A

Directed Test Generation for Validation of Multicore Architectures

Xiaoke Qin and Prabhat Mishra, University of Florida

Functional validation is widely acknowledged as a major challenge for multicore architectures. 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 temporal, structural, and spatial symmetry in multicore designs at the same time. Our proposed technique enables the reuse of the knowledge learned from one core to the remaining cores in multicore architectures (structural symmetry), from one bound to the next for a give property (temporal symmetry), as well as from one property to other properties (spatial symmetry). The experimental results demonstrate that our approach can significantly (3-10 times) reduce overall test generation time compared to existing approaches.

Categories and Subject Descriptors: B.7.2 [**INTEGRATED CIRCUITS**]: Design Aids——Verification

General Terms: Algorithms, Verification

Additional Key Words and Phrases: multicore architecture, bounded model checking, SAT solving, test generation

1. 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 challenges. As a result, conventional random test 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 (discussed in Section 3.1). 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) [Biere et al. 1999; Clarke et al.

This work was partially supported by NSF CAREER award 0746261.
Author's addresses: X. Qin and P. Mishra, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

2001] 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 (discussed in Section 3.2). 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 three 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 [Strichman 2001; 2004] to accelerate the SAT solving process. On the other hand, the structural similarity of multiple cores also introduces a second category of symmetry or *structural* symmetry. This symmetry appears among the CNF clauses for different cores at the same time step. Intuitively, we can also exploit structural 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. The third category of symmetry or *spatial* symmetry is the artifact of solving multiple related properties for the same multicore design. It creates the opportunity to share learning across increasing bounds, different cores as well as related properties at the same time.

In this paper, we address the directed test generation challenges 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. We also extend our proposed approach to reduce the test generation time for multiple properties. 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 extends this approach in the context of multiple properties. Section 6 discusses the implementation details of our approaches. Section 7 presents our experimental results. Finally, Section 8 concludes the paper.

2. RELATED WORK

Model checking techniques are promising for functional verification and test generation for complex systems [Gargantini and Heitmeyer 1999; Mishra and Dutt 2004; Koo and Mishra 2006; Qin and Mishra 2012]. 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. [Biere et al. 1999] 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 [Moskewicz et al. 2001]. 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 [Strichman 2004; Hooker 1993; Whittemore et al. 2001; Chen and Mishra 2010; Qin et al. 2010; Qin and Mishra 2011]. The basic idea is to exploit the regularity of the SAT instances between different bounds. For example, incremental SAT solvers [Hooker 1993; Whittemore et al. 2001] 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 [Strichman 2004] 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 clauses 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 [Kuehlmann 2004] 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. Since this technique is based on the AIG representation of logic designs, it is difficult to use for accelerating the solving process of CNF instances, which 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. [Bhadra et al. 2008] used executable specification to validate multiprocessor systems-on-chip designs. Chen et al. [Chen and Mishra 2010][Chen et al. 2010] 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. Similarly, the simultaneous SAT solver [Khasidashvili et al. 2005] enabled the learned clauses to be reused by properties. *These approaches did not take the advantage of structural symmetry in multicore architectures.*

When SAT instance contains symmetric structure, symmetry breaking predicate [Aloul et al. 2002; Aloul et al. 2003; Darga et al. 2004; Tang et al. 2005; Miller et al. 2006] 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. [Biere and Sinz 2006] 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.*

To the best of our knowledge, our work is the first attempt to simultaneously exploit temporal, structural, and spatial symmetry to improve test generation time involving multiple properties for multicore architectures.

3. BACKGROUND

This section briefly describes the basic concepts of directed test generation using model checking, bounded model checking (BMC), and acceleration techniques for BMC.

3.1. Directed Test Generation with Model Checking

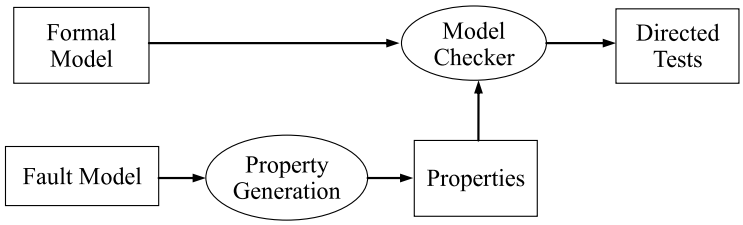


Fig. 1. Directed test generation using model checking

Figure 1 shows the general framework of the directed test generation based on model checking. In order to create directed tests, the formal model of the design specification and a suitable fault model (coverage criteria) are provided as inputs. A set of properties are then generated for the desired behaviors (faults) that should be activated in the simulation based validation stage. For example, when a graph model of the design and a functional coverage fault model are provided, a coverage-driven property generation can be used [Mishra and Dutt 2004]. After that, a model checker is employed to check whether there exists some states which violate the negated version of the property. It reports a counterexample, if it finds a violation. This counterexample contains a sequence of input information (e.g., instruction sequences for a processor design), which will drive the system from an initial state to a state, which does not satisfy the negated version of the property. In other words, the generated counterexample satisfies the original property. Therefore, we can use it as a test to activate the corresponding property or behavior during simulation-based validation.

