Automated Generation of Directed Tests for Transition Coverage in Cache Coherence Protocols

Xiaoke Qin and Prabhat Mishra
Computer and Information Science and Engineering, University of Florida, USA
{*xqin, prabhat*}@*cise.ufl.edu*

***Abstract*—Processors with multiple cores and complex cache coherence protocols are widely employed to improve the overall performance. It is a major challenge to verify the correctness of a cache coherence protocol since the number of reachable states grows exponentially with the number of cores. In this paper, we propose an efficient test generation technique, which can be used to achieve full state and transition coverage in simulation based verification for a wide variety of cache coherence protocols. Based on effective analysis of the state space structure, our method can generate more efficient test sequences (50% shorter) compared with tests generated by breadth first search. Moreover, our proposed approach can generate tests on-the-fly due to its space efficient design.**

I. Introduction

Caching has been the most effective approach to reduce the memory access time for several decades. When the same data is cached by different processors, cache coherence protocols are employed to guarantee that a read always returns most recently written data. Due to the power wall encountered by single core architectures, more and more cores are integrated into the same chip to boost the performance. As a result, the modern cache coherence protocols, like MOESI in AMD Opteron, are becoming quite complex. Unfortunately, since the reachable protocol state space grows exponentially with the number of processing units (cores) and states, the verification teams are facing significant challenges to achieve the required coverage within tight time-to-market window.

Since all possible behaviors of the cache blocks in a system with n cores can be defined by a global finite state machine (FSM), the entire state space is the product of n cache block level FSMs. Although the FSM of each cache controller is easy to understand, the structure of the product FSM for modern cache coherence protocols usually have quite obscure structures that are hard to analyze. Clearly, it is inefficient to use breadth first search (BFS) on this product FSM to achieve full state or transition coverage, because a large number of transitions may be unnecessarily repeated, if they are on the shortest path to many other states.

Simulation using random and constrained-random tests is widely used in industry because of its good scalability. However, the random nature of test sequences also introduces unacceptable time requirement to cover all possible state transitions in modern cache coherence protocols with many cores. Directed tests, on the other hand, are promising to achieve high coverage with a drastically small number of tests [1][2]. Therefore, they can be applied in addition to random tests to further improve the chances of capturing potential bugs. In case of complex protocols, it is also desirable to have an on-the-fly test generator with a space- and time-efficient test generation algorithm.

In this paper, we propose an on-the-fly test generation technique for cache coherence protocols by analyzing the state space structure of their corresponding global FSMs. Instead of using structure-independent BFS to obtain the directed tests, we show that the complex state space can be decomposed into several components with simple structures. Since the activation of states and transitions can be viewed as a path searching problem in the state space, these decomposed components with known structures can be exploited for efficient test generation. Our contributions in this paper are: i) we develop a graphical state space description of several commonly used cache coherence protocols, which can be viewed as a composition of simple structures; ii) we present an on-the-fly directed test generation algorithm based on Euler tour [12], which requires linear space with respect to the number of cores.

The rest of the paper is organized as follows. Section II introduces relevant existing research works. Section III provides related background information. Section IV describes our contributions in details. Experimental results are presented in Section V. Finally, Section VI concludes the paper.

II. Related Work

Existing protocol validation techniques can be broadly classified into two categories: formal verification and simulation based validation. Formal methods using model checking can prove mathematically whether the description of certain protocol violates the required property. For example, Murφ [3] was used to verify various cache coherence protocols based on explicit model checking. Counter-example guided refinement [4] is employed to verify complex protocols with multilevel caches. Symbolic model checking tools are also developed for coherence verification. For example, Emerson et al. [5] investigated the verification problem with parameterized cache coherence protocol using Binary Decision Diagrams (BDD). Fractal coherence [10] enables the scalable verification of a family of properly designed coherence protocols. Although formal methods can guarantee the correctness of a design, they usually require that the design should be described in certain input languages. As a result, it is usually difficult to apply model checking on implementations directly.

