Directed Test Generation for Validation of Cache Coherence Protocols

Yangdi Lyu, Xiaoke Qin, Mingsong Chen, *Member, IEEE* and Prabhat Mishra, *Senior Member, IEEE*

***Abstract*—Computing systems utilize multi-core processors with complex cache coherence protocols to meet the increasing need for performance and energy improvement. 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) on-the-fly compared with tests generated by breadth-first search. While our on-the-fly method can reduce the numbers of required tests by half, it can still be impractical to verify all possible transitions in the presence of large number of cores. We propose scalable on-the-fly test generation techniques using quotient state space. The proposed approach guarantees selection of important transitions by utilizing equivalence classes, and omits only similar transitions. Our experimental results demonstrate that our proposed approaches can efficiently trade-off between transition coverage and validation effort.**

***Index Terms*—Cache coherence, quotient space, test generation, verification.**

I. Introduction

SYSTEM designers incorporate multi-core processors to meet the increasing performance requirements. To address the memory bottleneck, 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 [1], 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.

This work was partially supported by the NSF grant CNS-1441667, Natural Science Foundation of China 61672230 and Cisco.

Y. Lyu and P. Mishra are with the Department of Computer and Information Science and Engineering at the University of Florida, Gainesville FL 32611-6120, USA.
E-mail: lvyangdi@ufl.edu
X. Qin is an architect with Nvidia.
E-mail: xqin@nvidia.com.
M. Chen is with the Shanghai Key Laboratory of Trustworthy Computing at East China Normal University, Shanghai 200062, China.
E-mail: mschen@sei.ecnu.edu.cn.

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 [2]. Therefore, they can be applied in addition to random tests to further improve the chances of capturing potential bugs. However, directed test generation is not practical in this case since the time and memory requirements can be prohibitive. Therefore, it is 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 directed tests, we show that 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:

1) We develop a graphical state space description of several commonly used cache coherence protocols, which can be viewed as a composition of simple structures [3], and present an on-the-fly directed test generation algorithm based on Euler tour [4].
2) We propose an efficient quotient space based test generation approach to address the scalability concerns in existing test generation techniques. The proposed approach utilizes the symmetric structure of protocol state space, which enables designers to cover important state transitions within limited verification budget.

The rest of the paper is organized as follows. Section II introduces related works. Section III provides background on cache coherence protocols. Section IV presents our on-the-fly test generation algorithms. Section V proposes scalable on-the-fly test generation using quotient space. Experimental results are presented in Section VI. Finally, Section VII concludes the paper.

II. Related Work

Researchers have designed many cache coherence protocols for different platforms and architectures, such as MSI, MESI, MOSI, MOESI, MESIF [5], MEUSI [6] and many other variations. As these protocols are becoming more and more complex, validating the correctness of these protocols also becomes challenging. Existing cache coherence 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 [7] was used to verify various cache coherence protocols based on explicit model checking. Symbolic model checking tools are also developed for coherence verification. For example, the verification problem with parameterized cache coherence protocol is investigated by Emerson et al. [8] and Li et al. [9]. Fractal coherence [10] [11] and PVCoherence [12] enable the scalable verification of a family of properly designed coherence protocols. Deadlock detection techniques [13] [14] are designed to automatically detect deadlock in cache 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. Moreover, manual translation (implementation to formal language) associated abstractions may introduce errors.

Simulation based approaches, on the other hand, are able to handle designs at different abstraction levels and therefore widely used in practice. For example, Wood et al. [15] used random tests to verify the memory subsystem of SPUR machine. Genesys Pro test generator [16] from IBM extended this direction with complex and sophisticated test templates. To reduce the search space, Abts et al. [17] introduced space pruning technique during their verification of the Cray processor. Wagner et al. [18] 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. Rather than generating test sequences to verify cache coherence, Cunha et al. [19] proposed a technique to detect violation by analyzing the traces of executing some benchmarks.

Quotient space is one of the symmetry reduction techniques. Through defining equivalence classes of states and restricting state space to representatives, verification techniques can be used to deal with large number of states. Clarke et al. [20] and Emerson [21] exploited symmetry reduction techniques in model checking. Kamkin [22] address state exploration problem by projecting state space to a number of subspace. However, to the best of our knowledge, quotient space was never utilized to improve test generation and validation of cache coherence protocols.

III. Background and Motivation

In modern computer systems, 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.

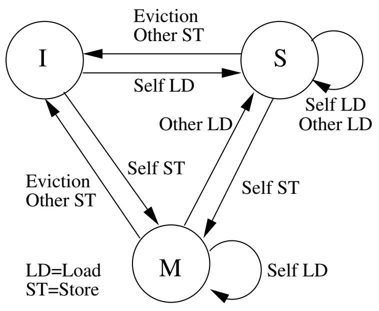


Fig. 1. State transitions for a cache block in MSI protocol.

One of the simplest cache coherence protocol is the MSI snoopy protocol [23]. 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 (Self LD) arrives, the cache controller requests the data from the main memory and switches 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 the 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. However, as outlined in the next section, traditional breadth-first search would lead to exponential memory and time requirements. Section IV describes our on-the-fly test generation algorithms to drastically reduce the test generation time and memory requirements. Section V introduces the concept of quotient space to develop scalable test generation techniques for complex cache coherence protocols.

IV. On-the-fly test generation

Our approach is motivated by breadth-first search 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. 1) Transitions close to the initial state are visited many times. Thus, a large portion of the overall test time is wasted. 2) 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 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 transition exactly once.

This section is organized as follows. For the ease of illustration, 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 including MSI, MESI, MOSI, and MOESI protocols. In this paper, we focus on the transitions between two stable states. We assume that the transitions between stable states and 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 for each global state. It is easy to see that the entire state space of SI protocol with n cores is an 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 core 0 loads the cache block. Similarly, state IIS can also be transformed to III, when core 0 evicts this cache block.

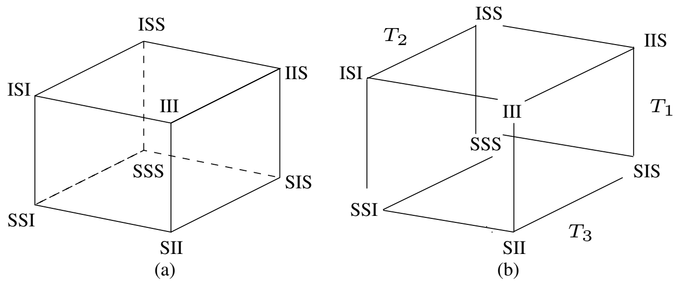


Fig. 2. (a) State space of SI protocol with 3 cores. Each global state is presented with 3 letters, e.g., IIS means core 2, core 1, and core 0 are in states I, I, and S, respectively. (b) Viewed as a composition of 3 isomorphic trees.

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 [4] of the state space, which visits each edge exactly once in both directions.

Algorithm 1 Test generation for SI protocol with n cores

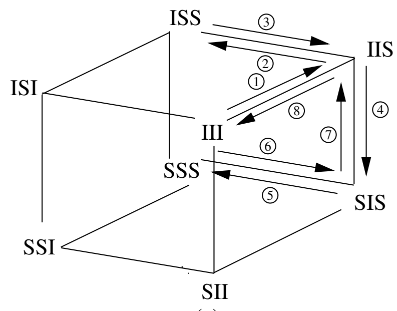
```
$CreateTestsSI(n)$
1: for $r = 0$ to $n-1$ do
2:     $VisitHypercube(n, r)$

$VisitHypercube(m, r)$
3: $p = (m + r) \bmod n$
4: Output "load(p)"
5: for $i = 1$ to $m-1$ do
6:     $VisitHypercube(i, r)$
7: Output "evict(p)"
8: return
```

Algorithm 1 outlines our test generation procedure for SI protocol, which performs an Euler tour on an n dimensional hypercube. Here, *load(p)/evict(p)* means that the p^{th} core performs a load/evict operation in a particular cycle, while all other cores remain idle.