3.2. Bounded Model Checking for Test Generation

Although model checking is effective for directed test generation, the capacity of the conventional symbolic model checking is usually limited. Bounded model checking (BMC) was proposed to address this problem by checking whether there is a counterexample for the property within a given bound [Clarke et al. 2001]. Given a design D, a safety property p, and a bound k, BMC will unroll the design k times and encode it using the following formula:

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

where $I(s_0)$ is the initial state of the system, $R(s_i, s_{i+1})$ represents the state transition from state s_i to state s_{i+1}, and $p(s_i)$ checks whether property p holds on state s_i. The formula is then transformed to CNF and checked by a SAT solver. If the SAT solver finds some assignment which makes the CNF true, it implies that the property does not hold at bound k, i.e., $M \nvDash_k p$. If no such assignment is found, we conclude that the property holds up to k, or $M \vDash_k p$. In directed test generation, the negated version of the property is checked by BMC. The SAT solver will find an assignment of all input and state variables, which satisfies Equation (1). As a result, we can extract the

assignment sequence of input variables and use it as a test to activate the desired property in the system.

Many techniques and heuristics are employed in SAT solvers to accelerate the solving process. Modern SAT solvers like zChaff [Princeton] and GRASP [Marques-Silva and Sakallah 1999] adopt the Davis-Putnam-Logemann-Loveland (DPLL) [Davis and Putnam 1960; Davis et al. 1962] algorithm and conflict-driven non-chronological backtracking. The basic idea behind these techniques is to save the knowledge learned during resolving current conflict to avoid the same conflict in the future [Zhang et al. 2001; Zhang et al. 2004]. A conflict occurs, when the current assignment of some variables, through a set of clauses, implies that one variable must be true and false at the same time. In this case, conflict analysis will trace back along the implication relations and find the closest assignment of variables that led to the conflict. We can forbid such assignment from occurring again by adding a carefully designed clause, i.e., *conflict clause*, to the original CNF. Generally, conflict clauses are only meaningful within the same SAT instance. However, when the set of clauses that led to the conflict clause are shared by multiple SAT instances, we can also forward conflict clauses across instances.

In practice, designers usually require multiple directed tests to validate different aspects of the same design. In this case, the total test generation time can be reduced by property clustering [Chen and Mishra 2010]. Since the knowledge learned during the solving process of a single property can be shared among similar properties, we can reduce total time consumption by clustering properties into different groups and solving all the properties in the same group together. In order to create directed tests, the formal model of the design, a set of properties for the desired behaviors (faults) are accepted as inputs. Next, the SAT instances are grouped into different clusters based on their similarity and then solved simultaneously to create the test suite, which can be used to trigger the desired behaviors during simulation-based validation.

4. TEST GENERATION FOR MULTICORE ARCHITECTURES

Our technique for directed test generation is motivated by previous works on incremental SAT-based BMC [Strichman 2004]. 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 for multicore designs also exhibit remarkable structural 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 [Strichman 2004]. 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 structural symmetry is not exploited.

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 structural 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 es-

[1]For ease of illustration, we first discuss our approach using a two-core architecture with homogeneous cores. However, our technique is applicable to general multicore architectures. The application of our approach on heterogeneous cores will be presented in Section 4.2.

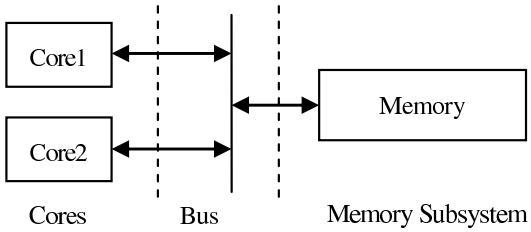


Fig. 2. Abstracted architecture of a two core system

tablished directly. Although the structural symmetry can be partially recovered by solving a graph automorphism problem [Aloul et al. 2002; Aloul et al. 2003; Darga et al. 2004], 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 structural symmetry.

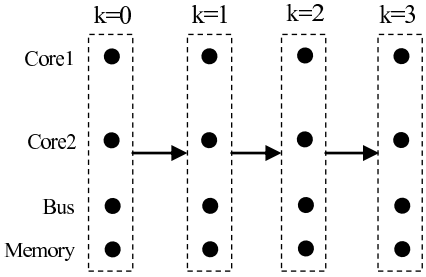


Fig. 3. Incremental SAT solving technique [Strichman 2004]

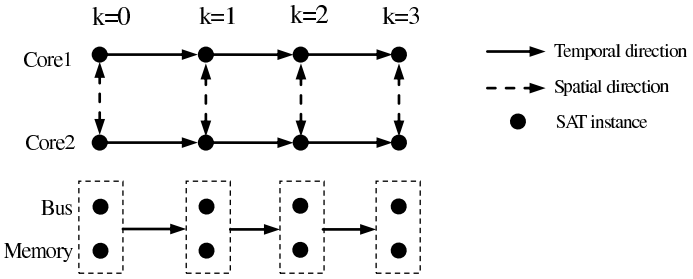


Fig. 4. Test generation for multicore architectures

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 structural 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 j^{th} FSM within a SC, we denote its initial condition and transitional constraints as $I(s^{is}_{0,j})$ and $R(s^{is}_{i,j}, s^{in}_{i,j}, s^{is}_{i+1,j}, s^{out}_{i+1,j})$ $(0 \leq i \leq k-1)$, where $s^{in}_{i,j}, s^{out}_{i+1,j}, s^{is}_{i,j}$ are its input variables, output variables, and internal state variables at the i^{th} $(i+1^{th})$ 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.*