Simulation based approaches, on the other hand, are able to handle designs at different abstraction levels and therefore

978-3-9810801-8-6/DATE12/©2012 EDAA

widely used in practice. For example, Wood et al. [6] used random tests to verify the memory subsystem of SPUR machine. Genesys Pro test generator [7] from IBM extended this direction with complex and sophisticated test templates. To reduce the search space, Abts et al. [8] introduced space pruning technique during their verification of the Cray processor. Wagner et al. [9] designed the MCjammer tool which can get higher state coverage than normal constrained random tests. Since an uncovered transition can only be visited by taking a unique action at a particular state, it may not be feasible for a random test generator to eventually cover all possible states and transitions. To address this problem, some random testers are equipped with small amount of memory, so that the future search can be guided to the uncovered regions. Unfortunately, unless the memory is large enough to hold the entire state space, it is still hard to achieve full coverage by such guided random testing.

III. Background and Motivation

In modern computer systems, since the latency to transfer data from the main memory to processing units is much larger than the computation time, each processing unit usually maintains its local copy of the main memory, or cache for fast access. One major problem of caching is that when the same data, memory block, is cached in two or more different places, any modification should be propagated to all the cached copies. Cache coherence protocols are used to define the correct behavior of each cache controller.

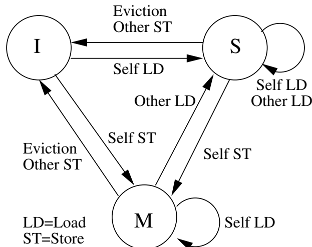

Fig. 1. State transitions for a cache block in MSI protocol.

One of the simplest cache coherence protocol is the MSI snoopy protocol [11]. The behavior of the cache controller in a processing unit is modeled as an FSM (Figure 1). The state of a cache block (line) can be either “Invalid”(I), “Modified”(M), or “Shared”(S). At the beginning, all cache blocks are in the invalid state. When a load request arrives from the core side (Self LD), the cache controller requests the data from the main memory and switch to shared state. When the core issues a store request (Self ST), the cache controller first broadcasts an invalidate request on the bus and then changes to modified state. Such an invalidate request will inform all other cache controllers that are in shared or modified states to change to invalid state. A cache block may also change to invalid state, when it is evicted by another cache block which is mapped to the same location in the cache, or when other cores issue store requests (Other ST).

Although MSI protocol can guarantee the coherence of the cache system, it causes some unnecessary delay and traffic on the communication channels. Many variants of the MSI protocols are invented to further improve its performance. For example, “Exclusive” (E) state is introduced in MESI protocol to avoid the traffic when a cache block is only used by one core. “Owned” (O) state is used in MOSI and MOESI protocol to reduce the delay when a modified block is loaded by other cores. As cache coherence protocols are becoming more and more complex, it is getting harder to verify their implementations. From the validation perspective, it is always desirable to activate all possible state transitions of the entire multicore cache system.

IV. Test generation for transition coverage

Our approach is motivated by Breadth First Search (BFS) in the state space of a global FSM. Given the FSM description of any cache coherence protocol, it is possible to compose a test suite which can activate all states and transitions using two steps: 1) for each state, we determine the instruction sequence to reach it by performing a BFS on the global FSM; 2) for each transition, we create the test by appending the required instructions after the instruction sequence to reach the initial state of this transition. However, such a naive approach has two problems. Transitions close to the initial state are visited many times. Thus, a large portion of the overall test time is wasted. Besides, since we have to remember all visited states in BFS, its runtime memory requirement also grows exponentially.

To address these challenges, our approach needs to satisfy two requirements: 1) we should reduce the number of transitions as much as possible without sacrificing the coverage goal; and 2) the space requirement for the test generation algorithm should be small. Fortunately, we can exploit the highly symmetric and regular structure of the state space and design a deterministic test generation algorithm, which can efficiently activate all states and transitions of popular cache coherence protocols. *The basic idea is to divide the complex state space into several components with regular structure.* Structures like hypercubes and cliques can be traversed by visiting each transitions exactly once.