Example 1: 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, $VisitHypercube(n, 0)$

[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.

is called. The system performs the transition III-IIS by executing $load(0)$ followed by two recursively call with $i = 1$ and $i = 2$. When $VisitHypercube(1, 0)$ is called, the transitions IIS-ISS and ISS-IIS are visited by executing $load(1)$ and $evict(1)$ without any further recursion. When $VisitHypercube(2, 0)$ is called, IIS-SIS is visited by $load(2)$, then $VisitHypercube(1, 0)$ is invoked to activate two transitions SIS-SSS and SSS-SIS, and at last SIS-IIS is covered by executing $evict(2)$. Finally, the global state goes back to III via the transition IIS-III after $evict(0)$ in line 7. The detailed steps and corresponding test sequence are shown in Figure 3, which forms an Euler tour on T_1 in Figure 2b. 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. ■



(a)

No.	Core	Instr.
1	0	load
2	1	load
3	1	evict
4	2	load
5	1	load
6	1	evict
7	2	evict
8	0	evict

(b)

Fig. 3. (a) $VisitHypercube(n, 0)$ (1st iteration of $CreateTestsSI(n)$, line 2 in Algorithm 1) visits the upper right part of the hypercube (T_1 in Figure 2b). The arrows are the steps to traverse the transitions with numbers to indicate the sequence. (b) The test sequence generated by the first iteration of $VisitHypercube(n, 0)$.

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. To show the correctness of our algorithm, we are going to prove 1) There are $n * 2^n$ transitions within the state space of SI protocol with n cores; 2) The transition sequence produced by Algorithm 1 has a length of $n * 2^n$; 3) No transition is visited more than once while we apply the transition sequence produced by Algorithm 1 on SI protocol. Clearly, Algorithm 1 does perform an Euler tour when these three statements hold. We prove these statements as follows.

Definition 1: A global state in the state space of SI protocol with n cores, is an n-dimensional vector $\boldsymbol{s} = [s_0, s_1, ..., s_{n-1}]$, where $s_k \in \{S, I\}$ indicates that the k^{th} core is in shared or invalid state.

Definition 2: An operation op is a function, which takes a global state in the protocol state space with n cores as input, and returns another global state in the same state space. We denote the application of an operation on state s, as $op \circ s$. Let $\boldsymbol{s_1} = load(p) \circ \boldsymbol{s}$. The p^{th} component of $\boldsymbol{s_1}$ must be S. Similarly, the p^{th} component of $evict(p) \circ \boldsymbol{s}$ must be I. When a sequence of operations $OP = op_1, op_2, ..., op_k$ are applied on a state, the resultant state is defined as $OP \circ \boldsymbol{s} = op_k \circ \cdots \circ op_2 \circ op_1 \circ \boldsymbol{s}$.

For the ease of presentation, we also denote $VH(m, r)$ as the operation sequence produced by invocation of $VisitHypercube(m, r)$.

Lemma 4.1: There are $n * 2^n$ transitions within the state space of SI protocol with n cores.

Proof: Notice that an n dimensional hypercube has $n * 2^{n-1}$ edges. Since each edge corresponds to two transitions, the total number of transitions becomes $n * 2^n$. ■

Lemma 4.2: The test sequence generated by Algorithm 1 contains $n * 2^n$ operations.

Proof: We show the number of load/evict operations performed by $CreateTestsSI(n)$ is $n * 2^n$. Since r does not affect the number of operations in each $VisitHypercube(m, r)$, we denote the length of $VH(m, r)$ as $l(m)$. We have

$$l(1) = 2, \quad l(m) = 2 + \sum_{i=1}^{m-1} l(i) \tag{1}$$

because the for loop in line 5 is repeated for $m - 1$ times. Two operations are performed in line 4 and line 7. It is trivial to verify that $l(m) = 2^m$.

We invoke $VisitHypercube(n, r)$ in $CreateTestsSI$ for n times. The total number of load/evict operations performed by $CreateTestsSI(n)$ is $n * 2^n$. ■

For the ease of illustration, we define a predicate $RI(\boldsymbol{s}, m, r)$ on a global state $\boldsymbol{s}$ which is true iff $\forall s_j$ is I where $j = (i + r) \mod n$, where $0 \le i \le m$.

Lemma 4.3: For any state $\boldsymbol{s}$, $RI(\boldsymbol{s}, m, r)$ implies $VH(m, r) \circ \boldsymbol{s} = \boldsymbol{s}$.

Proof: We prove this lemma by induction. For the base case $m = 1$, let $p = (1 + r) \mod n$. We have

$$VH(1, r) \circ \boldsymbol{s} = evict(p) \circ load(p) \circ \boldsymbol{s} = \boldsymbol{s}$$

because $RI(\boldsymbol{s}, 1, r)$ implies the p^{th} core in $\boldsymbol{s}$ is in I state.

Suppose $RI(\boldsymbol{s}, m, r) \implies VH(m, r) \circ \boldsymbol{s} = \boldsymbol{s}$ for all $m < m_0$. For $m = m_0$, the operations in $VH(m_0, r)$ are

load(p) $VH(1, r)$ $VH(2, r)$ ... $VH(m_0 - 1)$ evict(p)

where $p = (m_0 + r) \mod n$.

Let $\boldsymbol{s'} = load(p) \circ \boldsymbol{s}$. If $RI(\boldsymbol{s}, m_0, r)$ is true, we have $RI(\boldsymbol{s'}, m_0 - 1, r)$ is also true. Notice that $RI(\boldsymbol{s'}, m_0 - 1, r)$ implies $RI(\boldsymbol{s'}, m, r)$ for all $m \le m_0 - 1$. We have

$$VH(i, r) \circ \boldsymbol{s'} = \boldsymbol{s'}, \quad i \le m_0 - 1$$

Therefore, $VH(m_0, r) \circ \boldsymbol{s} = evict(p) \circ load(p) \circ \boldsymbol{s} = \boldsymbol{s}$. $RI(\boldsymbol{s}, m_0, r) \implies VH(m_0, r) \circ \boldsymbol{s} = \boldsymbol{s}$, which proves the lemma. ■

We define the **global invalid state** as a global state within which all cores are in the invalid state (e.g., the state III in Figure 2).

Lemma 4.4: If $\boldsymbol{s}$ is the global invalid state, every operation in $VH(n, r)$ covers an uncovered transition.

Proof: First, let's consider the call stack of $\mathit{VisitHypercube}$ immediately after we invoke line 6. Suppose the call stack is

$$\begin{array}{r} VH(n,r) \\ VH(k_1,r) \\ VH(k_2,r) \\ ... \\ VH(k_c,r) \end{array}$$

As i loops until $m-1$ in line 5, we have $n > k_1 > k_2 > ... > k_c$.

We prove the lemma by contradiction. Obviously, a load operation and an evict operation cannot cover the same transition. Assume that two load operations cover the same transition. They must be the results of line 4 during invocation of some VH, say $VH(k_a,r)$ and $VH(l_b,r)$. Since they are two different elements of $VH(n,r)$, the call stack of $VH(k_a,r)$ and $VH(l_b,r)$ must be different. Suppose the call stacks are

$$\begin{array}{rr} VH(n,r) & VH(n,r) \\ VH(k_1,r) & VH(l_1,r) \\ VH(k_2,r) & VH(l_2,r) \\ ... & ... \\ VH(k_a,r) & VH(l_b,r) \end{array}$$

Suppose their first difference is on the q^{th} location, i.e., $k_1 = l_1$, $k_2 = l_2$, ... , $k_{q-1} = l_{q-1}$ but $k_q < l_q$ (without loss of generality). Since both loads cover the same transition, the global state $\boldsymbol{s^a}$ and $\boldsymbol{s^b}$ must be the same after the load operation. However, the s^a_p is S, where $p = (k_q + r) \mod n$, while s^b_p is I. Therefore, we can conclude that two load operations will never cover the same transition.

A similar approach can be applied to evict operations by considering the state before performing the evict operation. It can be shown that two evict operations will never start from the same state. Thus, the lemma is correct. ■

Lemma 4.5: If $\boldsymbol{s}$ is the global invalid state, operations from $VH(n,r_1)$ and $VH(n,r_2)$ cannot cover the same transition.

Proof: It is enough to show that operations from $VH(n,0)$ and $VH(n,r)$ cannot cover the same transition, because if two operations from $VH(n,r_1)$ and $VH(n,r_2)$ cover the same transition, there must be corresponding operations from $VH(n,0)$ and $VH(n,r_2-r_1)$ covering the same transition.

