Efficient Directed Test Generation for Validation of Multicore Architectures

Xiaoke Qin and Prabhat Mishra

Department of Computer and Information Science and Engineering
University of Florida, Gainesville FL 32611-6120, USA
{*xqin, prabhat*}@*cise.ufl.edu*

Abstract

Functional verification of multicore architectures is widely acknowledged as a major challenge. Directed tests are promising since a significantly smaller number of directed tests can achieve the same coverage goal compared to constrained-random tests. SAT-based bounded model checking is effective for automated generation of directed tests (counterexamples). While existing approaches focus on clause forwarding between different bounds to reduce the test generation time, this paper proposes a novel technique that exploits the structural similarity within the same bound as well as between different bounds. Our proposed technique enables the reuse of the knowledge learned from one core to the remaining cores in multicore architectures. The experimental results demonstrate that our approach can significantly (2-10 times) reduce overall test generation time compared to existing approaches.

I. Introduction

Multicore architectures are widely used in today's desktop and embedded computing systems to circumvent the power wall and memory wall encountered by single core architectures. While more and more cores are integrated into the same chip to boost the throughput, their increasing complexity also introduces significant verification challenge. As a result, conventional random tests based simulation becomes inadequate to achieve the required coverage within ever decreasing time-to-market window. *Directed tests are promising to solve this problem, because a drastically small number of directed tests are required to achieve the same coverage goal compared to random tests.* Unfortunately, most directed tests are currently manually written, which is time consuming and error-prone. Fully automatic directed test generation schemes are desired to accelerate the verification process of multicore architectures.

Model checking appears to be a good candidate for automatic test generation. To activate a particular scenario, we can feed the negated version of a property to the model checker, and use the resultant counterexample as a directed test. However, BDD-based symbolic model checking is not suitable for test generation involving large designs and complex properties due to the state space explosion problem. SAT-based bounded model checking (BMC) [1], [2] is proposed to address this problem, which tries to falsify a property on the states reachable from the initial state within a fixed number (k) of time steps. This is implemented by unrolling the design k times, then encoding the design and the property description as a satisfiability (SAT) problem. Next, a SAT solver is used to find a satisfying assignment for all variables (if any), which can be translated into a counterexample (a directed test).

When SAT-based BMC is applied to generate directed tests for multicore architectures, there are two different categories of symmetry in the corresponding SAT instances. The first category is the "temporal" symmetry. It occurs because the SAT instance is encoded by unrolling the same architecture for multiple times. This regularity has already been exploited by existing research [3] to accelerate the SAT solving process. On the other hand, the structural similarity of multiple cores also introduces a second category of symmetry or "spatial" symmetry. This symmetry appears among the CNF clauses for different cores at the same time step. Intuitively, we can also exploit spatial symmetry by reusing the knowledge obtained from one core to other cores. Unfortunately, this intuitive reasoning is hard to implement because it is very difficult to reconstruct the symmetry from the CNF formula. The high level information is lost during CNF synthesis, and it is inefficient as well as computationally expensive to recover through "reverse engineering" methods.

In this paper, we address the directed test generation for multicore architectures by developing a novel BMC based test generation technique, which enables the reuse of learned knowledge from one core to the remaining cores in the multicore architecture. Instead of direct synthesis of the CNF for the multicore design, we compose the CNF description of the entire design using CNF formulae for cores and the memory subsystem. Since the CNF representation of cores are generated by performing variable substitution of the CNF for one of them, the correct mapping information is easily obtained. In this way, we are able to translate and reuse the conflict clauses learned on any core to other cores. We prove that the CNF description generated by our approach has the same satisfiability as original methods. Our experimental results demonstrate that our approach can remarkably reduce the overall test generation time.

The rest of the paper is organized as follows. Section 2 describes related work on BMC and directed test generation. Section 3 briefly discusses the background on SAT-based BMC. Section 4 describes our test generation methodology for multicore architectures. Section 5 presents our experimental

results. Finally, Section 6 concludes the paper.

II. Related Work

Model checking techniques are promising for functional verification and test generation for complex systems [4], [5]. Due to the state explosion problem, conventional symbolic model checking approaches are not suitable for large designs. SAT-based bounded model checking is introduced by Biere et al. [1] as an alternative solution. Although BMC cannot prove the validity of a safety property to hold globally when no counterexample is found within a specific bound, it is quite effective to falsify a design when the bound is not large. The reason is that SAT solvers usually require less space and time than conventional binary decision diagram based model checkers [6]. Therefore, SAT-based BMC is suitable for directed test generation, where a counterexample typically exists within a relatively small bound.

A great deal of work has been done to reduce the SAT solving time during BMC [3], [7], [8], [9]. The basic idea is to exploit the regularity of the SAT instances between different bounds. For example, incremental SAT solvers [7], [8] reduce the solving time by employing the previously learned conflict clauses. Generated conflict clauses are kept in the database as long as the clauses which led to the conflicts are not removed. Strichman [3] proposed that if a conflict clause is deduced only from the transition part of a SAT instance, it can be safely forwarded to all instances with larger bounds, because the transition part of the design will still be in the SAT instance when we unroll the design for more times. Besides, the learned conflict clause can also be replicated across different time steps. However, the existing approaches did not exploit the symmetric structure within the same time step. In directed test generation for multicore architectures, same knowledge about the core structure needs to be re-discovered for each core independently, which can lead to significant wastage of computational power.