This section is organized as follows. We first describe how to generate tests to activate all transitions of a simplified protocol: SI protocol. Next, we discuss our test generation techniques for a wide variety of popular protocols. In this work, we focus on the transition between two stable states. We assume that the transition between stable states to transient states are correct.

A. SI Protocol

SI protocol is a trimmed version of MSI protocol, in which we do not allow cores to issue store operation. For a system with n cores, a valid **global state** of the system allows the cache blocks in any m cores in I state and cache blocks in the other $n-m$ cores in S state. Thus, there are 2^n valid global states. Since any core in I (or S) state can be converted to S (or I) state within one transition, there are n outgoing and n incoming edges. It is easy to see that the entire state space

of SI protocol with n cores is a n dimensional hypercube[1]. Figure 2a shows such a state space with three cores. Figure 2b shows the representation of Figure 2a as a composition of three isomorphic trees (T_1, T_2, and T_3). *Since all edges are bidirectional for state transitions, we do not show transition directions explicitly.* For example, state III can be transformed to IIS when the first core loads the cache block. Similarly, state IIS can also be transformed to III, when the first core evicts this cache block.

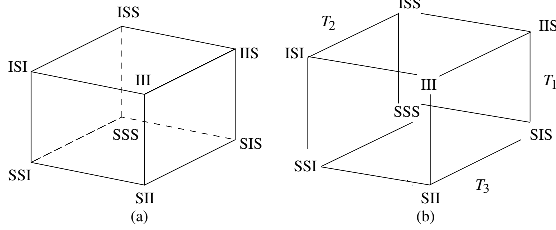

Fig. 2.(a) State space of SI protocol with 3 cores. Each global state is presented with 3 letters, e.g., IIS means core 2, core 1, and core 0 are in states I, I, and S, respectively. (b) Viewed as a composition of 3 isomorphic trees.

To achieve full state and transition coverage of the state space, we need to traverse each edge of the hypercube at least once in both directions. Since each global state has the same number of incoming and outgoing edges, it is possible to form an Euler tour [12] of the state space, which visits each edge exactly once in both directions.

Algorithm 1 Test generation for SI protocol with n cores

```
CreateTestsSI(n)
1: for i = 0 to n − 1 do
2:     Output "load(i)"
3:     VisitHypercube(1, n − 1, i)
4:     Output "evict(i)"
VisitHypercube(id, m, shift)
5: for i = 1 to m do
6:     newid = id + 2^i
7:     p = (i + shift) mod n
8:     Output "load(p)"
9:     if i > 1 then
10:        VisitHypercube(newid, i − 1, shift)
11:    Output "evict(p)"
12: return
```

Algorithm 1 outlines our test generation procedure for SI protocol, which performs an Euler tour on a n dimensional hypercube. Here, *load(p)/evict(p) means the pth core performs a load/evict operation in a particular cycle, while all other cores remain idle*. We use the state space in Figure 2 to show the execution of Algorithm 1. The algorithm starts by calling $CreateTestsSI(n)$. All cores are in I state at the beginning. In the first round of the *for* loop in line 2, the system performs transition III-IIS by executing load(0). During $VisitHypercube$, we visit transition IIS-ISS and ISS-IIS for $i = 1$ and IIS-SIS for $i = 2$. Since $i > 1$, we invoke *VisitHypercube* at line 10, which activates two transitions: SIS-SSS and SSS-SIS. Next, transition SIS-IIS is covered by executing evict(2) in line 11. Finally, the global state goes back to III via transition IIS-III after evict(2) in line 4. In the next two rounds of the *for* loop in *CreateTestsSI*, we essentially perform a "rotated" version of the previous traversal, which covers all transitions in paths III-ISI-SSI-ISI-ISS-SSS-ISS-ISI-III and III-SII-SIS-SII-SSI-SSS-SSI-SII-III (T_2 and T_3 in Figure 2b). Once the algorithm terminates, all transitions in the hypercube are covered by the generated test sequences.

