Test Generation using SAT-based Bounded Model Checking for Validation of Pipelined Processors

Heon-Mo Koo, Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA
hkoo@cise.ufl.edu, prabhat@cise.ufl.edu

ABSTRACT

Functional verification is one of the major bottlenecks in microprocessor design. Simulation-based techniques are the most widely used form of processor verification. Efficient test generation is crucial for the simulation-based verification. We present an efficient test generation methodology using SAT-based bounded model checking (BMC). This paper addresses two important challenges in test generation using SAT-based BMC: determination of *bound* for each property, and application of design and property decompositions to improve test generation time as well as memory requirement. Our experimental results using a MIPS processor demonstrate the feasibility and usefulness of our approach.

Categories and Subject Descriptors: B.6.3 [Logic Design]: Design Aids - verification

General Terms: Verification

Keywords: Test Generation, Functional Validation

1. INTRODUCTION

Functional verification is a major bottleneck in processor design due to the combined effects of increasing design complexity and decreasing time-to-market. Simulation-based validation is the most widely used form of processor verification using test programs that consist of instruction sequences. There are three types of test generation techniques: random, directed, and directed-random. The directed tests can reduce overall validation effort since shorter tests can obtain the same coverage goal compared to the random tests. However, directed test generation is mostly performed by human intervention. Hand-written tests entail laborious and time consuming effort of verification engineers who have deep knowledge of the design under verification. Due to the manual development, it is infeasible to generate all directed tests to achieve comprehensive coverage goal. Automatic test generation is the alternative to address this problem.

Test generation using model checking is one of the most promising approaches due to its capability of automatic test generation. In this test generation scenario, processor model is described in a temporal specification language and a desired behavior is expressed in the form of temporal logic property. A model checker exhaustively searches all reachable states of the model to check if the property holds (*verification*) or not (*falsification*), which is called unbounded model checking (UMC). If the model checker finds any reachable state that does not satisfy the property, it produces a counterexample. The counterexample contains a sequence of instructions (test program) from an initial state to a state where the negated version of the property fails. However, this approach is unsuitable for large designs due to the state explosion problem in UMC.

SAT-based bounded model checking (BMC) restricts search space that is reachable from initial states within a fixed number (k) of transitions, called *bound*. After unwinding the model of design k times, the BMC problem is converted into a propositional satisfiability (SAT) problem. SAT solver is used to find a satisfiable assignment of variables that is converted into a counterexample. If the *bound* is known in advance, SAT-based BMC is typically more effective for falsification than UMC because search for counterexample is faster and SAT capacity reaches beyond BDD capacity [3]. However, finding *bound* is a challenging problem since the depth of counterexamples is unknown in general.

Choosing an incorrect bound increases test generation time and memory requirement. In the worst case, test generation may not be possible. For example, we can increase the bound iteratively starting from a small bound until a counterexample is found. This approach is advantageous for shallow counterexamples, but disadvantageous for deep counterexamples due to accumulation of iterative running time. Alternatively, a large bound can be used such that all counterexamples are found. This approach loses the benefits of BMC due to search in a large number of irrelevant states. Therefore, the performance of test generation closely depends on the schemes of deciding the bound. We propose a method for determining the bound for each property instead of using a maximum bound for all properties.

Figure 1 shows our test generation methodology. Processor model is generated from the architecture specification. We use pipeline interaction fault model to define functional coverage [12]. Temporal logic properties are created from pipeline interaction faults based on the specification. We have developed a procedure for determining a bound for each property. Processor model, negated property, and the

GLSVLSI'06, April 30–May 1, 2006, Philadelphia, PA, USA.
Copyright 2006 ACM 1-59593-347-6/06/0004 ...$5.00.

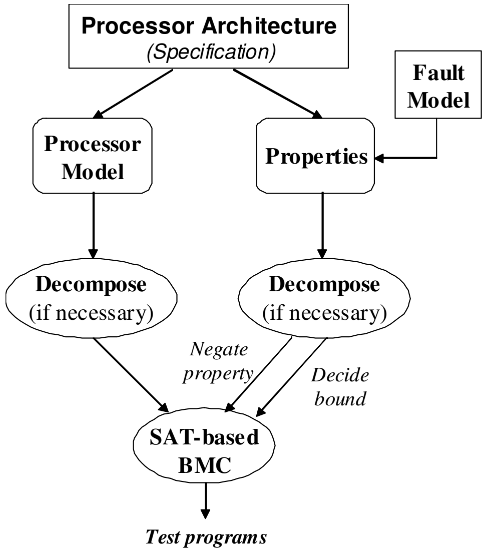

Figure 1: Test Program Generation Methodology

bound are applied to SAT-based BMC to generate a test program. Based on the coverage criteria, more properties can be added. We use design and property decompositions to further improve the performance of test generation.