When BMC is applied in circuits, Kuehlmann [10] proposed that the unfolded transition relation can be simplified by merging vertices that are functionally equivalent under given input constraints. In this way, the complexity of transition relation is greatly reduced. However, since this technique was developed based on the AIG representation of logic designs, it is difficult to use it accelerate the solving process of CNF instances that are directly created from high level specifications.

Verification and validation based on high level specification are proved to be effective. For example, Bhadra et al. [11] used executable specification to validate multiprocessor systems-on-chip designs. Mishra et al. [9] proposed directed test generation based on high level specification. To accelerate the test generation process, conflict clauses learned during checking of one property are forwarded to speed up the SAT solving process of other related properties, although the bound is required as an input.

When the SAT instance contains symmetric structure, symmetry breaking predicate [12], [13], [14], [15], [16] can be used to speed up the SAT solving by confining the search to non-symmetric regions of the space. By adding symmetry breaking predicates to the SAT instance, the SAT solver is restricted to find the satisfying assignments of only one representative member in a symmetric set. However, this approach cannot effectively accelerate the directed test generation for multicore processors, because the properties for test generation are usually not symmetric with respect to each core. Thus, the symmetric regions in the entire space are usually small despite the fact that the structure of each core is identical. On the other hand, in component analysis for SAT solving, Biere et al. [17] proposed that each component can be solved individually to accelerate the solving process. However, the symmetric structure is not used at the same time for further speedup.

III. Background : SAT-based BMC

BMC is widely used to verify whether a safety property holds within a given bound [2]. The basic idea is to encode the verification problem as a propositional SAT problem by unrolling the design. Given a design M, a safety property p, and a bound k, the general formula for BMC is the following:

$$BMC(M,p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \quad (1)$$

where $I(s_0)$ is the initial state of the system, $R(s_i, s_{i+1})$ is the state transition constraint from state s_i to state s_{i+1}, and $p(s_i)$ represents whether property p holds on state s_i. This formula is satisfiable if and only if there exists a state within bound k, on which the property p does not hold. The system is usually modeled by some high level language, like SMV, and is converted to CNF format before checked by a SAT solver. Generally, high level information is lost in CNF, i.e., it is impossible to map auxiliary variables introduced during CNF encoding with high level components.

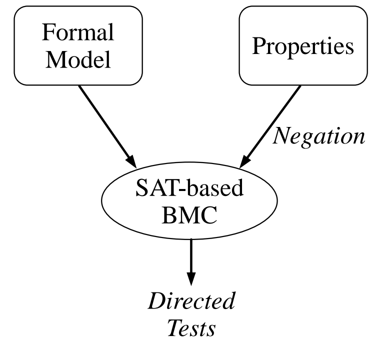

Fig. 1. Directed test generation

To generate directed tests [4], [5], we can first obtain a set of properties for the desired behaviors (faults), which should be activated in the simulation based validation stage, based on functional coverage requirements. The formal model of the system and the negated version of the property are then fed to BMC, as shown in Figure 1. For a suitable bound, the SAT solver will find a satisfying assignment for all variables, which can be then translated into an input sequence of the system. This input will drive the system from the initial state

to the desired state, which contradicts the negated version of the property. Therefore, it can be used as a test to activate the intended functionality during simulation-based validation.

Many techniques and heuristics are employed in SAT solvers to accelerate the solving process. Modern SAT solvers such as zChaff [18] adopt the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and conflict-driven non-chronological backtracking. Conflict means the current partial assignment implies that one variable must be true and false at the same time to make the entire formula true. When a conflict occurs, we have to undo some previous assignments to continue the search. At the same time, we can also add some conflict clauses into the database, which prevents the SAT solver to make the same partial assignment that can lead to the same conflict in the future. Notice that if a set of clauses S that deduce a conflict clause C have a symmetric counterpart S', we can produce the symmetric conflict clause C' by performing symmetric implications based on S'. In other words, C' can be directly added to SAT solver's database, without changing the satisfiability. This technique [3] is very effective in BMC because the transitional relations are repeated between different bounds.

IV. Test Generation for Multicore Architectures

Our work is motivated by previous works on incremental SAT-based BMC [3]. Based on the temporal symmetry between different bounds, these methods accelerate the SAT solving process by passing the knowledge (deduced conflict clauses) in the temporal direction. Nevertheless, the SAT instances generated by multicore designs also exhibit remarkable spatial symmetry. Figure 2 depicts the high level structure of a system with 2 cores. Both cores are identical[1] and connected to memory subsystem with a bus. Figure 3 shows the SAT solving process when we perform BMC for bounds 0, 1, 2, and 3 on this multicore architecture using the technique proposed in [3]. We use solid dots to represent different SAT instances and lines to indicate the conflict clause forwarding paths. Although different cores have identical structures, this spatial symmetry is not exploited.

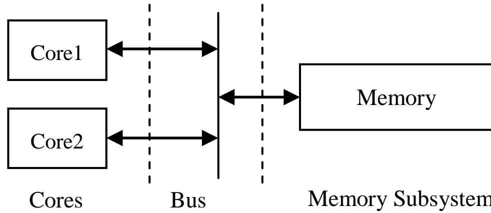

Fig. 2. Abstracted architecture of a two core system

Intuitively, it should be beneficial if the knowledge or conflict clauses can also be shared "vertically" among different cores as shown in Figure 4, because the solving effort spent on a single core can be reused by other cores to save overall time consumption. Unfortunately, the spatial symmetry is difficult to recover from the CNF representation of the SAT instance. The reason is that most clauses contain auxiliary variables introduced during the CNF encoding process. Since these auxiliary variables are unlabeled, the correspondence between clauses from different cores cannot be established directly. Although the spatial symmetry can be partially recovered by solving a graph automorphism problem [12], [13], [14], it may require impractical time for large designs, because no polynomial time solution is found for graph automorphism problem. The underlying reason for this dilemma is that the high level information is lost after the CNF encoding. In other words, a single flattened CNF SAT instance is not suitable to exploit the spatial symmetry.