Although the execution of Algorithm 1 seems to be complicated for larger n, the basic idea of this algorithm is quite easy: the hypercube is partitioned into n isomorphic trees with no overlapping edges. Once the hypercube is correctly partitioned, an Euler tour is performed on trees, because all edges are bidirectional. The space complexity of Algorithm 1 is linear with the number of cores n. The reason is that the function $VisitHypercube(id, m, shift)$ can be recursively called for at most $n-1$ times. The algorithm therefore requires a stack with at most $n-1$ levels. As a result, the space complexity is $O(n)$. The time complexity is linear to the number of transitions.

B. MSI Protocol

The difference between MSI protocol and SI protocol is that a cache block can be changed to the modified (M) state, when it receives a store request. For the ease of discussion, we define the following terms. A **global shared state** is a global state within which cores are in either shared or invalid states (e.g., IIS, ISI, ISS, SII, SIS, SSI, and SSS in Figure 3). A **global invalid state** is a global state within which all cores are in the invalid state (e.g., III in Figure 3). A **global modified state** is a global state within which exactly one core is in the modified state (e.g., IIM, IMI, and MII in Figure 3).

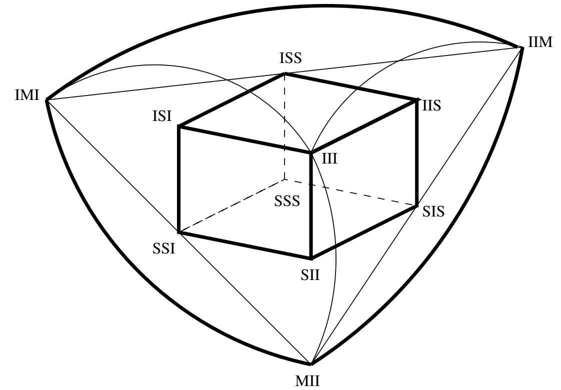

Fig. 3. State space of MSI protocol with 3 cores. For the clarity of presentation, the transitions to global modified states (IIM, IMI, MII) are omitted, if the transition in the opposite direction does not exist. The hypercube (at the center) and clique are highlighted.

Figure 3 shows the state space of MSI protocol with three cores. Since only one core can be in the modified state for MSI protocol, there are n global modified states in the state space of a system with n cores. Global modified states are reachable from any other global states by store requests from

[1]There are many transitions that start and end in the same state. For example, the global state will not change if a core in S state issues a load operation. These transitions are easier to cover, because they can be activated by appending one more operation at the end of existing tests, which are used to activate corresponding initial states. As a result, we omit them in the state space structure description in this section. However, all possible transitions are considered in our implementation to produce experimental results.

corresponding cores. Besides, a global modified state can also be converted to the global invalid state or global shared states. For example, global modified state IMI can be converted to global invalid state III by evict(1), or global shared states ISS and SSI by load(0) or load(2), respectively.

Clearly, all n global modified states form a clique, because there are two transitions (both directions) between each pair of global states. As a result, these transitions can be covered with an Euler tour. Unfortunately, *it is not possible to cover all transitions in the state space of MSI by a single Euler tour.* The reason is that for some global shared state like IIS, there are only outgoing transitions to global modified states, but no incoming transitions from them. Therefore, outgoing transitions are twice of incoming transitions. The similar scenario can also be observed for global modified states, which have more incoming transitions than outgoing transitions. To cover all transitions, some of them must be reused. In fact, the problem to minimize the number of reused transitions is similar to Chinese Postman Problem (CPP) [12], which can be solved by calculating the min-cost max-flow. Since we need to perform the test generation on-the-fly, finding the optimal solution by solving CPP is not an option, because the state space can be too large to fit into memory when there are many cores. Instead, we visit the uncovered transition to global modified state one by one and use the shortest path to link the end state of the previous transition and start state of the next transition.