We prove the lemma by contradiction. Obviously, a load operation and an evict operation cannot cover the same transition. Assume that two load operations from $VH(n,0)$ and $VH(n,r)$ cover the same transition. The global state before and after we apply them must be same, say $\boldsymbol{s} \rightarrow \boldsymbol{s'}$.

Since this is a load operation, the only difference between $\boldsymbol{s}$ and $\boldsymbol{s'}$ must be on a single core, i.e., s_p is I but s'_p is S. In other words, they must have call stacks like

$$\begin{array}{rr} VH(n,0) & VH(n,r) \\ VH(i_1,0) & VH(j_1,r) \\ VH(i_2,0) & VH(j_2,r) \\ ... & ... \\ VH(i_c,0) & VH(j_c,r) \end{array}$$

and $i_c = j_c + r \mod n$. Now let's consider the cores in S (shared) state within $\boldsymbol{s}$. We claim that $s_0 = s_r = S$, because $load(0)$ and $load(r)$ are the first elements in $VH(n,0)$ and $VH(n,r)$, respectively.

Since $\boldsymbol{s}$ is reached by $VH(n,0)$ from the global invalidate state, there must exist an i_k such that $i_k = r$. Therefore, $i_c < i_k = r$, since the first argument in the call stack of VH is strictly decreasing.

On the other hand, since $\boldsymbol{s}$ is also reached by $VH(n,r)$ from the global invalidate state, there must exist a j_k such that $j_k + r \mod n = 0$ or $j_k = n - r$. Since the first argument in the call stack of VH is strictly decreasing, $j_c < i_k = n - r$. Therefore, $j_c + r \mod n \geq r$. However, this is impossible, because $i_c = j_c + r \mod n$, and $i_c < r$. In other words, two load operations from $VH(n,0)$ and $VH(n,r)$ will never cover the same transition.

A similar argument can be applied to evict operations by considering the states before and after performing the evict operation. Thus, we conclude the proof. ■

Theorem 4.1: The test sequence constructed by Algorithm 1 does perform an Euler tour of the entire state space.

Proof: From all previous lemmas, we know that the number of operations generated by Algorithm 1 is same as the number of transitions within the state space of SI protocol. We also know that every operation performed by $VH(n,r)$ always covers a different transition. So the test sequence constructed by Algorithm 1 must perform an Euler tour. ■

The space complexity of Algorithm 1 is linear with the number of cores n. The reason is that the function $\mathit{VisitHypercube}(m,r)$ 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, the global invalid state excluded (e.g., IIS, ISI, ISS, SII, SIS, SSI, and SSS in Figure 4). 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 4).

Figure 4 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 state by store requests from 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 modified states. As a result, these transitions can be covered with an Euler tour. Unfortunately, for some global shared state like IIS, there are only outgoing transitions to global modified states, but no incoming transitions from them. A 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.

Observation 1: It is impossible to cover all transitions in the state space of MSI by a single Euler tour.

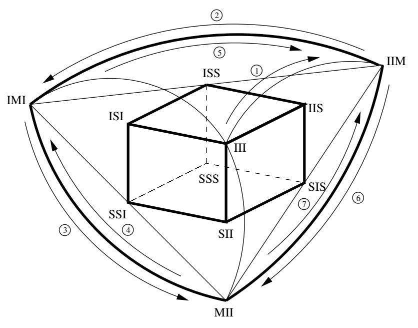
In fact, the problem to minimize the number of reused transitions is similar to Chinese Postman Problem (CPP) [4], which can be solved by calculating the min-cost max-flow. Since we need to perform the test generation on-the-fly, finding the optimal solution by solving CPP is not an option, because the state space can be too large to fit into memory when there are many cores. Instead, we visit the uncovered transitions to global modified states 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 or global invalid state
 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.

Example 2: We execute $VisitClique$ in the state space shown in Figure 4. We first cover the transition III-IIM and IIM-IMI in line 7 and line 10. In the recursive call of $VisitClique$ in line 12, the transitions IMI-MII and MII-IMI are visited. Next, the 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. The detailed steps and corresponding test sequence of $VisitClique$ are shown in Figure 4, with transitions to bidirectionally reachable shared states omitted for simplicity. Finally, in line 3-6 we visit all uncovered transitions from global shared states to global modified states. Notice that we do need to run Dijkstra's algorithm to find the shortest path in line 6. For example, there is a transition from SSS to IMI but no transition from IMI to SSS. So, SSS-IMI is not covered by Line 8 which are not bidirectionally reachable. As we are in a global modified state after executing the store operation in line 5, we can simply go back to III and perform the corresponding load operations (three load operations are required for SSS) to reach the expected global shared state. ■



(a)

No.	Core	Instr.
1	0	store
2	1	store
3	2	store
4	1	store
5	0	store
6	2	store
7	0	store

(b)

Fig. 4. (a) 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. $VisitClique(0)$ in Algorithm 2 recursively perform an Euler tour in the clique of all global modified states. The arrows are the steps to traverse the transitions with numbers to indicate the sequence. (b) The test sequence generated by $VisitClique(0)$.

C. MESI Protocol

In MESI protocol, a cache block goes to exclusive (E) state when it is the first one to load a memory address. In a system with n cores, there are n global exclusive states[2]. Figure 5a 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). Algorithm 3 presents our test generation

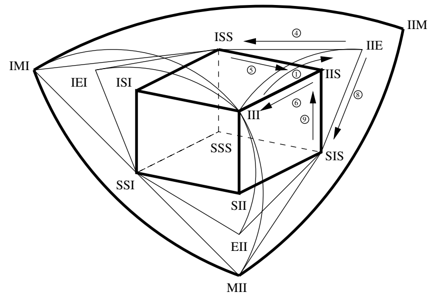
[2]A **global exclusive state** is a global state with a cache block in exclusive state (e.g., IIE, IEI, and EII in Figure 5a).

procedure for MESI protocol. $VisitE$ in Algorithm 3 covers the transitions between the hypercube and global exclusive states.

Example 3: We execute the first iteration ($i = 0$) of $VisitE$ in the state space shown in Figure 5a. The first iteration of inner loop ($j = 0$) covers the transition III-IIE. The second iteration ($j = 1$) covers the transitions IIE-ISS, ISS-IIS and IIS-III. The last iteration ($j = 2$) covers the transitions IIE-SIS, SIS-IIS and IIS-III. The detailed steps and corresponding test sequence are shown in Figure 5. ■

Algorithm 3 Test generation for MESI protocol with n cores

```
$CreateTestsMESI(n)$
 1: $CreateTestsSI(n)$ /* Invoke modified Algorithm 1 */
 2: $VisitClique(0)$ /* Invoke VisitClique() Algorithm 2 */
 3: $VisitE()$
 4: for each global shared/exclusive state $s$ do
 5:     for $i = 0$ to $n - 1$ do
 6:         Output “store(i)”
 7:         Output the shortest path from current state to $s$

$VisitE()$
 8: for $i = 0$ to $n - 1$ do
 9:     for $j = 0$ to $n - 1$ do
10:         Output “load(i)” /* $GI \rightarrow GE$ */
11:         Output “load(j)” /* $GE \rightarrow GS$ */
12:         if $i \neq j$ then
13:             Output “evict(j)”
14:             Output “evict(i)” /* $GS \rightarrow GI$ */
15: return
```



(a)

No.	Core	Instr.
1	0	load
2	0	load
3	0	load
4	1	load
5	1	evict
6	0	evict
7	0	load
8	2	load
9	2	evict
10	0	evict

(b)

Fig. 5. (a) State space of MESI protocol with 3 cores. The hypercube and clique are highlighted. The first iteration of $VisitE()$ ($i = 0$) in Algorithm 3 covers transitions between the hypercube and global exclusive states. Notice that repeated transitions are labelled with their first sequence numbers. (b) The test sequence generated by $VisitE()$.