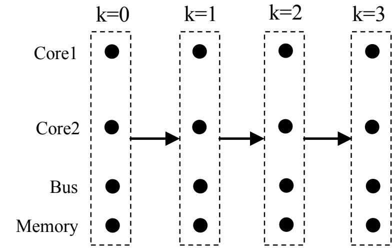

Fig. 3. Incremental SAT solving technique [3]

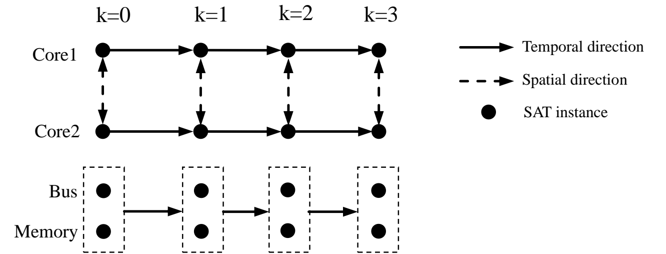

Fig. 4. Test generation for multicore architectures

Instead of using a monolithic CNF as input, our approach solves this problem by composing the CNF description of the system using CNF formulae for one core, bus and the memory subsystem. Since the cores are identical, their CNF representations are identical as well. We just need to perform variable name substitution to obtain the CNF for all other cores. As shown in Theorem 1, when the state variables are substituted by the correct names, the system CNF composed by these replicated CNF for cores, bus as well as memory subsystem will have the same satisfiability behavior as the original monolithic CNF representation. Since both the state variables and auxiliary variables in replicated cores are assigned by our algorithm, it is easy to obtain the correct mapping between variables and clauses in different cores. The spatial symmetry can then be effectively exploited during the SAT solving process. Before we describe our algorithm in details, we first introduce some notations.

Definition 1: **Symmetric Component (SC)** is a set of identical finite state machines (FSM). For the jth FSM within a SC, we denote its initial condition and transitional constraints as $I(s_{0,j}^{is})$ and $R(s_{i,j}^{is}, s_{i,j}^{in}, s_{i+1,j}^{is}, s_{i+1,j}^{out})$ $(0 \leq i \leq k-1)$,

[1] We first discuss our approach in the context of homogeneous cores. The application of our approach on heterogeneous cores will be presented in Section IV-C.

where $s_{i,j}^{in}, s_{i+1,j}^{out}, s_{i,j}^{is}$ are its input variables, output variables, and internal state variables at the ith ($i+1^{th}$) time step. It should be noted that a symmetric component itself can also be viewed as FSM, whose input and output variables are the collection of all the input and output variables of FSMs within it.

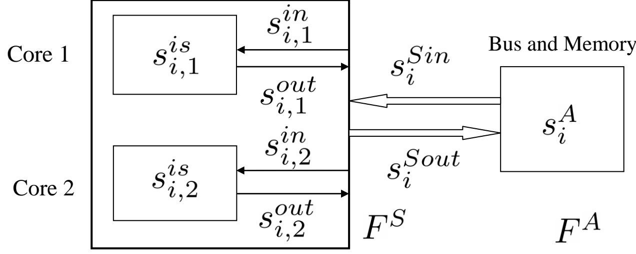

Fig. 5. FSM representation of Figure 2 at time step i

In a multicore system with N_S identical cores, we model the set of all cores as a symmetric component F^S. Other asymmetric components, such as bus and memory subsystem, are modeled as a single finite state machine F^A. We also map the input and output of F^A to the output and input of F^S so that different cores can perform communication through bus and memory subsystem. Formally, we denote the initial condition and transition constraints of F^A as $I(s_0^A)$ and $R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin})$ $(0 \leq i \leq k-1)$, where s_i^A represent internal state variables in bus and memory subsystem at the ith time step. Moreover, $s_i^{Sin} = \{s_{i,j}^{in} | 1 \leq j \leq N_S\}$ and $s_i^{Sout} = \{s_{i,j}^{out} | 1 \leq j \leq N_S\}$ are the input and output variables of the symmetric component F^S, which is the combination of the inputs and outputs of all cores. For example, Figure 5 shows the FSM representation of the system in Figure 2. The symmetric component F^S is composed of core 1 and core 2. The rest of the system is represented by F^A. In the ith time step, the internal state variable of F^S are $\{s_{i,1}^{is}, s_{i,2}^{is}\}$ and s_i^A. The input and output variables of F^S (also the output and input variable of F^A) are $s_i^{Sin} = \{s_{i,1}^{in}, s_{i,2}^{in}\}$ and $s_i^{Sout} = \{s_{i,1}^{out}, s_{i,1}^{out}\}$, respectively.