Algorithm 2 Test generation for MSI protocol with n cores

$CreateTestsMSI(n)$
1: $CreateTestsSI(n)$ /* Invoke Algorithm 1 */
2: $VisitClique(0)$
3: **for** each global shared state s **do**
4:     **for** $i = 0$ to $n-1$ **do**
5:         Output "store(i)"
6:         Output the shortest path from current state to s

$VisitClique(p)$
7: Output "store(p)"
8: Output operations to visit all bidirectionally reachable global shared states
9: **for** $i = p+1$ to $n-1$ **do**
10:     Output "store(i)"
11:     **if** $i = p+1$ **then**
12:         $VisitClique(i)$
13:     Output "store(p)"
14: **return**

Algorithm 2 presents our test generation procedure for MSI protocol. We first invoke $CreateTestsSI(n)$ (Algorithm 1) to cover all transitions that also exist in SI protocol. Next, *VisitClique* will recursively perform an Euler tour in the clique of all global modified states. For example, when we execute *VisitClique* in the state space shown in Figure 3, we are first going to cover transition IIM-IMI. In the recursive call of *VisitClique* in line 12, transition IMI-MII and MII-IMI are visited. Next, transition IMI-IIM is covered by execution of line 13. In the next iteration, IIM-MII and MII-IIM are visited. To improve the efficiency, we also traverse all global shared states that are bidirectionally reachable from current global modified state. Finally, in line 3-6 we visit all uncovered transitions from global shared states to global modified states. Notice that we do not need to run Dijkstra's algorithm to find shortest path in line 6, because we must be in a global modified state after executing the store operation in line 5. The target global shared state can be reached by issuing load and evict requests based on the position of "S" in its state vector.

C. MESI Protocol

In MESI protocol, a cache block goes to exclusive (E) state when it is the first one, which loads a memory address. In a system with n cores, there are n global exclusive states[2]. Figure 4 shows the state space with three cores. Unlike global modified states, global exclusive states cannot be converted to each other directly. Therefore, the test generation algorithm *CreateTestsMSI* for MSI protocol needs to be modified to create tests for MESI protocol. We need to add n groups of operations to cover transitions from the global invalid state to global exclusive states as well as transitions from global exclusive states to global modified states. Notice that the *CreateTestsSI* routine, which is used to visit all transitions between global shared states, also needs to be modified slightly. The reason is that in MESI protocol, the global invalid state will be converted to global exclusive states after any load request (III goes to IIE instead of IIS when the first core issues a load request).

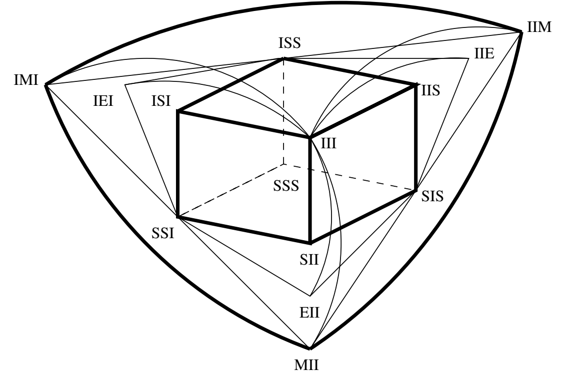

Fig. 4. State space of MESI protocol with 3 cores. The hypercube and clique are highlighted.

D. MOSI and MOESI Protocols

Algorithm 2 can be modifed to generate tests for MOSI and MOESI protocols. The details are omitted due to page-limit, and available in the technical report [14]. We have implemented all these algorithms and present the results in Table I.