D. MOSI Protocol

The MOSI protocol contains a new state “owned” (O), which can be used to avoid unnecessary writeback to memory. A cache block in the modified state is converted to the owned state, when other cores are trying to load the same cache block. The owned state can coexist with shared and invalid states. As a result, for a system with n cores, there are $n * 2^{n-1}$ global owned states[3]. Considering the fact that there are only $n + 2^n$ global states in MSI protocol with n cores, the state space of MOSI is much larger. Despite the large number of states, the state space structure of MOSI protocol is not complex. The entire space can be divided into three parts. The first and second parts are the hypercube of global shared states and the clique of global modified states, respectively. They are identical to corresponding structures in MSI protocol. The third part is a set of n hypercubes with dimension $n - 1$. Each of the $n - 1$ dimensional hypercubes consists of 2^{n-1} global owned states, whose state vectors have an “O” in the same position. For example, Figure 6a shows the state space with three cores. It is easy to see that states (IOI,IOS,SOS,SOI) (IIO,SIO,SSO,ISO) and (OII,OSI,OSS,OIS) are composed of three two-dimensional hypercubes ,i.e., squares.

Observation 2: There is no transition among the n hypercubes of global owned states.

As a result of Observation 2, a large number of transitions between global owned states can be efficiently covered. We can perform an Euler tour in each $n-1$ dimensional hypercube by invoking routine $CreateTestsSI$ on global owned states like IIO, IOI and OII, where all but one core are in invalid state. In order to cover transitions from global owned states to global shared states, like IOS-IIS, we have to use a similar technique that was used in $CreateTestsMSI(n)$ to cover the store transitions. Algorithm 4 presents our test generation procedure for MOSI protocol.

Example 4: We execute the first iteration ($i = 0$) of $VisitO$ in the state space shown in Figure 6a. Line 9-11 ensures the ith core in O state and the others in I state, which covers III-IIM, IIM-ISO, and ISO-IIO in this case. Then we iterate over the remaining $n-1$ cores in the inner loop. Line 13-15 choose the jth core of the $n-1$ cores, which is the pth core among all cores. During the first iteration of the inner loop ($j = 0$), IIO-ISO is covered in line 16. The function of $VisitOHypercubes$ is similar to $VisitHypercube$ except that $VisitOHypercubes$ performs an Euler tour on the $n - 2$ dimensional hypercube by fixing the ith core in O state and the pth core in S state. When $j = 0$ ($p = 1$), it covers ISO-SSO and SSO-ISO. Then ISO-IIO is traversed in line 18. The second iteration of inner loop ($j = 1$) covers IIO-SIO-SSO-SIO-IIO. The detailed steps and corresponding test sequence are shown in Figure 6. ■

E. MOESI Protocol

A more complicated protocol MOESI can be obtained by adding the exclusive (E) state to MOSI. Since the exclusive state can only be converted to global shared states or global modified states, there is no transition between the global exclusive states and global owned states. The test generation approach for MOSI protocol can be easily adapted to MOESI protocol using the same modifications as we discussed in Section IV-C for MESI protocol. Algorithm 5 presents our test generation procedure for MOESI protocol.

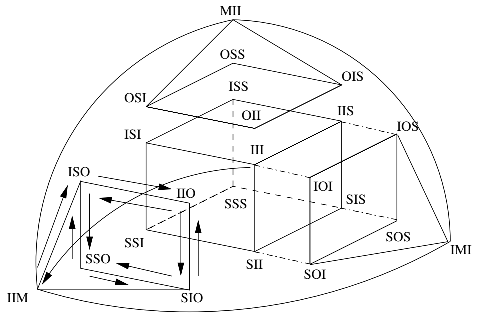
[3]A **global owned state** is a global state with a cache block in owned state (e.g, IOI, IOS, ... , OSS in Figure 6a).

Algorithm 4 Test generation for MOSI protocol with n cores

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

$VisitO()$
8: for $i = 0$ to $n-1$ do
9:     Output "store(i)"
10:    Output "load($(i+1) \bmod n$)"
11:    Output "evict($(i+1) \bmod n$)"
12:    for $j = 0$ to $n-2$ do
13:        $p = j$
14:        if $p \geq i$ then
15:            $p = p + 1$
16:        Output "load(p)"
17:        $VisitOHypercubes(1, n-2, j, i)$
18:        Output "evict(p)"
19: return

$VisitOHypercubes(state, m, shift, opos)$
20: for $i = 1$ to $m$ do
21:    $newid = state + (1 << i)$
22:    $p = (i + shift) \bmod (n-1)$
23:    if $p \geq opos$ then
24:        $p = p + 1$
25:    Output operations to visit all bidirectionally reachable
       global shared states
26:    Output "load(p)"
27:    if $i > 1$ then
28:        $VisitOHypercubes(newid, i-1, shift, opos)$
29:    Output "evict(p)"
30: return
```

Algorithm 5 Test generation for MOESI protocol with n cores