The BMC formula of the multicore system can be expressed as

$$BMC(M,p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

$$= I(s_0^A) \wedge \bigwedge_{j=1}^{N_S} I(s_{0,j}^{is}) \wedge \bigwedge_{i=0}^{k-1} (R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin})$$

$$\wedge \bigwedge_{j=1}^{N_S} R(s_{i,j}^{is}, s_{i,j}^{in}, s_{i+1,j}^{is}, s_{i+1,j}^{out})) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

The basic idea of our approach is to generate CNF formula

$$BMC'(M,p,k) = CNF_I^A \wedge \bigwedge_{j=1}^{N_S} CNF_I^S(j)$$

$$\wedge \bigwedge_{i=0}^{k-1} (CNF_R^A(i) \wedge \bigwedge_{j=1}^{N_S} CNF_R^S(i,j)) \wedge CNF^p(k)$$

Algorithm 1: Test Generation for Multicore Architectures

Input: CNF formulae CNF_I^A, $CNF_I^S(1)$, $CNF_R^A(i)$, $CNF_R^S(i,1)$, $CNF^p(k)$, Number of cores N_S, Maximum bound K_{max},
Output: Test $test_p$
Bound $k \longleftarrow 0$
Initialize variable mapping table T
Common Clause Set $CCS \longleftarrow \emptyset$
Generate $CNF_I^S(j)$ using $CNF_I^S(1)$ for $1 < j \leq N_S$
Add Clauses in $CNF_I^S(j)$ to CCS for $1 \leq j \leq N_S$
Update T
Add Clauses in CNF_I^A to CCS
while $k \leq K_{max}$ **do**
  Generate $CNF_R^S(k,j)$ using $CNF_R^S(k,1)$ for $1 < j \leq N_S$
  Add Clauses in $CNF_R^S(k,j)$ to CCS $1 \leq j \leq N_S$
  Update T
  Add Clauses in $CNF_R^A(k)$ to CCS
Step1: $(ConflictC, test_p) \longleftarrow$ SAT($CCS \bigcup CNF^p(k)$,T)
Step2: $CCS \longleftarrow CCS \bigcup$ Filter($ConflictC$)
  if $test_p \neq null$ **then** return $test_p$
  $k \longleftarrow k+1$
end

and perform SAT solving on $BMC'(M,p,k)$ instead of solving the CNF formula directly synthesized from $BMC(M,p,k)$, where CNF_I^A, $CNF_I^S(j)$, $CNF_R^A(i)$, $CNF_R^S(i,j)$ and $CNF^p(k)$ are the CNF representations of $I(s_0^A)$, $I(s_{0,j}^{is})$, $R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin})$, $R(s_{i,j}^{is}, s_{i,j}^{in}, s_{i+1,j}^{is}, s_{i+1,j}^{out})$ and $\bigvee_{i=0}^{k} \neg p(s_i)$, respectively.

Algorithm 1 shows our test generation method for multicore architectures. It accepts the CNF representation of one core, bus, the memory subsystem as well as the properties at different time steps as inputs and produces corresponding directed tests. As indicated before, we first generate the CNF representations of the initial condition and transition constraints of all other FSMs in F^S based on the input CNF formulae $CNF_I^S(1)$ and $CNF_R^S(i,1)$, which are the initial condition and transition constraints of the first FSM (Core 1). It is accomplished by replacing variable in $CNF_I^S(1)$ and $CNF_R^S(i,1)$ with corresponding variables for other FSMs (cores). At the same time, we maintain a table T [2] to record the symmetric set of variables for both state variables and auxiliary variables. After that, we invoke the SAT solving process on the conjunction of clauses in CCS and $CNF^p(k)$, which is equivalent to $BMC'(M,p,k)$ defined above. Next, we perform the following 2 steps.

1) During SAT solving, analyze any conflict clause cls found by the SAT solver. If cls is purely deduced by the clauses which belong to a single FSM, replicate and forward cls to all other FSMs. This is implemented by substituting the variables in cls by their counterparts for each FSM in F^S based on table T. At the same time, we also replicate the cls in temporal direction, as discussed in [3].

[2]As discussed in Section IV-B, a physical table is not required, instead a mapping function is used in our framework.

2) After the solving process, only keep new conflict clauses that are deduced independent of $CNF^p(k)$, and merge them into CCS.

If the satisfied assignment, or a counterexample $test_p$ is found in step 1, the algorithm returns it as a test. Otherwise, the algorithm repeats for each bound k until the maximum bound is reached.

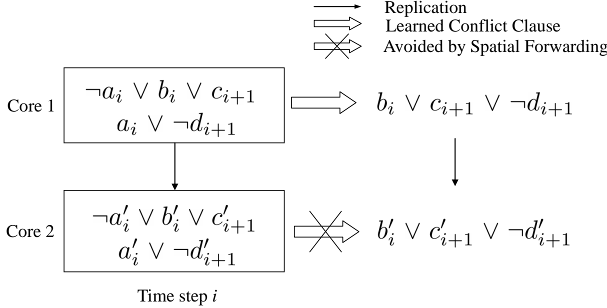

Fig. 6. Test generation for multicore architectures

We use the same example in Figure 2 to illustrate the flow of Algorithm 1. The two different clause forwarding paths employed in our approach are shown in Figure 6. Suppose $(\neg a_i \vee b_i \vee c_{i+1})$ and $(a_i \vee \neg d_{i+1})$ are two clauses within $CNF^S_R(i,1)$ (transition constraint of Core 1), in the first iteration for $k = 0$, two clauses $(\neg a'_i \vee b'_i \vee c'_{i+1})$ and $(a'_i \vee \neg d'_{i+1})$ will be produced during the generation of $CNF^S_R(i,2)$ (transition constraint of Core 2). In the subsequent SAT solving process, suppose a conflict clause $(b_i \vee c_{i+1} \vee \neg d_{i+1})$ is deduced based on $(\neg a_i \vee b_i \vee c_{i+1})$ and $(a_i \vee \neg d_{i+1})$, it will be forwarded to Core 2, because its two parent clauses are all from the CNF formula for Core 1. Therefore, $(b'_i \vee c'_{i+1} \vee \neg d'_{i+1})$ can now be used by Core 2 to prevent the partial assignment $\{b'_i, c'_{i+1}, d'_{i+1}\} = \{0, 0, 1\}$, which will result in a conflict on a'_i. Such forwarding of conflict clauses is not possible using Strichman's approach [3], which only considers temporal symmetry but not spatial symmetry.