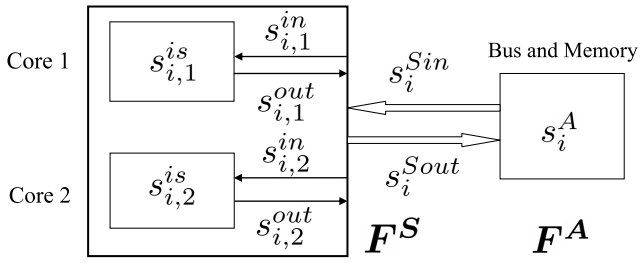


Fig. 5. FSM representation of Figure 2 at time step i

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 $I(s^A_0)$ and $R(s^A_i, s^{Sout}_i, s^A_{i+1}, s^{Sin}_{i+1})$ $(0 \leq i \leq k-1)$, where s^A_i represent internal state variables in bus and memory subsystem at the i^{th} time step. Moreover, $s^{Sin}_i = \{s^{in}_{i,j} | 1 \leq j \leq N_S\}$ and $s^{Sout}_i = \{s^{out}_{i,j} | 1 \leq j \leq N_S\}$ 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 i^{th} time step, the internal state variable of F^S are $\{s^{is}_{i,1}, s^{is}_{i,2}\}$ and s^A_i. The input and output variables of F^S (also the output and input variable of F^A) are $s^{Sin}_i = \{s^{in}_{i,1}, s^{in}_{i,2}\}$ and $s^{Sout}_i = \{s^{out}_{i,1}, s^{out}_{i,1}\}$, respectively.

The BMC formula of the multicore system can be expressed as

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

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

ALGORITHM 1: Test Generation for Multicore Architectures

Input: i) CNF formulae CNF_I^A, $CNF_I^S(1)$, $CNF_R^A(i)$, $CNF_R^S(i,1)$, $CNF^p(k)$
ii) Number of cores N_S
iii) Maximum bound K_{max}
Output: Test $test_p$
Bound $k \longleftarrow 0$
Initialize variable mapping table T
Common Clause Set $CCS \longleftarrow \emptyset$
if *All cores have the same initial state* **then**
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 $1 \leq j \leq N_S$
end
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

The basic idea of our approach is to generate CNF formula

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

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.

It should be noticed that we use symmetric term $CNF_I^S(j)$ to model the initial state constraints of each homogeneous core, because the processor cores are usually configured to the reset state before testing. When the cores are homogeneous, different cores usually have the same reset (initial) state. Thus, the initial state constraints are also symmetric. *However, the initial states in different cores can be different, when the reset state of each core are different by design.* In this case, the initial states are no longer symmetric and should be presented in CNF_I^A part. The corresponding CNF formula therefore becomes

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

where CNF_I^A is the conjunction of the CNF representations of $I(s_0^A)$ and $I(s_{0,j}^{is})$.

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 $CNF_I^S(1)$ [2] 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 [3] 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 two 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 [Strichman 2004].
(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.

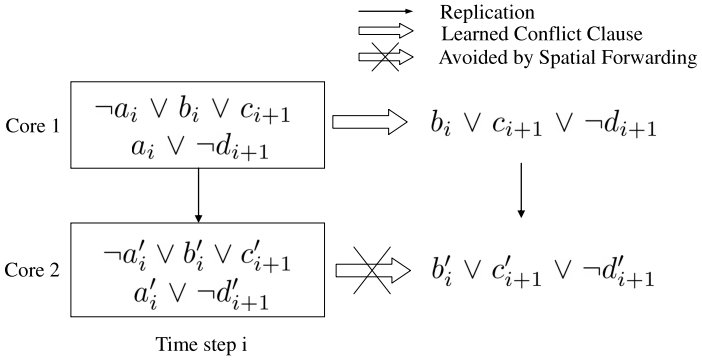


Fig. 6. Test generation for multicore architectures

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 $CNF_R^S(i,1)$ (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 $CNF_R^S(i,2)$ (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

[2] Only when all cores have the same initial state
[3] As discussed in Section 6, a physical table is not required, instead a mapping function is used in our framework.

forwarding of conflict clauses is not possible using Strichman's approach [Strichman 2004], which only considers temporal symmetry but not structural symmetry.

In the remainder of this section, we prove the correctness of our approach and discuss application of our approach in the context of heterogeneous multicore architectures.

4.1. 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

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

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

By their definitions, CNF formulae CNF_I^A, $CNF_I^S(j)$, $CNF_R^A(i)$, $CNF_R^S(i,j)$ and $CNF^p(k)$ are CNF representation of propositional formulae $I(s_0^A)$, $I(s_{0,j}^{is})$, $R(s_i^A, s_i^{Sout}, s_{i+1}^A, s_{i+1}^{Sin})$, $R(s_{i,j}^{is}, s_{i,j}^{in}, s_{i+1,j}^{is}, s_{i+1,j}^{out})$ and $\bigvee_{i=0}^{k} \neg p(s_i)$, where $0 \leq i \leq k-1$ and $1 \leq j \leq N_S$.

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

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

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_I^A, $CNF_I^S(j)$, $CNF_R^A(i)$, $CNF_R^S(i,j)$ and $CNF^p(k)$. Thus, the value of the state variables will satisfy corresponding propositional formulae $I(s_0^A)$, $I(s_0^j)$, $R(s_i^A, s_{i+1}^A)$, $R(s_i^j, s_{i+1}^j)$ 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.

4.2. Test Generation for 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 all cores are completely different, there is no structural symmetry. Therefore, it is not possible to reduce the test generation time. 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.