The rest of the paper is organized as follows. Section 2 presents related work addressing test generation approaches. Section 3 describes a functional fault model for pipeline interactions. Section 4 presents our test generation methodology followed by a case study in Section 5. Finally, Section 6 concludes the paper.

2. RELATED WORK

A variety of techniques for generating test programs have been developed for architectural and micro-architectural validation of pipelined processors. In [11], processor model is described as a finite state machine (FSM) and reachable states and state transitions are used to generate test programs based on FSM coverage. To handle the large size of FSMs for modern processors, Shen and Abraham [15] have proposed an RTL abstraction technique that creates an abstract FSM model while preserving clock accurate behaviors. Ur and Yadin [16] have also used abstraction of processor model to generate test programs for micro architectural validation of a superscalar PowerPC processor. Adir et al. [1] have separated model of design from a test generation engine to avoid state explosion in formal methods and to facilitate test generation for modern processors.

Model checking [5] has been successfully used in software and hardware verification as the test generation engine [7, 12]. Model of design is applied to a model checker along with negated temporal logic properties to exploit falsification capability of model checking. However, traditional model checking does not scale well due to the state explosion problem. Biere et al. [4] introduced bounded model checking (BMC) combined with satisfiability solving. The recent developments in SAT-based BMC techniques have been presented in [14]. BMC is an incomplete method that cannot guarantee a true or false determination when a counterexample does not exist within a given bound. However, once the bound of a counterexample is known, large designs can be falsified very fast since SAT solvers [8, 13] do not require exponential space, and searching counterexample in an arbitrary order consumes much less memory than breadth first search in model checking. The performance of bounded and unbounded algorithms was analyzed on a set of industrial benchmarks in [2]. The capacity increase of BMC techniques has become attractive for industrial use. Intel study [6] showed that BMC has better capacity and productivity over unbounded model checking for real designs taken from the Pentium-4 processor. Recently, Gurumurthy et al. [9] have adopted BMC for mapping pre-computed module-level test sequences to processor instructions using structural fault model.

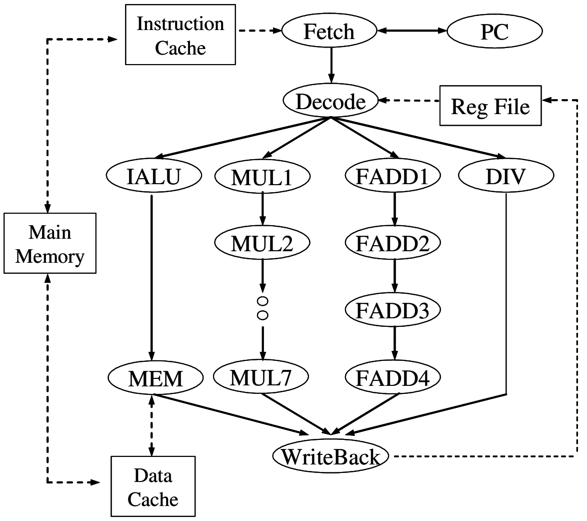

Figure 2: Graph Model of the MIPS processor

3. FUNCTIONAL FAULT MODEL

Today's test generation techniques and verification methods are very efficient to find bugs in a single module. Hard-to-find bugs arise often from the interactions among multiple components of a complex design. We primarily focus on the interactions among functional units in a pipelined processor. First, we briefly describe a graph-based modeling of pipelined processors. Next, we define a pipeline interaction fault model using the graph model.

The structure of a pipelined architecture is modeled as a graph $G = (V, E)$. V denotes two types of components in the processor: *units* (e.g., Fetch, Decode, etc) and *storages* (e.g., register file or memory). E consists of two types of edges: *pipeline edges* and *data transfer edges*. A pipeline edge transfers an instruction (operation) from a parent unit to a child unit. A data-transfer edge transfers data between units and storages. For illustration, we use a simplified version of the MIPS processor [10] as shown in Figure 2. In the figure, ovals denote units, rectangles are storages, bold edges are pipeline edges, and dashed edges are data-transfer edges. A path from a root node (Fetch) to a leaf node (WriteBack) consisting of units and pipeline edges is called a *pipeline path.* For example, {Fetch - Decode - IALU - MEM - WriteBack} is a pipeline path. A path from a unit to main memory or register file consisting of storages and data transfer edges is called a *data-transfer path.* For example, {MEM - DataCache - MainMemory} is a data-transfer path.

Using the graph model shown in Figure 2, interactions can be described as a combination of nodes and their activities.