In the remainder of this section, we prove the correctness of our approach and discuss the implementation details of our directed test generation algorithm for multicore architectures.

A. Correctness of Our Proposed Approach

To prove the correctness of our test generation approach, we need to ensure that the produced CNF formula $BMC'(M,p,k)$ in Algorithm 1 has the same satisfiability as $BMC(M,p,k)$.

Theorem 1: $BMC(M,p,k)$ and $BMC'(M,p,k)$ have the same satisfiability.

Proof: Clearly, we have

$$BMC(M,p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

$$= I(s^A_0) \wedge \bigwedge_{j=1}^{N_S} I(s^{is}_{0,j}) \wedge \bigwedge_{i=0}^{k-1} (R(s^A_i, s^{Sout}_i, s^A_{i+1}, s^{Sin}_{i+1})$$

$$\wedge \bigwedge_{j=1}^{N_S} R(s^{is}_{i,j}, s^{in}_{i,j}, s^{is}_{i+1,j}, s^{out}_{i+1,j})) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

By their definitions, CNF formulae CNF^A_I, $CNF^S_I(j)$, $CNF^A_R(i)$, $CNF^S_R(i,j)$ and $CNF^p(k)$ are CNF representation of propositional formulae $I(s^A_0)$, $I(s^{is}_{0,j})$, $R(s^A_i, s^{Sout}_i, s^A_{i+1}, s^{Sin}_{i+1})$, $R(s^{is}_{i,j}, s^{in}_{i,j}, s^{is}_{i+1,j}, s^{out}_{i+1,j})$ and $\bigvee_{i=0}^{k} \neg p(s_i)$, where $0 \leq i \leq k-1$ and $1 \leq j \leq N_S$.

Therefore, $BMC(M,p,k)$ has the same satisfiability as

$$BMC'(M,p,k) = CNF^A_I \wedge \bigwedge_{j=1}^{N_S} CNF^S_I(j)$$

$$\wedge \bigwedge_{i=0}^{k-1} (CNF^A_R(i) \wedge \bigwedge_{j=1}^{N_S} CNF^S_R(i,j)) \wedge CNF^p(k)$$

because the auxiliary variables introduced during CNF conversion do not change the satisfiability. In other words, $BMC(M,p,k)$ and $BMC'(M,p,k)$ have the same satisfiability. ■

In fact, the value of state variables in a satisfying assignment of $BMC'(M,p,k)$ also satisfy $BMC(M,p,k)$ and therefore can be used as a counterexample of the property p. The reason is that the value of the variables in a satisfying assignment of $BMC'(M,p,k)$ will also satisfy all CNF formulae CNF^A_I, $CNF^S_I(j)$, $CNF^A_R(i)$, $CNF^S_R(i,j)$ and $CNF^p(k)$. Thus, the value of the state variables will satisfy corresponding propositional formulae $I(s^A_0)$, $I(s^j_0)$, $R(s^A_i, s^A_{i+1})$, $R(s^j_i, s^j_{i+1})$ and $\bigvee_{i=0}^{k} \neg p(s_i)$. Hence, they together will satisfy $BMC(M,p,k)$, which is a conjunction of above propositional formulae. Therefore, the correctness of our algorithm is justified.

B. Implementation Details

Our test generation algorithm for multicore architectures is built around NuSMV model checker [19] and zChaff SAT solver [18]. We first model the system using SMV language, then use NuSMV to generate the CNF formulae CNF^A_I, $CNF^S_I(1)$, $CNF^A_R(i)$, $CNF^S_R(i,1)$ and $CNF^p(k)$ in DIMACS format as the input of Algorithm 1. zChaff is employed as the internal SAT solver. In this section, we briefly explain CNF generation process and the implementation of Step 1 and Step 2 in Algorithm 1.

The generation of CNF descriptions for a single core, bus and memory subsystem using NuSMV is straight forward. The only practical consideration is that all variables are represented by their indices in CNF clauses. As a result, it is important to avoid the same index to be used by two different variables. Since NuSMV does not offer any external interface to control the index assignment, we modified the source code to make the index space suitable for our purpose. The basic idea is to make the assignment of indices satisfy the following two constraints: 1) the indices of variables from the same core at the same time step are assigned continuously; 2) the indices of variables of the same time step across cores are assigned continuously as well. For example, in a 2-core system with each core having 100 variables, in time step 1 for core 1 we can use indices from 1-100 (controlled by the first constraint) whereas the second constraint indicates that the variables for core 2 at time step 1 should be 101-200. Therefore, 201-300 can be used to represent variables of core 1 in time step 2, and

so on. Based on these two constraints, the computation of the indices of symmetric variables can be efficiently implemented as increasing or decreasing by a certain offset.