E. Multi-level Cache Coherence Protocols

Due to the broadcast nature of snoopy protocol, modern processors with multiple cores usually employ multiple levels of cache to improve the utilization of the communication media. For example, in Intel Nehalem microarchitecture, while

[2] A **global exclusive state** is a global state with a cache block in exclusive state (e.g., IIE, IEI, and EII in Figure 4).

cores on the same die share a common on chip cache, the coherence among different processors are maintained using another level of cache coherence protocol. Figure 5 shows a simplified illustration of such a system with two cache levels. While the level 1 cache is private to each core, the level 2 cache is shared by two cores. The coherence should be maintained on both levels.

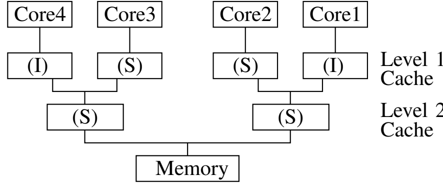

Fig. 5. System with two cache levels.

Clearly, our test generation approach can be applied directly, if each cache level is tested independently. However, this may not be adequate, because designers are more interested in the test cases, where the state transitions in different cache levels are involved. Therefore, we have to explore the state space of the global FSM, which captures the states of the same cache line in all levels of cache. We denote the global state of the system shown in Figure 5 as (IS)S(SI)S, which means the first level caches are in state I, S, S, and I, respectively, while the second level caches are both in shared state.

We start our discussion using the SI protocol, i.e., both levels are using SI protocol. *When the inclusive property is not enforced*, a memory block that is cached in upper level caches does not need to be cached in lower level. In this case, the state space of the global FSM is still structured. For the system in Figure 5, if the implementation does not enforce the inclusive property, the state space is a cross product of the state space of an SI protocol with four cores (represented by vertical transitions) and the state space for two cores (represented by horizontal transitions), as shown in Figure 6a.

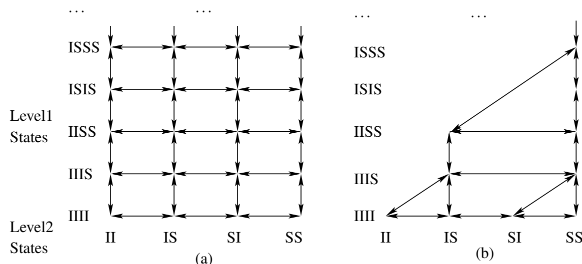

Fig. 6. (a) State space of a system with two cache levels. (b) State space when the inclusive property is enforced

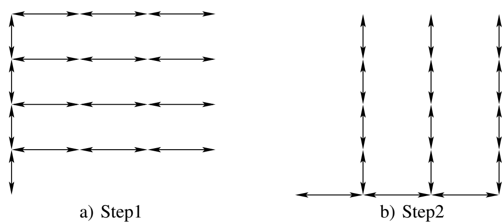

Fig. 7. Euler tour in Figure 6 by two steps.

As discussed in Section IV-A, all horizontal and vertical subspaces are hypercubes. Therefore, we can perform Euler tours on each of them. A global Euler tour or *CreateTestsSI* can be constructed by two steps as shown in Figure 7. In step 1, we start from initial state (II)I(II)I and perform an Euler tour within its corresponding vertical subspace, i.e., all vertical transitions in Figure 7a. We also visit the corresponding horizontal subspace, i.e., all horizontal transitions in Figure 7a, when we visit a state in the vertical subspace by a load operation. In step 2, we start from initial state and perform an Euler tour within its corresponding horizontal subspace, i.e., all horizontal transitions in Figure 7b, and visit the corresponding vertical subspace. Clearly, all vertical lines and horizontal transitions are covered exactly once. As a result, we obtain an Euler tour of the product space. When the inclusive property is enforced, the state space (Figure 6b) becomes smaller. The reason is that some states, like (II)I(IS)I, is not allowed. In this case, we can still start from the initial state (II)I(II)I and perform an Euler tour within its corresponding vertical subspace.