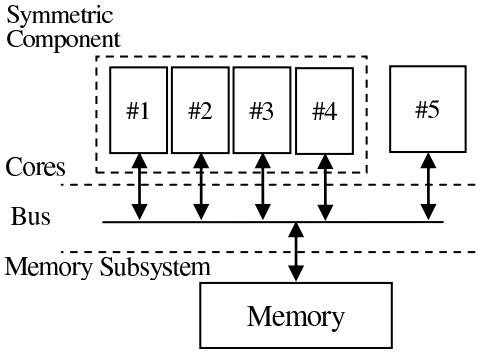


Fig. 7. Multicore system with different types of cores

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

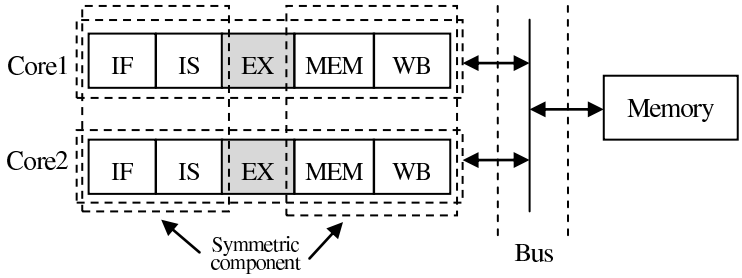
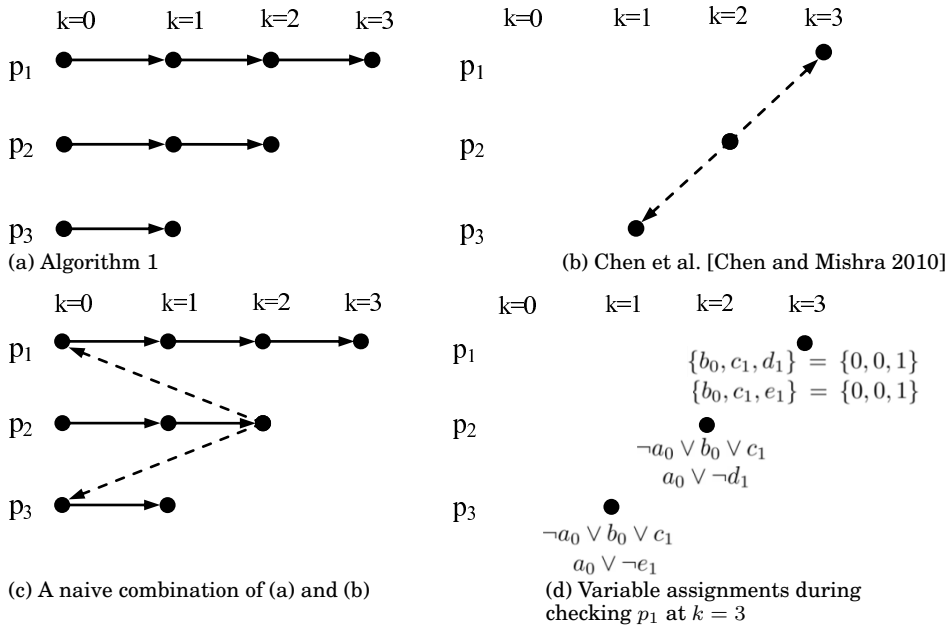


Fig. 8. Multicore system with different types of execution units

5. TEST GENERATION INVOLVING MULTIPLE PROPERTIES

Section 4 describes how to improve test generation time for a given property in a multicore architecture, but in reality, the same multicore design is validated using multiple different properties. Existing research [Chen and Mishra 2010] suggests that the overall test generation time can be effectively reduced by exploiting the similarity among similar properties, i.e., spatial symmetry. If we can group similar properties into clusters, and reuse the knowledge we learned during solving different properties within each cluster, we may be able to avoid unnecessary repetition of solving effort during the test generation of similar properties.



(a) Algorithm 1

(b) Chen et al. [Chen and Mishra 2010]

(c) A naive combination of (a) and (b)

(d) Variable assignments during checking p_1 at $k = 3$

Fig. 9. Different incremental SAT solving techniques

Figure 9 illustrates the clause forwarding paths in two different techniques: i) direct application of Algorithm 1 on each property and ii) test generation for multiple properties with known bounds [Chen and Mishra 2010]. In this example, there are three properties p_1, p_2, and p_3 with bounds 3, 2, and 1 respectively. We use solid dots to represent different SAT instances (conjunction of the mutlicore design and property CNF clauses) and lines to indicate the conflict clause forwarding paths. Algorithm 1 solved each property separately, and passed the knowledge (deduced conflict clauses) "horizontally" within instances for the same property (Figure 9a). Although the knowledge is shared among different cores within same property, there is no knowledge reused across different properties. In contrast,Chen et al. [Chen and Mishra 2010] solved one "base" property first, (e.g., p_2 in this case), then forward the learned clause "vertically" between other SAT instances for different properties, as shown in Figure 9b. As discussed in Section 2, their work did not consider multicore architectures, and was not designed to exploit structural symmetry..