During SAT solving, we also need to track the dependency of generated conflict clauses to determine whether they can be forwarded to other cores. This can be easily implemented within zChaff, which provides clause management scheme to support incremental SAT solving. For each clause in its clause database DB, zChaff uses a 32-bit group ID to track the dependency. Each bit identifies whether that clause belongs to a certain group. When a conflict clause is deduced based on clauses from multiple groups, its group ID is a "OR" product of the group ID of all its parent clauses, i.e., this clause belongs to multiple groups. zChaff also allows user to add or remove clauses by group ID between successive solving processes. If one clause belongs to multiple groups, it is removed when any of these groups are removed.

With these mechanisms, the step 1 and 2 in Algorithm 1 can be implemented efficiently as follows:

1) Add clauses in $CNF_I^S(j)$ and $CNF_R^S(i,j)$ with group ID j, $1 \leq j \leq N_S$
2) Add clauses in CNF_I^A, $CNF_R^A(i)$with group ID N_S+1.
3) Add clauses in $CNF^p(k)$ with group ID N_S+2.
4) When a new conflict clause is obtained during SAT solving, if it only belongs to a single group with ID smaller than N_S+1, replicate this clause to all other cores with proper group ID.
5) After solving all clauses in DB with zChaff, remove clauses with group ID N_S+2.

The overhead introduced by dependency identification and tracking in our algorithm is negligible compared to the improvement in SAT solving time. At the same time, since the indices of variables in symmetric cores are carefully assigned, the mapping table T is not maintained explicitly, but implemented as a simple mapping function, which is used to generate forwarding clauses for different cores. In that way, we avoid the potential caching overhead which may deteriorate the performance of the SAT solver.

C. Heterogeneous Multicore Architecture

So far, we discussed our algorithm using homogeneous cores. This section describes the application of our approach in the presence of heterogeneous cores. In a heterogeneous multicore system, if any two cores are completely different, it is not possible to reduce the test generation time by exploiting the symmetry. However, most real systems usually employ a cluster of identical cores for same computational purpose. In this case, we can first group them into symmetric components based on their types, then apply our algorithm to each symmetric component. For example, in the 5-core system shown in Figure 7, core 5 is used for monitoring and core 1-4 are identical cores for computation. We can define core 1-4 as the symmetric component and apply our algorithm on them. In general, we can apply our algorithm on each cluster of identical cores in a system.

However, when the heterogeneous cores are not completely different, i.e., only some functional units in them are different,

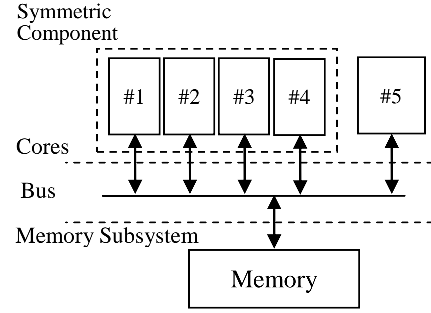

Fig. 7. Multicore system with different types of cores

our proposed algorithm can be employed in a more efficient way. Recall that the FSMs in a symmetric component are not restricted to cores. We can actually define the symmetric component in such a way that it includes only the identical functional units in different cores. For example, Figure 8 shows a system with heterogeneous cores. Both of the cores are pipelined with five stages: fetch, decode, execute, memory access, and writeback. The only difference is that they have different implementation in the execute stage EX. In this case, we define our symmetric component F^Sas the set of all functional units in two cores except EX. These two execution stages as well as bus and memory subsystem are modeled in the asymmetric part F^A. Of course, the input and output of F^S here will include not only the input and output variable of the cores, but also all the interface variables between EX and other stages. In this way, the information learned on all other stages of one core can still be shared by the other core. Clearly, the correctness of our approach is still guaranteed, because the selection of the symmetric component satisfies its definition.

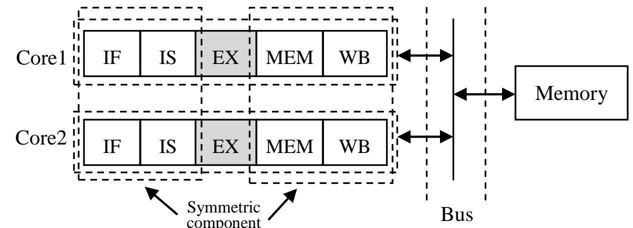

Fig. 8. Multicore system with different types of execution units

V. Experiments

We have evaluated the applicability and usefulness of our test generation technique on different multicore architectures.

A. Experimental Setup

As described in Section IV-B, the designs and properties are described in SMV language and converted to required CNF formulae (DIMACS files) using modified NuSMV [19]. We used zChaff [18] as our SAT solver to implement our test generation algorithm. Experiments were performed on a PC with 3.0GHz AMD64 CPU and 4GB RAM.

First, we present results of our approach using a multicore design that is composed of different number of identical cores, one bus, and memory subsystem. The pipeline inside each

core has five stages: fetch, decode, execute, memory access, and writeback. Besides, each core has its own cache, which is connected with the memory through the bus. Next, we will present (in Figure 11) the applicability of our approach on heterogeneous multicore architectures.

In order to activate the desired system behaviors, we used different number of properties on designs with different complexity. For instance, we used 375 properties in case of 16 core design that trigger two simultaneous activities between cores. We have also used several properties that involves multicore interactions. For example, one test will activate the following scenario: "if the value in a memory location which is initialized as one by core 1, is increased by one by all other cores, it should be equal to the number of cores when it is readback by core 2". It should be noted that the corresponding property is not symmetric with respect to all cores.