```
$CreateTestsMOESI(n)$
1: $CreateTestsSI(n)$ /* Invoke modified Algorithm 1 */
2: $VisitClique(0)$ /* Invoke $VisitClique()$ in Algorithm 2 */
3: $VisitE()$ /* Invoke $VisitE()$ in Algorithm 3 */
4: $VisitO()$ /* Invoke $VisitO()$ in Algorithm 4 */
5: for each global shared/exclusive/owned state $s$ do
6:     for $i = 0$ to $n-1$ do
7:         Output "store(i)"
8:         Output the shortest path from current state to $s$
```



No.	Core	Instr.
1	0	store
2	1	load
3	1	evict
4	1	load
5	2	load
6	2	evict
7	1	evict
8	2	load
9	1	load
10	1	evict
11	2	evict

(b)

Fig. 6. (a) State space of MOSI protocol with 3 cores. The first iteration of $VisitO()$ ($i = 0$) in Algorithm 4 covers transitions of the hypercube in the bottom left. (b) The test sequence generated by the first iteration of $VisitO()$.

V. Scalable Test Generation using Quotient Space

Since the number of states in coherence protocols grows exponentially as the number of core increases, it may not be realistic to cover all possible transitions of many-core designs within given verification budget. A widely used technique to address this limitation is to perform verification on quotient space. By grouping states into equivalent sets and checking only the representative state per set, the total validation effort is greatly reduced by eliminating similar transitions. However, since the original state space is prohibitively large to explore, validation on quotient space still faces two critical challenges: 1) how to maximize the utilization of each transition by avoiding revisit of the same transition unnecessarily, and 2) how to make the test simulation/execution time configurable to provide trade-off between state/transition coverage (confidence) and verification budget (available time).

In this section, we are going to address these challenges by extending our on-the-fly test generation techniques (discussed in Section IV) to support test generation for many-core coherence protocols using quotient space. For the ease of presentation, we are going to employ several group theory terminology in the following discussion.

Definition 3: Let X be a finite set. A permutation of X is a bijection from X to X. The set of all permutations of X forms a group under composition of mappings. Any subgroup of this group is called a *permutation group* acting on the set X. We denote permutations using cycle notation. For example, $G_0 = (0,2)(1,3,4)$ acting on $X_0 = \{a_0, a_1, a_2, a_3, a_4\}$ repeatedly performs the following permutation: $a_0 \to a_2$, $a_2 \to a_0$, $a_1 \to a_3$, $a_3 \to a_4$, $a_4 \to a_1$.

Definition 4: Given a permutation group G acting on a finite set X, for $x \in X$ the set $\{\pi(x) : \pi \in G\}$ is called the *orbit* of x under G, denote $[x]_G$. $G_0 = (0,2)(1,3,4)$ divides X_0 into two orbits: $\{a_0, a_2\}$ and $\{a_1, a_3, a_4\}$.

Given a set of nodes and a permutation group, we define *orbit state* as follows: the orbit is in

1) I state, if all nodes in the orbit are in I state.
2) S state, if all nodes in the orbit only contain I or S state, and at least one node is in S state.
3) E state, if at least one node in the orbit is in E state.
4) O state, if at least one node in the orbit is in O state.
5) M state, if at least one node in the orbit is in M state.

Let $[\boldsymbol{s}]_G$ be the global orbit state of $\boldsymbol{s}$, where each element of $[\boldsymbol{s}]_G$ is the state of corresponding orbit. We use α to denote the number of orbits.

Definition 5: The *quotient protocol* P_G of protocol P with respect to permutation group G is a tuple $P_G = (S_G, T_G)$, where $S_G = \{[\boldsymbol{s}]_G : \boldsymbol{s} \in S\}$ (S is the state space of P), $T_G = \{([\boldsymbol{s}]_G, [\boldsymbol{t}]_G) : (\boldsymbol{s}, \boldsymbol{t}) \in T\}$ (T is the transition rule of P). We denote the quotient protocol of certain standard protocol by adding prefix ‘P’ to the protocol name. For instance, the quotient protocol of MSI is denoted as PMSI.

Theorem 5.1: The state space of quotient protocol PSI with n cores and α orbits is equivalent to the state space of an SI protocol with α cores.

Proof: First, it is easy to see that the global orbit state $[\boldsymbol{s}]_G$ contains α elements, which is the same as the number of elements in the state of SI protocol with α cores. Then, we prove that every state/transition in SI protocol also exists in PSI protocol.

For any state $\boldsymbol{s}$ in the state space of SI protocol, suppose $s_{i_1}, s_{i_2}, ..., s_{i_k}$ = S. To achieve the corresponding state in PSI protocol, we can randomly choose one node from each orbit i_1, i_2, ..., i_k and let their states be S. So, every state in SI protocol exists in PSI protocol.

For any transition from state $\boldsymbol{s}$ to $\boldsymbol{s'}$ in SI, $\boldsymbol{s'}$ either is $\boldsymbol{s}$ itself, or contains one different element. Suppose s_j = S and s'_j = I. To get the corresponding transition in PSI protocol, we first construct the corresponding state of $\boldsymbol{s}$ by the above method. As the above method makes at most one node in each orbit to be in S state, suppose node t in orbit j is in S state. A transition $\boldsymbol{s}$ to $\boldsymbol{s'}$ is achieved by the evict operation of node t. Similarly, we can get corresponding transition if s_j = I and s'_j = S. So, every transition in SI protocol exists in PSI protocol.

We can follow the similar arguments to prove that every state/transition in PSI protocol also exists in SI protocol. Therefore, the state space of quotient protocol PSI with n cores and α orbits is equivalent to the state space of an SI protocol with α cores. ■

Example 5: Consider PSI protocol with 3 cores and permutation group $G = (0, 1)(2)$. {II} in PSI represents {III} in SI, {IS} in PSI represents {IIS, ISI, ISS} in SI, {SI} in PSI represents {SII} in SI, and {SS} in PSI represents {SIS, SSI, SSS} in SI. Figure 7a shows the state space of the original SI protocol with 3 cores and Figure 7b shows the corresponding PSI protocol. It is easy to verify that the state space of PSI is equivalent to that of SI protocol with 2 cores. Every transition in PSI is a Cartesian product of the respective set of states (excluding invalid transitions) in the SI protocol. For example, II-IS in PSI represents {III-IIS, III-ISI, III-ISS} in SI protocol. Therefore, if we traverse II-IS in PSI protocol, we can guarantee that we have covered one of the transitions in {III-IIS, III-ISI, III-ISS} in SI protocol. In other words, we mark {III-IIS, III-ISI, III-ISS} as similar transitions, and want to cover the representative of the three transitions within verification budget.

■

We can prove similar arguments for PMSI vs MSI protocol. However, it is important to note that the state space of PMESI, PMOSI and PMOESI are no longer exactly the same as that of MESI, MOSI and MOESI protocols, respectively. This is because there are more transitions in the quotient protocol than the original one. For example, in a system with 4 nodes and permutation group $G = (0, 1)(2, 3)$, IIEI to IISS, IISO to IISI and IIIM to IISO will look like IE to IS, IO to IS, IM to IO, respectively, from the state space of quotient protocols. We call these transitions as *extra transitions*. Fortunately, the number of extra transitions is only $O(\alpha)$, which does not change the asymptotic size of the generated trace.

The total number of transitions can be computed by adding the extra transitions to the original transitions. The results are shown in Figure 8. As we can see, the number of transitions increase exponentially with the number of orbits α.

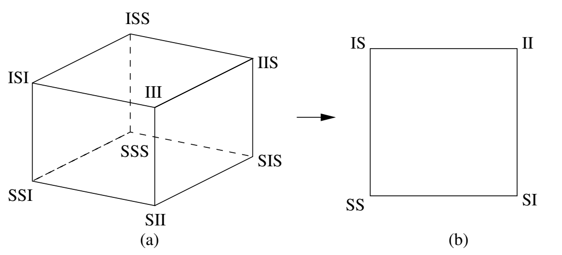


Fig. 7. (a) SI protocol with 3 cores. (b) PSI protocol with 3 cores and permutation group $G = (0, 1)(2)$.

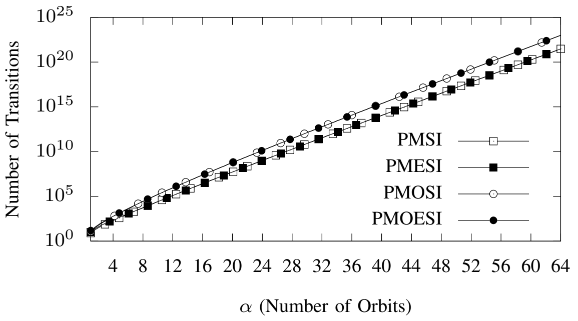


Fig. 8. Complexity of quotient protocol with respect to number of orbits α. In this figure, there are overlapping lines - (PMSI and PMESI) as well as (PMOSI and PMOESI), because their number of transitions are very similar.

The basic idea for verification using quotient protocol is to mark certain states as equivalent, i.e., acting permutation group G to partition the original nodes into α orbits, then define state on orbits. Transitions in quotient space are representatives of similar transitions in equivalence classes. When choosing α to be equal to the number of cores, full coverage is guaranteed. However, the number of total transitions grows so quickly that for large number of cores, it is unrealistic to verify all transitions, even using directed tests (one-to-one mapping between transitions and instruction sequences). Our quotient protocol identifies equivalence classes and selects the transitions to trade-off between transition coverage and validation time. For fixed number of cores, choosing larger number of orbits (α) means covering exponentially more representative transitions in the original protocol space, but it comes at the cost of increased validation effort. If we can cover all states and transitions in the quotient protocol with a test suite, the same test suite should be able to cover the most important transitions in the original protocol. The advantage of using orbits lies in the flexibility of grouping "similar" states. The way of forming orbits can be changed based on the verification budget and the functionality of the cores. In order to increase the probability of covering the transitions of an important node, we may construct one orbit containing the important core, and group the rest randomly.

To illustrate how to perform test generation using quotient protocol, we start our discussion from SI protocol. For simplicity, we assume that the n nodes are evenly partitioned into α orbits, i.e., choose $G = (0, ..., k-1)(k, ..., 2k-1)...((\alpha-1)k, ..., n-1)$ where $k = \lceil n/\alpha \rceil$. We use $R(\boldsymbol{s})$ to represent the global orbit state of $\boldsymbol{s}$, where p^{th} element of $R(\boldsymbol{s})$ is

$$R_p(\boldsymbol{s}) = \begin{cases} S & \exists\, pk \le i < (p+1)k,\ s_i = S \\ I & otherwise \end{cases}$$

It is easy to see that $R(\boldsymbol{s})$ has α elements and the p^{th} element of $R(\boldsymbol{s})$ is shared if and only if there exists i such that $pk \le i < (p+1)k$ and s_i is shared. Conceptually, quotient protocol PSI reduces the number of states by performing an "or" operation per k nodes in the original global state. Since PSI is also an SI protocol, we can simply apply Algorithm 1 to generate efficient transition sequence to cover all states and transitions of PSI. However, these transitions are "abstract" transitions in the quotient state space. They cannot be directly executed or simulated in the actual design. Therefore, we design Algorithm 6 to generate corresponding feasible transitions for original protocol SI.

Algorithm 6 Test generation for quotient SI protocol with n cores

```
$CreateTestsSI(n)$
1: for $r = 0$ to $\alpha - 1$ do
2:     $VisitHypercube(\alpha, r)$