Clearly, since multiple properties are checked on the same design, this spatial symmetry can be effectively exploited if we can appropriately forward conflict clauses "vertically" between properties while solving each property "horizontally". In this way, the knowledge learned during checking a property for a specific bound can benefit other

properties. One intuitive way to combine the two approaches, as shown in Figure 9c, is to choose some property as base property (p_2 in Figure 9c), check this property for different bounds, and then forward the learned conflict clauses to other SAT instances for other properties. Unfortunately, this naive combination has three problems. First, it is very hard to choose the base property, that can yield a set of beneficial conflict clauses which can be shared by other properties. Unlike [Chen and Mishra 2010], where each property has only one SAT instance for the known bound, in general, it is not possible to predict the total number of SAT instances to be solved. As a result, it is impossible to apply the clustering technique proposed in [Chen and Mishra 2010], to determine the base property. Secondly, even if we correctly find the optimal base property, it is still difficult to choose the suitable bound for the remaining properties (e.g., p_1 and p_3 in Figure 9c), to forward clauses, because SAT instances with inappropriate bounds may be solved trivially. Moreover, the learning during checking non-base properties is wasted. For example, in Figure 9d, suppose $(\neg a_i \vee b_i \vee c_{i+1})$, $(a_i \vee \neg d_{i+1})$ and $(a_i \vee \neg e_{i+1})$ are clauses within the transition constraint of the system at time step $i+1$. In the SAT solving process of p_2 with bound $k=2$, a conflict clause $(b_0 \vee c_1 \vee \neg d_1)$ is deduced based on $(\neg a_0 \vee b_0 \vee c_1)$ and $(a_0 \vee \neg d_1)$ to prevent the assignment $\{b_0, c_1, d_1\} = \{0, 0, 1\}$, which will result in a conflict on a_0. During the solving process of p_1 with bound $k=2$, the SAT solver may explore the assignment $\{b_0, c_1, d_1\} = \{0, 0, 1\}$ if Strichman's approach [Strichman 2004] is employed. Such assignment can be avoided by using [Chen and Mishra 2010] (as shown in Figure 9b and Figure 9c), because the learned conflict clause $(b_0 \vee c_1 \vee \neg d_1)$ is forwarded to p_1.

However, learned clauses are only allowed to be forwarded from the base property (p_2 in this case). The knowledge learned during solving non-base properties will not be reused. As indicated in Figure 9d, conflict clause $(b_0 \vee c_1 \vee \neg e_1)$ is deduced based on $(\neg a_0 \vee b_0 \vee c_1)$ and $(a_0 \vee \neg e_1)$ during the solving process of p_3 with bound $k=1$. Since p_3 is not a base property, this information will not be reused by p_1. Therefore, during the solving process of p_1 with bound $k=2$, the SAT solver will still try to make the assignment $\{b_0, c_1, e_1\} = \{0, 0, 1\}$. When the number of properties is large, this may cause a great waste of computational power, because we have to explore the same search space for many times, if the space is not visited during the solving process of the base property.

Our approach to solve this problem is based on the effective identification of conflict clauses that can be shared by other SAT instances across properties and bounds. *In fact, for any bound $k_0 \geq 0$, all SAT instances generated during BMC (Equation (1)) with $k \geq k_0$ clearly share the transition clauses $I(s_0) \wedge \bigwedge_{i=0}^{k_0-1} R(s_i, s_{i+1})$, although their property terms $\bigvee_{i=0}^{k} \neg p(s_i)$ are different. This observation implies that all conflict causes deduced based on these common clauses during solving process of any SAT instance can be forwarded to any other SAT instances with $k \geq k_0$, because all of them have the same set of clauses that led to the conflict clause.* Therefore, if we check all properties together for $k = 0, 1, 2, ...$, all conflict clauses can be safely shared by all subsequent SAT instances. In this way, we are able to utilize temporal, structural, as well as spatial symmetry at the same time. As discussed in Section 6, since only one copy of the transition relation is maintained, solving several properties together causes minor increase in total number of original clauses.

Algorithm 2 outlines our test generation method for clustered properties. It can be viewed as a natural extension of Algorithm 1. The only differences is that Algorithm 2 accepts a cluster of properties as input and produces the entire test set. For each property p at each bound k, Algorithm 2 performs SAT solving using the same technique as Algorithm 1 (Step 1), and keeps only new conflict clauses that are deduced indepen-

ALGORITHM 2: Test Generation For Properties in a Cluster

Input: i) CNF formulae CNF_I^A, $CNF_I^S(1)$, $CNF_R^A(i)$, $CNF_R^S(i,1)$
ii) Number of cores N_S
iii) Properties P,
iv) Maximum bound K_{max}
Output: Test Set TS
Bound $k \longleftarrow 0$
Initialize variable mapping table T
Common Conflict Clause Set $CCS \longleftarrow \emptyset$
if *All cores have the same initial state* **then**
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 $1 \leq j \leq N_S$
end
$TS \longleftarrow \emptyset$
Update T Add Clauses in CNF_I^A to CCS
while $P \neq \emptyset$ *and* $k \leq K_{max}$ **do**
Clause Set $CS_T^k \longleftarrow BMC(D, true, k)$
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
for $p \in P$ **do**
Create $CNF^p(k)$
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**
remove p from P
$TS \longleftarrow TS \bigcup test_p$
end
end
$k \longleftarrow k+1$
end
return *TS*

dent of $CNF^p(k)$ in CCS. In this way, the knowledge reuse across different properties are achieved implicitly.

