries the ID of the agent, second byte - store ID and the last two bytes carry lower bits of the address. Therefore, if this location is accessed later and the two lower bytes do not have value of 0x3456, it will indicate a problem with data transmission/manipulation. Unfortunately, agents do not have access to the current *state* of the memory location in the cache, since this information is not architectural and not available to the processor core. To check for errors in cache coherence protocols, MCjammer's coverage analysis system tracks the state of each accessed memory location and reports invalid states, for example, a state where one agent sees the memory location transition to 'Modified' state, while another still observes it in 'Shared' state.

Data tagging can be beneficial for diagnosing memory consistency problems. Memory consistency, which defines the order of memory accesses that are legal in a particular machine, is also a crucial aspect of multi-core computing. 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, in a system implementing Sequential Consistency [11], 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 violated. Therefore, the error checker does not 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 powerful technique for multi-core validation, since most of the memory consistency models are defined in terms of such axioms.

6. EXPERIMENTAL RESULTS

To analyze the performance of MCjammer we conducted several experiments on two multi-processor protocols using the Multifacet GEMS architectural simulator [13]. 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. 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 systems were configured to include only two banks of fully-associative L1 caches. Descriptions of the DFSMs for both MESI and MOSI designs were derived from the protocol FSM specifications. For performance comparison we created a constrained-random (*ConstRand*) generator that did not feature collaborating agents and feedback, but, for fairness, produced the same type of accesses to the same memory locations with timing comparable to MCjammer.

In our experiments, we compared MCjammer with the constrained-random stimulus generator in terms of transition and state coverage of the full protocol FSM for a single memory location of a 16-core system. The results of the experiments are shown in Figures 6.a and 6.b , for MESI and MOSI protocol, respectively. The x-axis of the graph shows the number of instructions executed by MCjammer or *ConstRand*. The y-axis (log scale) evaluates the number of covered states or

Table 1: Bug coverage for MCjammer and *ConstRand*. MCjammer is capable of finding more bugs, and finding bugs in fewer instructions than a constrained-random simulator.

Bug name	MCjammer # instructions	*ConstRand* # instructions
dc_write	1248	6235
dc_two_writes_1	116	3272
dc_two_writes_2	116	240
dc_read_write	363	710
dc_write_read	3134	—
dc_two_reads	1215	10095
cc_data_write	116	180
cc_data_forward	4227	—

transitions of the full system protocol state machine. Note that in both experiments MCjammer achieves significantly higher coverage with lower effort than the random generator.

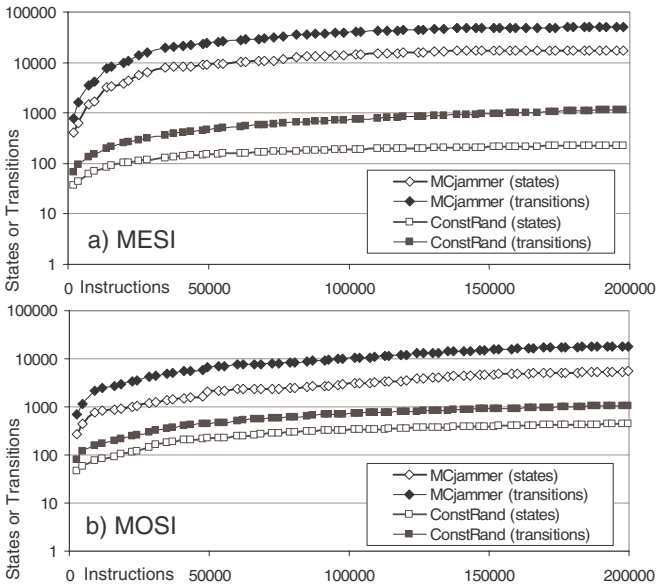

Figure 6: Comparison of state and transition coverage vs. effort for MCjammer and constrained-random simulation. MCjammer can achieve higher state and transition coverage on the 16-node full system state machine, with less effort.

In our final experiment, we inserted eight bugs of ranging complexity into a sixty-four-processor MOSI system and investigated how quickly MCjammer could find these bugs compared to *ConstRand*. The results of the study are presented in Table 1. The first six bugs were inserted into the logic of the directory controller (*dc*) of the system. The last two bugs, with prefix *cc*, were inserted into individual cache controllers. We ran both systems several times with a range of random seeds and measured the average number of instructions each system needed to execute to expose the bugs. For this experiment the *ConstRand* generator was configured to label the data similarly to MCjammer and used the same correctness checker to detect the bugs. As Table 1 shows, *ConstRand* was not able to find two of the bugs and required significantly more instructions than MCjammer to find the others. We believe this was due to *ConstRand*'s lack of coverage-based feedback that directs the test towards unexplored system behaviors, which is available to MCjammer.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented MCjammer, a novel scalable tool designed specifically for verification of memory coherence in multi-core/multi-processor systems. MCjammer uses multiple adaptive agents that are connected to individual processor nodes 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" operations of the design. To set verification goals and to evaluate coverage, each agent owns a simplified view of the full system coherence protocol, complexity of which is linear in the number of processors/cores in the system. MCjammer also features unique data tagging that allows it to quickly detect errors in the design and verify consistency rules. Our experiments on several MESI and MOSI systems of varying size indicate that MCjammer achieves higher coverage with lower effort than a constrained-random generator.

In future work we plan to extend our tool to work directly in RTL implementations. We also plan to work on systems with a large number of nodes and more complex protocols, memory hierarchies and interconnect structures.

8. REFERENCES

[1] D. Abts, S. Scott, and D. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *ISPDP*, 2003.

[2] A. Adir and G. Shurek. Generating concurrent test-programs with collisions for multi-processor verification. In *HLDVT*, pages 77–82, 2002.

[3] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *FMCAD*, 2006.

[4] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, Aug. 1998.

[5] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.

[6] E. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME*, 2003.

[7] S. German. Formal design of cache memory protocols in IBM. *Formal Methods in System Design*, 22(2):133–141, 2003.

[8] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with TLA+. *Formal Methods in System Design*, 22(2):125–131.

[9] K. Olukotun et al. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro*, pages 21 – 29, Mar. 2005.

[10] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Develment*, pages 589–604, 2005.

[11] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[12] N. Malik, S. Roberts, A. Pita, and R. Dobson. Automaton: An autonomous coverage-based multiprocessor system verification environment. In *Workshop on Rapid System Prototyping*, pages 168–172, 1997.

[13] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[14] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. *Lecture Notes in Computer Science*, 2001.

[15] S. Vangal, J. Howard, G. Ruhl, S. Dighe, 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 *Solid State Circuit Conference*, pages 5–7, 2007.

[16] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design and Test*, 7(4):13–25, 1990.