We consider four functional activities in a node: *operation execution, stall, exception, and NOP (no-operation).* A unit in *operation execution* carries out its functional operations such as fetching an instruction, decoding opcode/operand, arithmetic/logic computation etc. Stall in a unit can be caused by various reasons such as data hazard, structural hazard, child node stall etc. Exception in a node is an exceptional state such as divide-by-zero or overflow. We consider two types of faults: node fault, and interaction fault. A node is faulty if it produces incorrect output during an activity. An interaction is faulty if execution of multiple activities of the interaction produces incorrect result. In the presence of a fault, unexpected values are written to the primary outputs such as memory or register file, or the test program finishes at incorrect clock cycle during simulation.

4. TEST GENERATION

Algorithm 1 describes our test generation procedure. This algorithm takes processor model M and interaction faults S as inputs and generates test programs. For each fault S_i, the algorithm produces one test program. Fault S_i is composed of a set of node activities and their relations. The algorithm iterates over all the interaction faults in the fault model. Each fault S_i is converted to a temporal logic property P_i. Section 4.1 describes the procedure for creating and negating the property. Next, bound k_i for each property is decided as discussed in Section 4.2. SAT-based BMC takes processor model M, negated property $\overline{P_i}$, and bound k_i as inputs and generates a counterexample (test program).

```
Algorithm 1: Test Generation
Inputs: i) Processor model $M$
        ii) Set of faults $S$ from interaction fault model
Outputs: Test programs to excite the pipeline interactions
Begin
  TestPrograms = $\phi$
  for each fault $S_i$ in the set $S$
    $P_i$ = CreateProperty($S_i$)
    $\overline{P_i}$ = Negate($P_i$)
    $k_i$ = DecideBound($\overline{P_i}$)
    $test_i$ = DoSATbasedBMC($M$, $\overline{P_i}$, $k_i$)
    TestPrograms = TestPrograms $\cup$ $test_i$
  endfor
  return TestPrograms
End
```

So far, we assumed that the whole design model is applied to SAT-based BMC. This approach is effective when design is of moderate size and the bound is shallow. However, for the test generation scenarios consisting of large designs and deep counterexamples, SAT-based BMC may not be able to generate tests. In such cases, decompositions of property as well as design will reduce the test generation complexity. Section 4.3 describes test generation techniques using decompositional bounded model checking. Finally, Section 4.4 presents a test generation example using Algorithm 1.

4.1 Property Generation

A node fault is converted into a property $F(p_i)$ where F is a temporal operator (*eventually*) and p_i is described as ($module_i.activity$). $F(p_i)$ is true if p_i becomes true at any time step. The atomic proposition p_i is a functional activity at a node i such as operation execution, stall, exception or NOP. The negation of the property $F(p_i)$, $G(\neg p_i)$, is true if p_i is never true over all time steps. G is a temporal operator, *always*. For example, to exercise a node fault *"Decode in stall"*, the fault is converted into *F(Decode.Stall)*. Its negation *G(¬Decode.stall)* means *"Decode never in stall"*.

A pipeline interaction fault is converted into a property $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ that combines activities over n modules using logical *AND* operator. The property is true if $(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ becomes true at any time step. The negation of the property, $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$, becomes true if any of p_1, p_2, ..., *or* p_n is not true over all time steps. For example, to exercise an interaction fault *"Decode in stall and FADD1 in operation execution at the same time"*, the fault is converted into *F(Decode.stall ∧ FADD1.exe)*. Its negation *G(¬Decode.stall ∨ ¬FADD1.exe)* means *"Decode in stall and FADD1 in operation execution never occur at the same time"*.

4.2 Determination of Bound

The longest computation path in the pipeline corresponds to the bound to generate tests for all interaction scenarios. For example, in the graph model of the MIPS processor in Figure 2, the maximum bound is determined by the length of {FE→ DE→ IALU→ MEM→ Cache→ MM → Cache→ MEM→ WB} if cache miss takes more time than any other pipeline paths. However, this bound is over-conservative in most test scenarios because a lot of interactions do not include this longest path. Therefore, using bound for each interaction is more efficient for test generation.

Bound for each node fault is decided by the temporal distance between the root node (e.g., Fetch) and the node under verification. For example, bound for the property *"FADD1 in operation execution"* will be 3 if there is only one pipeline register between pipeline stages. Bound for each interaction fault is determined by the longest temporal distance from the root node to the nodes under consideration. For example, bound for the property *"IALU, FADD2, and FADD3 in operation execution at the same time"* will be 5 because FADD3 has the longest temporal distance from Fetch stage.