$VisitHypercube(m, r)$
3: $p = (m + r) \bmod \alpha$
4: $q = rand(k)$
5: Output "load($pk + q$)"
6: for $i = 1$ to $m - 1$ do
7:     $VisitHypercube(i, r)$
8: Output "evict($pk + q$)"
9: return
```

The difference between Algorithm 6 and Algorithm 1 is the randomness introduced in "load" and "evict" operations using $rand(k)$ which returns i.i.d. random integers uniformly distributed between 0 and $k-1$. It is introduced to provide fairness among equivalent states. If we view the generated transition sequence from the state space of PSI, the sequence corresponds to a deterministic Euler tour of PSI's state space (Theorem 5.2). The randomness introduced by $rand(k)$ does not affect the transition, because $pk + rand(k)$ actually belongs to the same orbit regardless of the return value of $rand(k)$. Therefore, the generated sequence covers the entire state space of PSI with no wasted transitions.

Theorem 5.2: The test sequence constructed by Algorithm 6 does perform an Euler tour of quotient protocol PSI's state space.

Proof: First, notice the fact that $VisitHypercube(i, r)$ in line 7 of Algorithm 6 does not change the state of any s_i with $pk \le i < (p+1)k$. This is because its first argument must be strictly less than m. It is impossible that local variable p takes the same value inside the recursion.

It is also straightforward to see that if $R_p(\boldsymbol{s}) = I$,

$$R_p(load(pk + q) \circ \boldsymbol{s}) = S$$

$R_p(\boldsymbol{s}) = I$ implies $\forall s_i$ with $pk \le i < (p+1)k$ must be in I state. Performing a load operation on any of them will force one and only one core into S state.

Since $VisitHypercube(i, r)$ in line 7 does not affect any s_i with $pk \le i < (p+1)k$, and

$$R_p(evict(pk + q) \circ load(pk + q) \circ \boldsymbol{s}) = I$$

We can see that line 8 of Algorithm 6 reverses the transition performed by the load operation in line 5.

Therefore, if we apply R to the global state $\boldsymbol{s}$ after each operation in Algorithm 6, the sequence of $R(\boldsymbol{s})$ would be the same as the state transition sequence triggered by Algorithm 1. Since the state space of PSI is a hypercube, we can conclude that the test sequence generated by Algorithm 6 does perform an Euler tour. ■

The space complexity of Algorithm 6 is linear with the number of orbits α, because this algorithm requires a stack with at most $\alpha - 1$ levels. The time complexity is linear to the number of transitions $\alpha * 2^{\alpha}$. Clearly, Algorithm 6 is asymptotically faster than Algorithm 1 which has time complexity of $O(n2^n)$. This is obvious considering that PSI has asymptotically smaller state space with only 2^{α} states.

Algorithm 2 can be modified by adding randomization within orbits to generate efficient transition sequence with minimum wasted transitions for PMSI. Although PMESI, PMOSI and PMOESI are no longer strict MESI, MOSI and MOESI protocols, respectively, our test generation algorithms for MESI, MOSI and MOESI protocols can also be modified to support the quotient version by taking care of the extra transitions with additional efforts. As the number of extra transitions is only $O(\alpha)$, the asymptotic size of the generated traces does not change.

VI. Experiments

A. Experimental Setup

To analyze the effectiveness of our proposed test generation framework, we conducted a number of experiments using Gem5 simulator [24] with Ruby memory subsystem. As we generate our test vectors to cover cache states and transitions using a certain path, correctness is verified by checking that the simulation using these vectors traverses the cache states and transitions following this exact same path. Ruby memory subsystem implements MESI and MOESI cache coherence protocols by default, and provides interface to define other protocols. The detailed parameters for the simulation is shown in Table II.

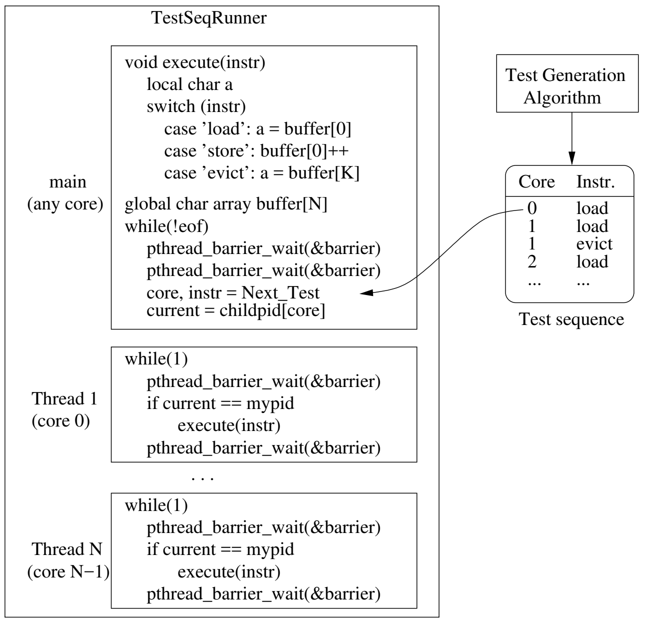
TABLE I
STATISTICS OF OUR TEST GENERATION ALGORITHM FOR DIFFERENT CACHE COHERENCE PROTOCOLS

	# States	# Transitions	BFS			Our proposed on-the-fly approach			
			Total cost (transitions)	Average cost per transition	Generation time (sec)	Total cost (transitions)	Average cost per transition	Improve factor	Generation time (sec)
MSI 8 cores	264	5256	36896	7.0	< 0.1	14664	2.8	60.3%	< 0.1
MESI 8 cores	272	5392	37712	7.0	< 0.1	15312	2.8	59.4%	< 0.1
MOSI 8 cores	1288	26248	196400	7.5	0.1	100975	3.8	48.7%	0.2
MOESI 8 cores	1296	26384	197216	7.5	0.1	101623	3.8	48.6%	0.2
MSI 16 cores	65552	2621968	29100096	11.1	7.6	11567888	4.4	60.2%	18.2
MESI 16 cores	65568	2622496	29103264	11.1	6.2	11570464	4.4	60.2%	15.1
MOSI 16 cores	589840	23855632	275254368	11.5	67.5	131122783	5.5	52.4%	183
MOESI 16 cores	589856	23856160	275257536	11.5	78.1	131125359	5.5	52.4%	216

TABLE II
GEM5 SIMULATION PARAMETERS

parameter	value
architecture	X86
cpu type	timing
clock frequency	1GHz
ruby	true
instruction cache size (L1)	4kB
data cache size (L1)	4kB
L1 hit latency	2ns
cache line size	4
memory size	4GB
number of cores	8, 16, 32, 64
debug flag	ProtocolTrace

The overview of our evaluation framework is shown in Figure 9. We first use our test generation algorithms to generate the load/store/evict sequence and expected state sequence. The output sequence is fed into a program (TestSeqRunner in Figure 9) that we have designed to run inside the Gem5 and Ruby framework. TestSeqRunner is compiled using gcc with m5threads (to support barriers to synchronize all the threads) to run in Gem5. For N-core system, we create $N+1$ threads: the main thread to read from the output of our algorithm, and the other N threads (one in each core) to execute the designated load/store instructions. For example, when the main thread gets an instruction "0, load" from the test sequence, it will set $current$ to be the pid of thread 1 (run in core 0). When the first *pthread_barrier_wait* of main thread is executed, it will wait on the second *pthread_barrier_wait*. At the same time, all the other threads will execute the *if* statement after their first *pthread_barrier_wait*. Only thread 1 will execute the load instruction as $current$ matches its pid. After all the threads finish the *if* statement, the main thread will move on and read the next instruction.