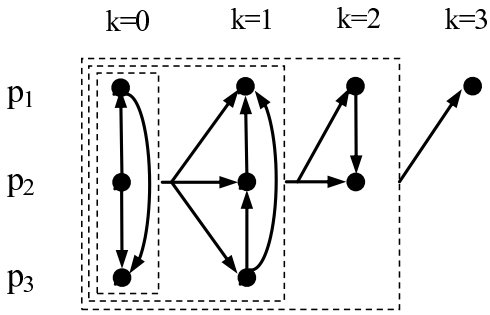


Fig. 10. Test generation for multiple properties

We use the same example in Figure 9 to illustrate the flow of Algorithm 2. The clause forwarding path are shown in Figure 10. In the first iteration for $k = 0$, suppose we

randomly pick p_2 from the property set. At the beginning, the common conflict clause set CCS is empty. Thus, p_2 is solved directly. Since the bound of p_2 is 2, the SAT instance is not satisfiable and no test is generated. However, all conflict clauses deduced based on clauses in CS_T^0 are now recorded in CCS, and will be used to accelerate the solving process of both p_1 and p_3 at bound 0. Similarly, the conflict clauses generated during solving p_1 at $k = 0$ will be used to speed up p_3 at $k = 0$ (assumes p_3 is solved last). In the next iteration, all instances will be solved with the help of conflict clauses learned by all three SAT instances at $k = 0$, because all conflict clauses are recorded in CCS. Eventually, three tests will be generated at bound 3, 2, and 1 for p_1, p_2 and p_3, respectively. In the case of Figure 9d, since both $(\neg a_0 \vee b_0 \vee c_1)$, $(a_0 \vee \neg d_1)$ and $(a_0 \vee \neg e_1)$ are clauses from the transition constraint of the system, both $(b_0 \vee c_1 \vee \neg d_1)$ and $(b_0 \vee c_1 \vee \neg e_1)$ will be recorded in CCS based on Algorithm 2. Therefore, during the solving process of p_1 with bound $k = 2$, the SAT solver will skip the assignment $\{b_0, c_1, d_1\} = \{0, 0, 1\}$ and $\{b_0, c_1, d_1\} = \{0, 0, 1\}$. In this way, the unnecessary waste of time is avoided. Reuse of learning accross a set of related properties enables Algorithm 2 to perform better than Algorithm 1, as demonstrated in Figure 11.

6. IMPLEMENTATION DETAILS

Our test generation algorithm for multicore architectures is built around NuSMV model checker [NuSMV] and zChaff SAT solver [Princeton]. We first model the system using SMV language, then use NuSMV to generate the CNF formulae CNF_I^A, $CNF_I^S(1)$, $CNF_R^A(i)$, $CNF_R^S(i, 1)$ 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 and Algorithm 2.

The generation of CNF descriptions for a single core, bus and memory subsystem using NuSMV is straightforward. 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 and Algorithm 2 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 algorithms 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.

7. EXPERIMENTS

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

7.1. Experimental Setup

As described in Section 6, the designs and properties are described in SMV language and converted to required CNF formulae (DIMACS files) using modified NuSMV [NuSMV]. We used zChaff [Princeton] 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, every core has its own cache, which is connected with the memory through the bus. Each core is described using 314 lines of SMV code. We also assume that all cores have the same reset (initial) state. Next, we will present (in Figure 13) 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. Our experiments also use very complex properties such as: “if the value in a memory location which is initialized as one by core1, is increased by one by all other cores, it should be equal to the number of cores when it is readback by core0”. It should be noted that the corresponding property is not symmetric with respect to all cores. Our property clustering approach for designs given in graph models is similar to [Chen and Mishra 2010]. The properties are grouped together by their similarity on structural or textual overlap.

7.2. Results

We compared our approach with Strichman’s approach [Strichman 2004] and original BMC [Clarke et al. 2001]. Algorithm 1 and Algorithm 2 represent our approaches described in Section 4 and Section 5, respectively. 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 described in Section 4. We also tried to compare with [Aloul et al. 2003]. Unfortunately, the implementation [ALOUL 2003] failed to produce the symmetry breaking predicates due to the large size of our input CNF (more than 600k clauses).

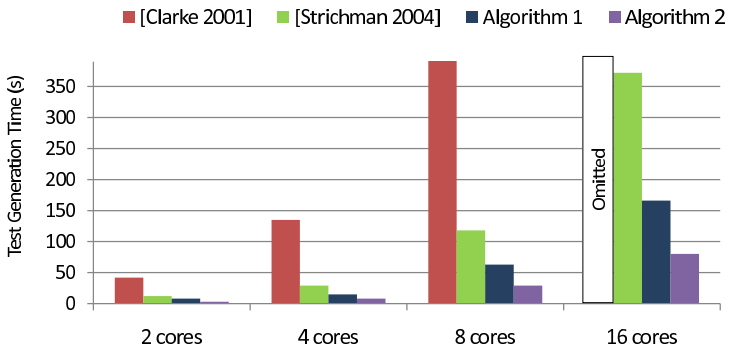


Fig. 11. Test generation time with different number of cores

Figure 11 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 [Strichman 2004] are remarkably faster than original BMC [Clarke et al. 2001]. By effective utilization of both structural and temporal symmetry, Algorithm 1 outperforms [Strichman 2004] (which only considers temporal symmetry) by nearly 2 times. Since multiple properties are checked on the same design, Algorithm 2 further reduces the average solving time by exploiting the spatial symmetry, which outperforms [Strichman 2004] by 3-4 times.