Our approach is also applicable to directory-based cache coherence protocols. Since the state of the directory node is uniquely determined by the states of all cache node, the structure of the global FSM remains the same as the snoopy protocol. Therefore, our test generation approach for snoopy protocols can also be used to reach transition coverage goals for the corresponding directory-based cache coherence protocols. When more complex protocols are used in different cache levels, we can extend the algorithms proposed in previous sections. Due to the limit of space, we discuss these modifications in the technical report [14].

V. Experiments

A. Experimental Setup

To analyze the effectiveness of our proposed test generation framework, we conducted a number of experiments using M5 simulator [13]. M5 is a full system simulator, which implements a MOESI cache coherence protocol. The load and store operations in the generated tests are translated into corresponding ALPHA instructions, while *evict* operation is achieved by loading a different memory address which is also mapped to the same location in the cache as the cache block under test. We use the *load-linked* and *store-conditional* instruction pairs to ensure the execution order of instructions in different cores. Since M5 only supports MOESI cache coherence protocol, we also developed a protocol simulator, which can be configured to simulate the state transition of a multicore system using MSI, MESI and MOSI protocols.

B. Experimental Results

In the first experiment, we compared the efficiency of our test generation method with the tests generated by performing breadth first search (BFS) directly on the global FSM for different cache coherence protocols with various number of cores. Since tests generated by BFS are the shortest tests to drive the system from the global invalid state to the required transition, we use additional operations to reset the global state after execution of each test. Table I shows the results. The second and third columns indicate the number of states and transitions in the respective protocol. Column "Total cost"

TABLE I
STATISTICS OF OUR TEST GENERATION ALGORITHM FOR DIFFERENT CACHE COHERENCE PROTOCOLS

	# States	# Transitions	BFS		Our approach			
			Total cost (transition)	Average cost per transition	Total cost (transition)	Average cost per transition	Improv. factor	Test generation time (sec)
MSI 8 cores	264	5256	36896	7.0	14664	2.8	60.3%	< 1
MESI 8 cores	272	5392	37712	7.0	15312	2.8	59.4%	< 1
MOSI 8 cores	1288	26248	196400	7.5	100807	3.8	48.7%	6.2
MOESI 8 cores	1296	26384	197216	7.5	101455	3.8	48.6%	6.2
MESI 16 cores	65568	2622496	29103264	11.1	11570464	4.4	60.2%	54.5
MOSI 16 cores	589840	23855632	275254368	11.5	131122063	5.5	52.4%	586

presents the total number of transitions traversed to activate all transitions. Column "Average cost per transition" provides the average number of transitions we need to traverse in order to activate an uncovered transition. It can be observed that the total size of the tests generated by our approach is 50%-60% smaller than the ones generated directly by BFS. This result can be explained by the fact that the Euler tour exploited in our algorithm typically covers load and evict transitions on global shared state. The store transitions on the other hand, are covered in a similar way as the BFS approach. Since the numbers of allowed load and evict transitions for any global state are equal, we can save almost around half of the tests by exploiting the space structure.

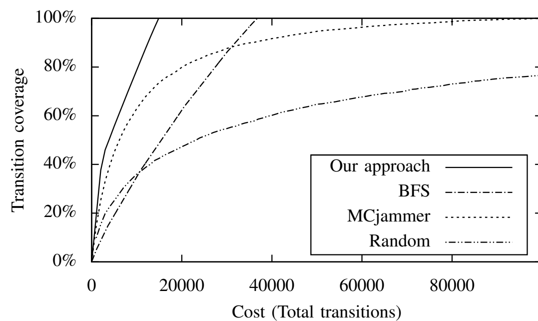

Fig. 8. Transition coverage vs. cost for MESI protocol with 8 cores.

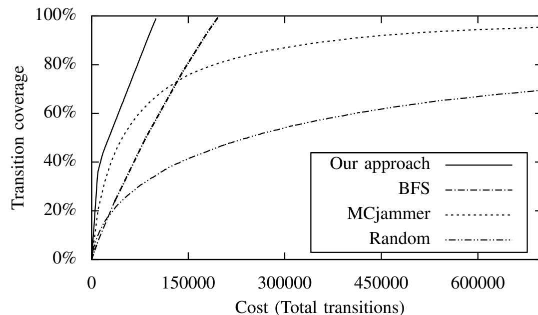

Fig. 9. Transition coverage vs. cost for MOSI protocol with 8 cores.