```
void execute(instr)
    local char a
    switch (instr)
        case 'load': a = buffer[0]
        case 'store': buffer[0]++
        case 'evict': a = buffer[K]
global char array buffer[N]
while(!eof)
    pthread_barrier_wait(&barrier)
    pthread_barrier_wait(&barrier)
    core, instr = Next_Test
    current = childpid[core]
```

```
while(1)
    pthread_barrier_wait(&barrier)
    if current == mypid
        execute(instr)
    pthread_barrier_wait(&barrier)
```

```
while(1)
    pthread_barrier_wait(&barrier)
    if current == mypid
        execute(instr)
    pthread_barrier_wait(&barrier)
```

```
Core  Instr.
0     load
1     load
1     evict
2     load
...   ...
```

Fig. 9. Evaluation framework of our experiment. Test sequence is generated by our test generation algorithms for a given protocol. TestSeqRunner read one test during each loop, and ask the expected thread to execute the corresponding *load/store/evict* operation. TestSeqRunner is compiled using gcc with m5threads to run inside Gem5 simulator.

We monitor the cache behaviors with respect to a certain memory location. We first initialize an array that is larger than our cache. As shown in Figure 9, we first define a global char array $buffer$, and $buffer[0]$ is our location of interest. The *load* and *store* are done by reading from and writing to $buffer[0]$, respectively. While the *evict* operation is achieved by loading a different memory address $buffer[4096]$ which is also mapped to the same location in the cache as the cache block under test. Since the cache size is 4K bytes, having the least significant 12 bits being the same ensures that $buffer[0]$ and $buffer[4096]$ map to the same cache block. The protocol trace of Gem5 contains intermediate states, which are not considered in our approach. So we remove these states before comparing with the expected outputs.

The remainder of this section is organized as follows. First, we present coverage results (by choosing the number of orbits (α) to be the number of cores) using on-the-fly test generation techniques outlined in Section IV. Then, we present the results using quotient space (by varying α) to trade-off between functional coverage and verification efforts.

B. On-the-Fly Test Generation

In the first experiment, we choose the number of cores to be small (less than 16) so that a full coverage is possible. We compared the efficiency of our test generation method with the tests generated by performing breadth-first search directly on the global FSM for different cache coherence protocols. 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" presents the total number of transitions traversed to activate all transitions, i.e., the total number of load/store/evict instructions of the generated tests . 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, as our approach traverses the regular structures efficiently.

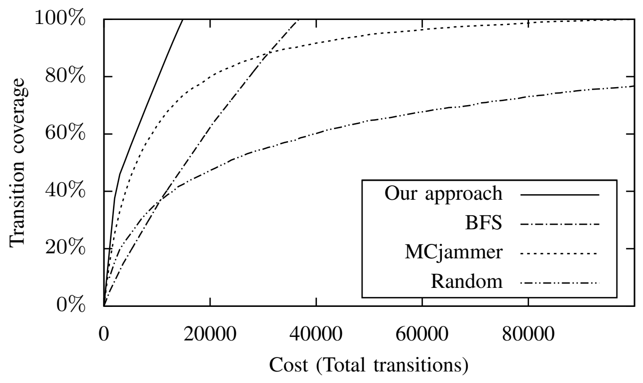


Fig. 10. Transition coverage vs. cost for MESI protocol with 8 cores.

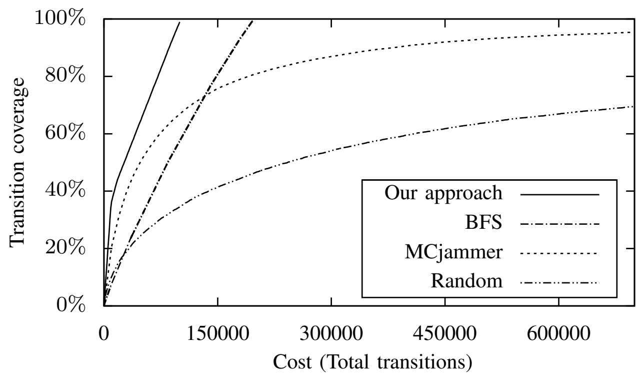


Fig. 11. Transition coverage vs. cost for MOSI protocol with 8 cores.

We also compared the state and transition coverage of our test generation approach with a directed random test generator, MCjammer [18]. Figure 10 and Figure 11 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.

The test generation time in Table I indicates that the runtime of our algorithms is reasonable. For MOESI protocol with 23 million transitions, we can create all the tests within 4 minutes, which indicates that our algorithm is quite light-weight for entire simulation based validation phase. Although the test generation time of our algorithms is up to 3 times slower than BFS, the difference is negligible compared to the simulation time improvement which is an advantage of our shorter test sequence.

C. Memory Usage Comparison

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. To show the memory usage of our approach compared to others, we compile each algorithm using g++ 5.4.0 with no optimization. The memory usage information is gathered by Valgrind 3.11.0. The results are shown in Table III. Since BFS needs to remember all the states that are already visited, the memory usage grows fast over time. At the end of BFS, every state should be marked as visited, which increases the memory requirement exponentially as the number of cores increases. However, the memory requirement of our approach depends on the number the recursions and does not increase over time. Intuitively, as the Euler traversals we proposed are deterministic, the next transition can be determined by current transition as shown in Figure 3. By inspecting the algorithms, our algorithm does not keep any additional information about the states and transitions that are already covered, except for the stacks incurred by recursive calls. Therefore, the memory requirement grows linearly as the number of cores increases.

TABLE III
THE MEMORY USAGE COMPARISON FOR OUR APPROACH WITH BFS

	BFS (KB)			Our approach (KB)		
# cores	4	8	16	4	8	16
MSI	77.5	115.2	12698	75.1	75.1	75.2
MESI	77.5	115.2	12698	75.1	75.1	75.2
MOSI	80.2	236.6	101511	75.1	75.1	75.2
MOESI	80.6	236.6	101511	75.1	75.1	75.2

D. Test Generation for Quotient Protocol

Transition coverage in quotient state space is an effective way for test size reduction. With our quotient space based test generation techniques, verification engineers can pick the number of orbits α according to their verification budget and protocol complexity without losing any important transitions.

To compare the transition coverage in the original state space with different number of orbits (α), we vary $\alpha = 4, 8, 12, 16$ for MESI protocol with 32 cores. For $\alpha = 4, 8, 16$, we simply divided all the cores evenly into α orbits. While for $\alpha = 12$, we choose 4 orbits with 4 cores, and 8 orbits with 2 cores. The test generation time and coverage in the original space are shown in Figure 12. Note that the coverage and test generation time grow exponentially with the number of orbits. As shown in Example 5, our approach guarantees the selection of important transitions and omits similar transitions by utilizing equivalence classes.

To select the suitable α for a given verification budget, we first gather the total cost to achieve full coverage in the quotient space with different number of orbits (α). An important feature of our quotient space protocol is that it can be applied on top of any existing test generation algorithms. We configure BFS, MCjammer and Random algorithm to run on the quotient protocols. For the MCjammer and Random algorithm, the mean of multiple measurements are used to reduce the variation introduced by randomization. The experimental result is shown in Figure 13. As expected, choosing

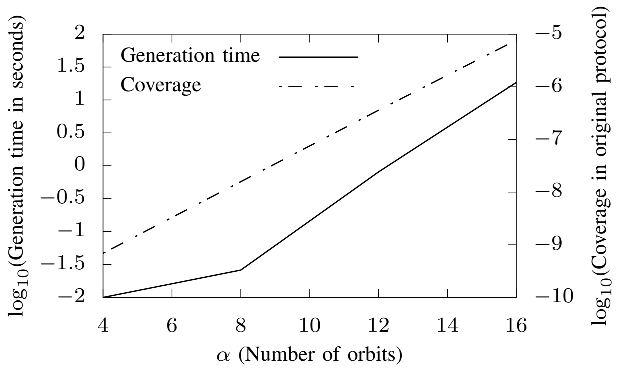


Fig. 12. Test generation and coverage in the original space (MESI with 32 cores) of PMESI protocol with different number of orbits. The left y-axis shows the logarithmic test generation time, and the right y-axis shows the logarithmic coverage in the original space.

a lower α would require less transitions to achieve 100% coverage in quotient space, but achieves exponentially smaller coverage in the original protocol space as shown in Figure 12. For the same α, our method requires the least amount of cost to achieve full coverage in quotient space, and outperforms other approaches by several orders-of-magnitude. For example, for $\alpha = 8$, our approach requires about 10^4 transitions, while BFS requires twice as much, and MCjammer and Random algorithm require about 10^5 and 10^6 transitions, respectively.