Table I. Test generation time for 8 core system

Prop.	Bound	BMC Time(s)	STR'04 Time(s)	Algorithm 1	Algorithm 2	Speedup over BMC	Speedup over STR'04
1	28	79	56	25	19	4.16	2.95
2	22	67	44	21	17	3.94	2.59
3	32	93	62	30	12	7.75	5.17
4	28	208	94	17	9	23.11	10.44
5	33	*	342	148	130	-	2.63
6	20	413	124	47	15	27.53	8.27
7	20	*	125	48	26	-	4.81
8	**23**	**883**	**140**	**63**	**22**	**40.14**	**6.36**
9	25	2106	157	128	105	20.06	1.50
10	25	1991	106	101	21	94.81	5.05
Total	-	5840	1250	628	376	15.53	3.32

* represent run times exceeding 3000 sec.

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 present the test generation time (in seconds) for each property using the original BMC [Clarke et al. 2001], Strichman's approach [Strichman 2004] (STR'04), and our approaches (Algorithm 1 and Algorithm 2), respectively. The first three techniques are applied to each property independently, while Algorithm 2 is applied to property clusters. 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 and Algorithm 2. The last two columns indicate the speedup of

Algorithm 2 over [Clarke et al. 2001] and [Strichman 2004]. It can be seen that Algorithm 1 outperforms both [Clarke et al. 2001] and [Strichman 2004] by 2 and 10 times, repectively. Due to the reuse of knowledge across different properties, Algorithm 2 outperforms Algorithm 1 by 40%. We also performed these experiments with asymmetric initial states constraints. The results suggest that our approaches still perform 2-10 times faster than existing approaches. This can be explained by the fact that the initial constraints only produces a quite small amount of clauses, which is much less than the clauses produced by the unrolled transition relations.

Table II. Detailed test generation information

k	[Strichman 2004]				Our approach (Algorithm 1)			
	#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

To inspect the reason of our improvement over [Strichman 2004], 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 [Strichman 2004]. Similar information of our approach is represented in the last four columns. Compared to [Strichman 2004], 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, Algorithm 1 saves the time to rediscover the same knowledge for each core, without the overhead of forwarding too many conflict clauses. Similar effects can also be observed during the application of Algorithm 2.

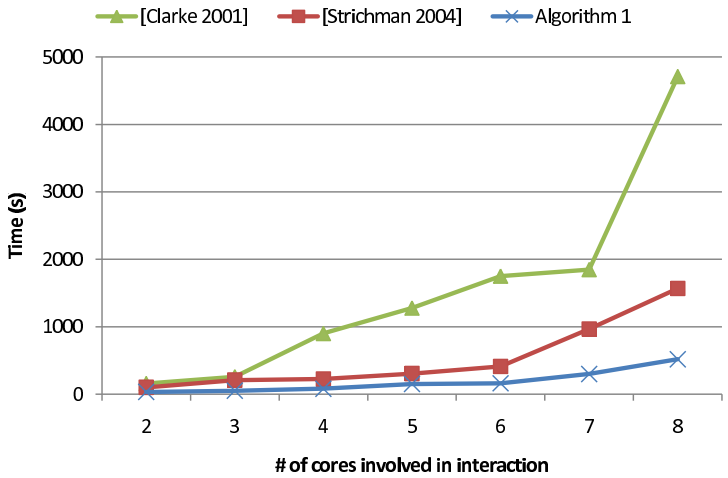


Fig. 12. Test generation time with different interactions

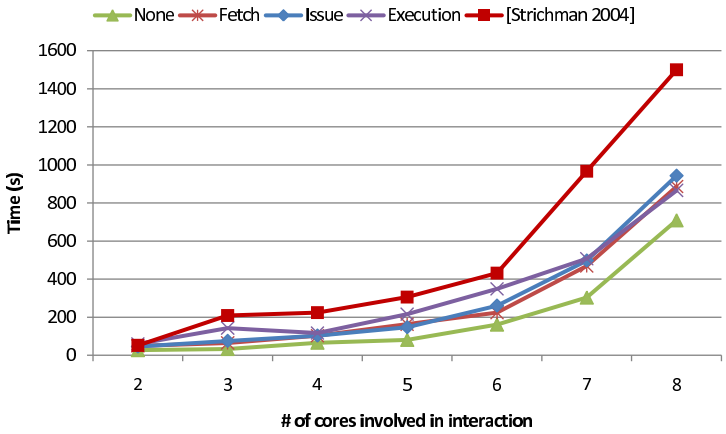


Fig. 13. Test generation time with heterogeneous cores

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. 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 12. When the interaction involves only a small number of cores, the difference in test generation time of [Clarke et al. 2001], [Strichman 2004], and our approach is quite small. However, when more and more cores are involved, our approach outperforms both [Clarke et al. 2001] and [Strichman 2004] remarkably, due to the usage of symmetry information.

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 experiments. As discussed in Section 4.2, we only replicate learned conflict clauses within the symmetric components. Figure 13 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 [Strichman 2004] 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 [Strichman 2004] especially for complicated interactions involving many cores.

8. 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 temporal symmetry in BMC across different time steps. In this paper, we presented a novel approach for directed test generation of multicore architectures that exploits temporal, structural, as well as spatial 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, i.e., structural

symmetry, is well preserved and the knowledge learned from a single core can be effectively shared by other cores during the SAT solving process. We also exploit the spatial symmetry of the same design by reusing the knowledge across different properties. The experimental results using homogeneous as well as heterogeneous multicore architectures demonstrated that the test generation time using our approach is remarkably smaller (3-10 times) compared to existing methods.

REFERENCES

ALOUL, F. A. 2003. Shatter. http://www.aloul.net/Tools/shatter/.