We also compared the state and transition coverage of our test generation approach with a directed random test generator, MCjammer [9]. Figure 8 and Figure 9 show the relation between transition coverage and testing cost on the same system. It can be seen that MCjammer is very efficient at the beginning. Actually, it is more efficient than BFS to achieve 70% coverage. However, it becomes much slower to cover all transitions. The reason is that it is very unlikely for the algorithm with randomness to cover remaining uncovered transitions among all allowed transitions. On the other hand, our proposed test generation approach can always achieve 100% state and transition coverage with stable higher coverage speed than the BFS based tests.

Although we described our algorithms in recursive forms to simplify the presentation, they can also be implemented as iterative routines. As discussed in Section IV-A, our algorithms have linear space complexity with the number of cores. Since our tests can be generated on-the-fly, its overall space requirement is very small. The test generation time in Table I suggests that the runtime of our algorithms is reasonable. For MOSI protocol with 23 million transitions, we can create all the tests within 10 minutes, which indicates that our algorithm is quite light-weight for entire simulation based validation phase.

VI. CONCLUSION

In this paper, we proposed an efficient test generation approach for a wide variety of cache coherence protocols. Based on detailed analysis of the space structure, our approach creates efficient test sequences for different parts of the global FSM state space to achieve 100% state and transition coverage for each cache coherence protocol. Compared with existing approaches based on constrained-random tests, our approach significantly improves the transition coverage with negligible memory requirement. Our experimental results demonstrated the effectiveness of our approach on systems with many cores and complex cache coherence protocols, making it suitable for future multicore architectures.

REFERENCES

[1] P. Mishra and N. Dutt, "Specification-driven Directed Test Generation for Validation of Pipelined Processors," *ACM TODAES*, 13(2), 2008.
[2] X. Qin and P. Mishra, "Directed Test Generation for Validation of Multicore Architectures," in *ACM TODAES*, 2012.
[3] D. Dill, A. Drexler, A. Hu, and C. Yang, "Protocol verification as a hardware design aid," in *Proc of ICCD*, 1992, pp. 522 –525.
[4] X. Chen et al. "Hierarchical cache coherence protocol verification one level at a time through assume guarantee," *HLVDT*, 2007, pp. 107 –114.
[5] E. Emerson and V. Kahlon, "Exact and efficient verification of parameterized cache coherence protocols," in *CHARME*, 2003, pp. 247–262.
[6] D. Wood et al. "Verifying a multiprocessor cache controller using random test generation," *IEEE Design&Test*, 7(9), pp. 13 –25, 1990.
[7] A. Adir et al. "Genesys-pro: innovations in test program generation for functional processor verification," *IEEE Design&Test* , 21(2), 2004.
[8] D. Abts, S. Scott, and D. Lilja, "So many states, so little time: verifying memory coherence in the Cray X1," in *Proc of ISPDP*, 2003.
[9] I. Wagner and V. Bertacco, "Mcjammer: adaptive verification for multi-core designs," in *Proc of DATE*, 2008, pp. 670–675.
[10] M. Zhange, A.Lebeck and D. Sorin, "Fractal Coherence: Scalably Verifiable Cache Coherence," in *Proc of MICRO*, 2010, pp. 471–482.
[11] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.
[12] J. Edmonds and E. L. Johnson, "Matching, euler tours, and the chinese postman," *Mathematical Programming*, vol. 5, pp. 88–124, 1973.
[13] N. Binkert et al. "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52 –60, 2006.
[14] X. Qin and P. Mishra, "Directed Test Generation for Cache Coherence Protocols," Technical Report, University of Florida, 2011.