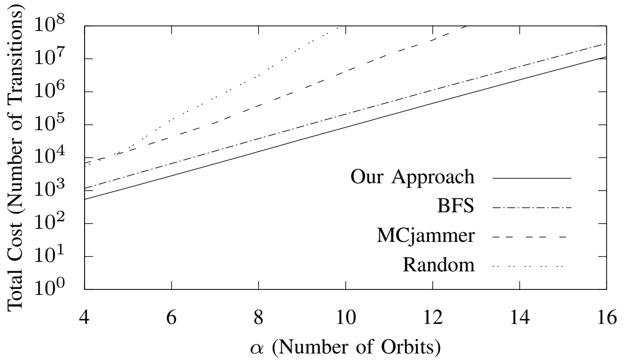


Fig. 13. Total cost vs. number of orbits (α) for PMESI protocol with 64 cores.

In the experiment, we consider MESI with 64 cores. We did not provide results for other coherence protocols since they lead to similar observations in terms of reduction in validation effort. Let us assume that the verification budget is 10^7, i.e., total number of transitions cannot exceed 10^7. Based on Figure 13, our quotient protocol PMESI chooses $\alpha = 15$. Now, we would like to compare transition coverage of our test generation approach with other approaches on quotient protocols given the same α. Figure 14 shows the relation between transition coverage and testing cost on the quotient protocol. As we can see, our test generation approach achieves full coverage quickly taking advantage of Euler traversals, while none of the existing approaches can achieve full coverage within 10^7 transitions budget and 15 orbits. Clearly, our test generation approach on quotient protocol significantly outperforms the existing test generation approaches by providing higher design quality (coverage) within specific verification budget.

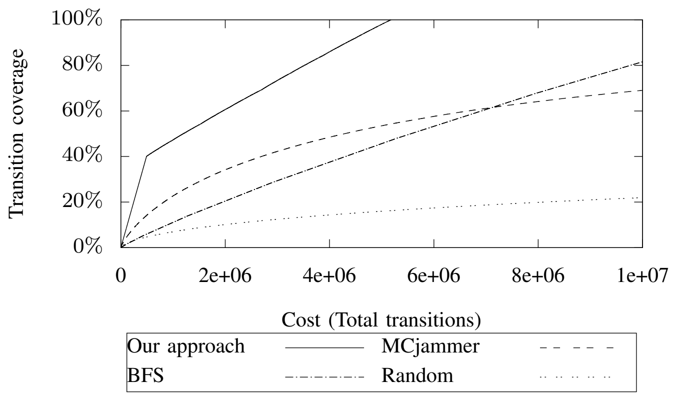


Fig. 14. Transition coverage vs. time cost for PMESI protocol with 64 cores and 15 orbits.

VII. 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. We also presented quotient space based scalable test generation algorithms that can trade-off between functional coverage and verification effort. Quotient space guarantees selection of important transitions by utilizing equivalence classes, and omits only similar transitions to provide scalable test generation framework. 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] "AMD64 Architecture Programmer's Manual Volume 2: System Programming," http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf, 2013.

[2] P. Mishra and N. Dutt, "Graph-based functional test program generation for pipelined processors," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, Feb 2004, pp. 182–187.

[3] X. Qin and P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012, pp. 3–8.

[4] J. Edmonds and E. L. Johnson, "Matching, euler tours and the chinese postman," *Mathematical Programming*, vol. 5, no. 1, pp. 88–124, 1973.

[5] M. E. Thomadakis, "The architecture of the Nehalem processor and Nehalem-EP SMP platforms," *Resource*, vol. 3, p. 2, 2011.

[6] G. Zhang, W. Horn, and D. Sanchez, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 13–25.

[7] D. Dill, A. Drexler, A. Hu, and C. Yang, "Protocol verification as a hardware design aid," in *Proc of ICCD*, 1992, pp. 522 –525.

[8] E. Emerson and V. Kahlon, "Exact and efficient verification of parameterized cache coherence protocols," in *Correct Hardware Design and Verification Methods*, 2003, vol. 2860, pp. 247–262.

[9] Y. Li, K. Duan, Y. Lv, J. Pang, and S. Cai, "A novel approach to parameterized verification of cache coherence protocols," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, Oct 2016, pp. 560–567.

[10] M. Zhang, A. R. Lebeck, and D. J. Sorin, “Fractal coherence: Scalably verifiable cache coherence,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO’43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 471–482.

[11] G. Voskuilen and T. Vijaykumar, “High-performance fractal coherence,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS’14. New York, NY, USA: ACM, 2014, pp. 701–714.

[12] M. Zhang, J. D. Bingham, J. Erickson, and D. J. Sorin, “PVCoherence: Designing flat coherence protocols for scalable verification,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 392–403.

[13] D. Sethi, M. Talupur, and S. Malik, *Using Flow Specifications of Parameterized Cache Coherence Protocols for Verifying Deadlock Freedom*. in ATVA 2014, ser. LNCS, vol. 8837.

[14] F. Verbeek, P. M. Yaghini, A. Eghbal, and N. Bagherzadeh, “Deadlock verification of cache coherence protocols and communication fabrics,” *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 272–284, Feb 2017.

[15] D. Wood, G. Gibson, and R. Katz, “Verifying a multiprocessor cache controller using random test generation,” *IEEE Design Test of Computers*, vol. 7, no. 4, pp. 13 –25, 1990.

[16] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, “Genesys-pro: innovations in test program generation for functional processor verification,” *IEEE Design Test of Computers*, vol. 21, no. 2, pp. 84 – 93, 2004.

[17] 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.

[18] I. Wagner and V. Bertacco, “Mcjammer: adaptive verification for multicore designs,” in *Proc of DATE*, 2008, pp. 670–675.

[19] M. A. P. Cunha, O. Matoussi, and F. Pétrot, “Detecting software cache coherence violations in mpsoc using traces captured on virtual platforms,” *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 2, pp. 30:1–30:21, Jan. 2017.

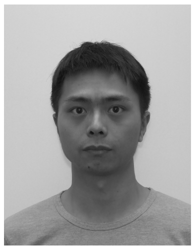
[20] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, “Exploiting symmetry in temporal logic model checking,” *Formal Methods in System Design*, vol. 9, no. 1, pp. 77–104, 1996.

[21] E. A. Emerson and A. P. Sistla, “Symmetry and model checking,” *Formal Methods in System Design*, vol. 9, no. 1, pp. 105–131, 1996.

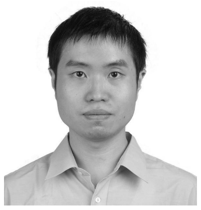
[22] A. S. Kamkin, “Projecting transition systems: Overcoming state explosion in concurrent system verification,” *Program. Comput. Softw.*, vol. 41, no. 6, pp. 311–324, Nov. 2015.

[23] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.



Yangdi Lyu received his B.E. degree in Department of Hydraulic Engineering from Tsinghua University, Beijing, China. He is currently pursuing his Ph.D. degree in the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, USA. His research interests include the area of verification, dynamic cache reconfiguration, and cache channel attack.



Xiaoke Qin received his Ph.D. degree in Computer Engineering from the University of Florida in 2012. He is currently an architect at Nvidia. His research interests are in the area of model checking and system verification.

Mingsong Chen (S’08-M’11-SM’17) received the B.S. and M.E. degrees from Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2003 and 2006 respectively, and the Ph.D. degree in Computer Engineering from the University of Florida, Gainesville, in 2010. He is currently a Professor with the Computer Science and Software Engineering Institute at East China Normal University. His research interests are in the area of design automation of cyber-physical systems, formal verification techniques and cloud computing. He is an Associate Editor of IET Computers & Digital Techniques, and Journal of Circuits, Systems and Computers.

Prabhat Mishra (S’00-M’04-SM’08) is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include design automation of embedded systems, energy-aware computing, hardware security and trust, system validation and verification, reconfigurable architectures, and post-silicon debug. He received his Ph.D. in Computer Science and Engineering from the University of California, Irvine. Prof. Mishra currently serves as the Deputy Editor-in-Chief of IET Computers & Digital Techniques, and as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems, IEEE Transactions on VLSI Systems, and Journal of Electronic Testing. He has served on many conference organizing committees and technical program committees of premier ACM and IEEE conferences. He is currently serving as an ACM Distinguished Speaker. Prof. Mishra is an ACM Distinguished Scientist and a Senior Member of IEEE.