ALOUL, F. A., MARKOV, I. L., AND SAKALLAH, K. 2003. Shatter: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the Design Automation Conference*. ACM, New York, NY, USA, 836–839.

ALOUL, F. A., RAMANI, A., MARKOV, I. L., AND SAKALLAH, K. 2002. Solving difficult SAT instances in the presence of symmetry. In *Proceedings of the Design Automation Conference*. ACM, New York, NY, USA, 731–736.

BHADRA, J., TROFIMOVA, E., AND ABADIR, M. S. 2008. Validating power architecture technology-based mpsocs through executable specifications. *IEEE Trans. Very Large Scale Integr. Syst. 16*, 388–396.

BIERE, A., CIMATTI, A., CLARKE, E. M., AND ZHU, Y. 1999. Symbolic model checking without BDDs. In *Proceedings of TACAS*. Springer-Verlag, London, UK, 193–207.

BIERE, A. AND SINZ, C. 2006. Decomposing SAT problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation 2*, 191–198.

CHEN, M. AND MISHRA, P. 2010. Functional test generation using efficient property clustering and learning techniques. *IEEE Trans. on CAD of Integrated Circuits and Systems 29,* 3, 396–404.

CHEN, M., QIN, X., AND MISHRA, P. 2010. Efficient Decision Ordering Techniques for SAT-based Test Generation. In *Proceedings of Design, Automation & Test in Europe*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 490–495.

CLARKE, E., BIERE, A., RAIMI, R., AND ZHU, Y. 2001. Bounded model checking using satisfiability solving. *Formal Methods in System Design 19,* 1, 7–34.

DARGA, P. T., LIFFITON, M. H., SAKALLAH, K. A., AND MARKOV, I. L. 2004. Exploiting structure in symmetry detection for cnf. In *Proceedings of the Design Automation Conference*. ACM, New York, NY, USA, 530–534.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem-proving. *Communication of ACM 5,* 7, 394–397.

DAVIS, M. AND PUTNAM, H. 1960. A computing procedure for quantification theory. *Journal of ACM 7,* 3, 201–215.

GARGANTINI, A. AND HEITMEYER, C. 1999. Using model checking to generate tests from requirements specifications. In *ACM SIGSOFT Software Engineering Notes*. Vol. 24. ACM, New York, NY, USA, 146–162.

HOOKER, J. N. 1993. Solving the incremental satisfiability problem. *Journal of Logic Programming 15,* 1-2, 177–186.

KHASIDASHVILI, Z., NADEL, A., PALTI, A., AND HANNA., Z. 2005. Simultaneous SAT-based model checking of safety properties. In *Proceedings of Haifa Verification Conference*. 56–75.

KOO, H.-M. AND MISHRA, P. 2006. Functional test generation using property decompositions for validation of pipelined processors. In *Proceedings of Design, Automation & Test in Europe*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 1240–1245.

KUEHLMANN, A. 2004. Dynamic transition relation simplification for bounded property checking. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*. IEEE Computer Society, Washington, DC, USA, 50–57.

MARQUES-SILVA, J. P. AND SAKALLAH, K. A. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers 48*, 506–521.

MILLER, A., DONALDSON, A., AND CALDER, M. 2006. Symmetry in temporal logic model checking. *ACM Comput. Surv. 38,* 3, 8.

MISHRA, P. AND DUTT, N. 2004. Graph-based functional test program generation for pipelined processors. In *Proceedings of Design, Automation & Test in Europe*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 182–187.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*. ACM, New York, NY, USA, 530–535.

ITC-IRST AND CMU. NuSMV. http://nusmv.irst.itc.it/.

PRINCETON. zChaff. http://www.princeton.edu/∼chaff/zchaff.html.

QIN, X., CHEN, M., AND MISHRA, P. 2010. Synchronized generation of directed tests using satisfiability solving. In *Proceedings of the International Conference on VLSI Design*. IEEE Press, Piscataway, NJ, USA, 351–356.

QIN, X. AND MISHRA, P. 2011. Efficient directed test generation for validation of multicore architectures. In *Proceedings of International Symposium on Quality Electronic Design*. IEEE Press, Piscataway, NJ, USA.

QIN, X. AND MISHRA, P. 2012. Efficient Decision Ordering Techniques for SAT-based Test Generation. In *Proceedings of Design, Automation & Test in Europe*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium.

STRICHMAN, O. 2001. Pruning techniques for the SAT-based bounded model checking problem. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*. 58–70.

STRICHMAN, O. 2004. Accelerating bounded model checking of safety properties. *Formal Methods in System Design 24,* 1, 5–24.

TANG, D., MALIK, S., GUPTA, A., AND IP, C. 2005. Symmetry reduction in sat-based model checking. In *Computer Aided Verification*, K. Etessami and S. Rajamani, Eds. Lecture Notes in Computer Science Series, vol. 3576. Springer Berlin / Heidelberg, 283–284.

WHITTEMORE, J., KIM, J., AND SAKALLAH, K. 2001. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference*. ACM, New York, NY, USA, 542–545.

ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. H., AND MALIK, S. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International conference on Computer-aided design*. IEEE Press, Piscataway, NJ, USA, 279–285.

ZHANG, L., PRASAD, M. R., AND HSIAO, M. S. 2004. Incremental deductive & inductive reasoning for SAT-based bounded model checking. In *Proceedings of the International conference on Computer-aided design*. IEEE Press, Piscataway, NJ, USA, 502–509.