B. Results

We compared our approach with Strichman's approach [3] and original BMC [2]. Each approach was used to solve a sequence of SAT instances for the same property with varying bounds until a satisfiable instance is found. The input SAT instances for Strichman's approach and the original BMC was directly synthesized from $BMC(M, p, k)$ to improve their performance. When our approach was applied, we performed the SAT solving on $BMC'(M, p, k)$ as indicated in Section IV. We also tried to compare with [13]. Unfortunately, the implementation [20] failed to produce the symmetry breaking predicates due to the large size of our input CNF (more than 600k clauses).

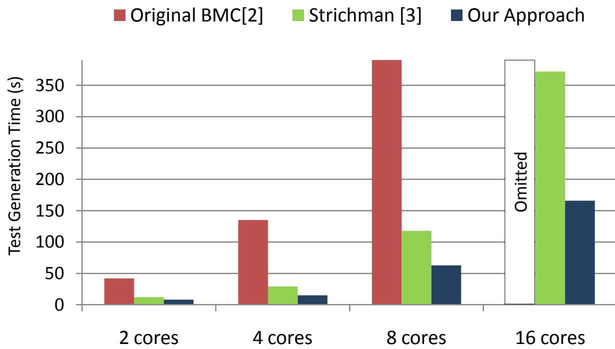

Fig. 9. Test generation time with different number of cores

Figure 9 presents the average test generation time for different number of cores. The original BMC failed to produce results within 3000 seconds on several properties for the 16 core system. Therefore, its time is omitted. As expected, the time consumption increases with the number of cores. Both our approach and Strichman's approach [3] are remarkably faster than original BMC [2]. By effective utilization of both spatial and temporal symmetry, our approach outperforms [3] (which only considers temporal symmetry) by nearly 2 times.

Table I shows a more detailed comparison of different approaches on the 8 core system for 10 most time consuming properties. The first column represents the names of properties used. The second column shows the corresponding bounds or time steps to activate each property. The next three columns

TABLE I. Test generation time for 8 core system

Prop.	Bound	[2] Time(s)	[3] Time(s)	Our Approach	Speedup over [2]	Speedup over [3]
1	28	79	56	25	3.16	2.24
2	22	67	44	21	3.19	2.10
3	32	93	62	30	3.10	2.07
4	28	208	94	17	12.24	5.53
5	33	*	342	148	-	2.31
6	20	413	124	47	8.79	2.64
7	20	*	125	48	-	2.60
8	**23**	**883**	**140**	**63**	**14.02**	**2.22**
9	25	2106	157	128	16.45	1.23
10	25	1991	106	101	19.71	1.05
Total	-	5840	1250	628	9.30	1.99

* represent run times exceeding 3000 sec.

present the test generation time (in seconds) for each property using the original BMC [2], Strichman's approach [3], and our approach, respectively. The time is calculated as the summation of the time to solve all the SAT instances from $k = 0$ to the bound of the property. The time calculation also includes the time consumed by non-SAT-solving steps in Algorithm 1. The last two columns indicate the speedup of our approach over [2] and [3]. It can be seem that our approach outperforms [3] by two times and [2] by an order of magnitude.

To inspect the reason of our improvement over [3], we analyze the behavior of the SAT solver. Table II shows details of the last five SAT instances immediately before the bound was found during the BMC of property 8 on the 8-core system (highlighted entry in Table I). The first column in Table II is the time step of each SAT instance. The next four columns contain the real size of the clause database before the solving process, the number of decisions made by zChaff, the number of forwarded conflict clauses and the time consumption in [3]. Similar information of our approach is represented in the last four columns. Compared to [3], the total number of decisions made by the SAT solver is much smaller when our approach is applied. At the same time, the number of forwarded clauses are comparable. In other words, our approach saves the time to rediscover the same knowledge for each core, without the overhead of forwarding too many conflict clauses.

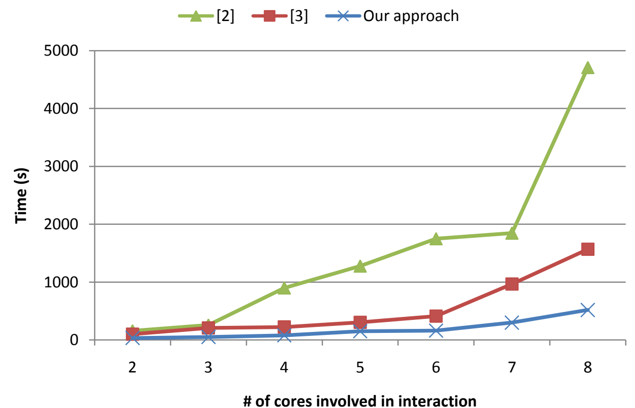

Fig. 10. Test generation time with different interactions

We also investigated the impact of different number of cores involved in the interaction on the test generation time. In this experiment, we use a processor with eight 3-stage cores. They are connected to the memory subsystem using snoopy protocol.

TABLE II. **Detailed test generation information**

k	[3]				Our approach			
	#Cls in DB	#Decision	#Fwd Cls	Time(s)	#Cls in DB	#Decision	#Fwd Cls	Time(s)
19	721427	40045	25608	2.4	756149	21231	4441	1.2
20	762855	71854	27329	3.6	857103	30049	26685	2.7
21	827272	56692	22824	3.4	900428	35687	24534	3.1
22	893382	203112	102202	15.4	965925	30873	6834	1.9
23	954998	2652411	142585	97.3	1029266	1228603	261989	52.8
Total	-	3024114	320548	122.1	-	1346443	324483	61.7

The desired test should trigger all cores perform read and write operation on the same shared memory variable in certain order. The results are given in Figure 10. When the interaction involves only a small number of cores, the difference in test generation time of [2], [3], and our approach is quite small. However, when more and more cores are involved, our approach outperforms both [2] and [3] remarkably, due to the usage of symmetry information.

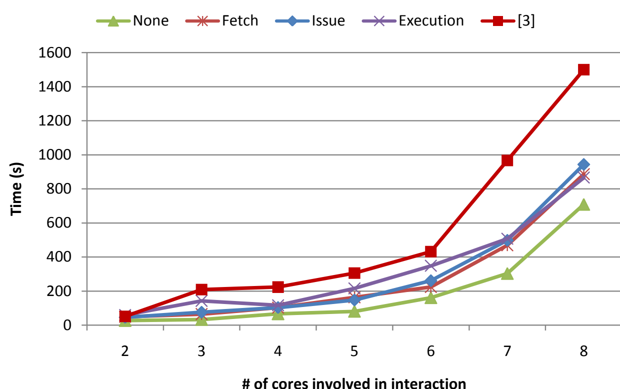

Fig. 11. Test generation time with heterogeneous cores

Finally, to illustrate the effectiveness of our approach in a more general scenario, we measure the test generation time on a system with heterogeneous cores. We use cores with different implementations in their fetch, issue, execution stages, and repeat the previous test generation experiment. As discussed in Section IV-C, we only replicate learned conflict clauses within the symmetric components. Figure 11 shows the result. The "fetch" curve corresponds to a system where the 8 cores are identical except their fetch stages. Similarly, curves marked as "Issue" and "Execution" represent cores with different issue and execution stages, respectively. We also show the test generation time for homogeneous cores using our approach ("None") and [3] as reference. It can be observed that due to less scope of knowledge reuse, the time consumption of our approach for heterogeneous cores are generally larger than homogeneous cores. Nevertheless, our approach still outperforms [3] especially for complicated interactions involving many cores.

VI. Conclusions

Functional verification of multicore architectures is challenging due to the increased design complexity and reduced time-to-market. Directed tests are promising because it requires significantly less number of tests to achieve the same coverage requirement compared to random tests. Unfortunately, the automatic generation of directed tests is time consuming due to the limitation of current model checking tools. Existing incremental SAT approaches have only exploited the symmetry in BMC across different time steps. In this paper, we presented a novel approach for directed test generation of multicore architectures that exploits both spatial and temporal symmetry in SAT-based BMC. The CNF description of the design is synthesized using CNF for cores, bus and memory subsystem to preserve the mapping information between different cores. As a result, the symmetric high level structure is well preserved and the knowledge learned from a single core can be effectively shared by other cores during the SAT solving process. The experimental results using homogeneous as well as heterogeneous multicore architectures demonstrated that the test generation time using our approach is remarkably smaller (2-10 times) compared to existing methods.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. of TACAS*, 1999, pp. 193–207.
[2] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.
[3] O. Strichman, "Accelerating bounded model checking of safety properties," *Formal Methods in System Design*, vol. 24, no. 1, pp. 5–24, 2004.
[4] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *ACM SIGSOFT Software Engineering Notes*, vol. 24, 1999, pp. 146–162.
[5] P. Mishra and N. Dutt, "Graph-based functional test program generation for pipelined processors," in *Proc. of DATE*, 2004, pp. 182–187.
[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proc. of DAC*, 2001, pp. 530–535.
[7] J. N. Hooker, "Solving the incremental satisfiability problem," *Journal of Logic Programming*, vol. 15, no. 1-2, pp. 177–186, 1993.
[8] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Proc. of DAC*, 2001, pp. 542–545.
[9] P. Mishra and M. Chen, "Efficient techniques for directed test generation using incremental satisfiability," in *Proc. of VLSI Design*, 2009, pp. 65–70.
[10] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Proc. of ICCAD*, nov. 2004, pp. 50 – 57.
[11] J. Bhadra, E. Trofimova, and M. Abadir, "Validating power architecture technology-based mpsocs through executable specifications," *IEEE Transactions on VLSI Systems*, vol. 16, no. 4, pp. 388 –396, apr. 2008.
[12] F. A. Aloul, A. Ramani, I. L. Markov, and K. Sakallah, "Solving difficult SAT instances in the presence of symmetry," in *Proc. of DAC*, 2002, pp. 731–736.
[13] F. A. Aloul, I. L. Markov, and K. Sakallah, "Shatter: efficient symmetry-breaking for boolean satisfiability," in *Proc of DAC*, 2003, pp. 836–839.
[14] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, "Exploiting structure in symmetry detection for cnf," in *Proc. of DAC*, 2004, pp. 530–534.
[15] D. Tang, S. Malik, A. Gupta, and C. N. Ip, "Symmetry reduction in SAT-based model checking," in *CAV*, 2005, pp. 125–138.
[16] A. Miller, A. Donaldson, and M. Calder, "Symmetry in temporal logic model checking," *ACM Comput. Surv.*, vol. 38, no. 3, p. 8, 2006.
[17] A. Biere and C. Sinz, "Decomposing SAT problems into connected components," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 191–198, 2006.
[18] zChaff. [Online]. Available: http://www.princeton.edu/∼chaff/zchaff.html
[19] NuSMV. [Online]. Available: http://nusmv.irst.itc.it/
[20] Shatter. [Online]. Available: http://www.aloul.net/Tools/shatter/