4.3 Design and Property Decompositions

Design and property decompositions can be used to further improve test generation performance. We consider only two partitioning techniques: vertical (path-level) partitioning and horizontal (stage-level) partitioning. These methods are similar to the cone of influence used in model checking tools. For example, in the graph model shown in Figure 2, the integer-ALU pipeline path PP_{IALU} = {Fetch, Decode, IALU, Mem, WriteBack} is treated as one path level partition. Horizontal partitioning is determined by the distance from the root node (e.g., Fetch). For example, for a given property *"IALU, FADD2, and FADD3 in operation execution at the same time"*, we can use partitioned modules {Fetch, Decode, IALU, FADD1, FADD2, FADD3} to generate a test program instead of using whole design because the descendent nodes of IALU and FADD3 do not affect the counterexample generation of the property.

4.4 Test Generation: An Example

Consider a test generation scenario for verifying the interaction *"Decode in stall, and IALU, FADD3 in operation execution at the same time"*. Based on Algorithm 1, the property *F(Decode.stall ∧ IALU.exe ∧ FADD3.exe)* is generated from the interaction. Its negation will be *G(¬Decode.stall*

$\vee \neg \mathit{IALU.exe} \vee \neg \mathit{FADD3.exe})$. According to the horizontal and vertical partitioning, we can use a partial set of modules {Fetch, Decode, IALU, FADD1, FADD2, FADD3} to generate a test program. Based on the procedure of deciding bound for each property, bound will be 5. SAT-based BMC accepts the decomposed processor model, the negated property, and the bound. The generated test program is shown in Table 1 where Decode unit is in stall due to the read-after-write(RAW) hazard by FADD instruction.

Table 1: An Example of Test Program

Fetch Cycle	Instructions
1	FADD *R1* R2 R2
2	NOP
3	ADD R3 R2 R2
4	ADD R3 *R1* R2
5	NOP

Table 2: Comparison of Test Generation Techniques based on the Number of Interactions

Interaction Modules	Decomposed Design	UMC	SAT-based BMC	
			Max. k	Each k
1	Whole	**X**	5.63	0.48
	Group	**X**	3.87	0.22
2	Whole	**X**	7.42	0.65
	Group	**X**	4.31	0.43
3	Whole	**X**	7.74	0.70
	Group	**X**	5.72	0.52
4	Whole	**X**	8.79	0.75
	Group	**X**	6.98	0.64
5	Whole	**X**	9.29	0.89
	Group	**X**	8.31	0.62
6	Whole	**X**	9.58	1.05
	Group	**X**	9.04	0.68

5. A CASE STUDY

We applied our methodology on a simplified MIPS architecture [12], as shown in Figure 2. For our experiments, we used Cadence SMV [17] as a model checker and zChaff [13] as a SAT solver. We used 16 16-bit registers in the implementation of the register file. All the experiments were run on a 1 GHz Sun UltraSparc with 8G RAM.

Table 2 compares our test generation technique with UMC-based test generation for different module interactions. The first column specifies a set of properties based on the number of interactions. For example, the third row presents average test generation time (in seconds) for all properties consisting of two ("2") module interactions. The second column presents the level of decomposition used during test generation. The entry *whole* implies that no decomposition is used. The entry *group* implies that either horizontal or vertical or both decompositions are used. The next three columns show the performance of three test generation techniques: UMC, BMC using maximum bound, and BMC using bound for each property. The maximum bound 45 was used assuming that the longest length is taken by memory operations i.e., the sum of the IALU pipeline path length (5) and data-transfer path length (40). In the table, **X** indicates that a counterexample was not found due to "*Out of Memory*" problem. As expected, Table 2 shows that the test generation time grows with the increase of the number of module interactions. Bound for each property reduces the test generation time by 90% compared to using BMC with maximum bound.

6. CONCLUSIONS

Functional verification is a major bottleneck in processor design. Simulation using test programs is the most widely used form of processor verification. Use of directed tests in simulation can reduce overall validation effort since shorter tests can obtain the same coverage goal compared to the random tests. There is a need for automatic directed test generation techniques based on functional coverage goal. Test generation using model checking is one of the most promising approaches. However, this approach is unsuitable for large designs due to the state explosion problem.

This paper presented a directed test generation technique for pipelined processors using SAT-based bounded model checking. We developed a procedure for determining *bound* for each property. We also developed a method for decomposing design and properties in the context of SAT-based BMC. Our experimental results using MIPS processor demonstrated that our technique reduces both test generation time and memory requirement. In this paper, we used pipeline interaction fault model for test generation. Our future work includes analysis of our fault model by comparing it to existing coverage metrics such as code coverage and toggle coverage.

7. REFERENCES

[1] A. Adir et al. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test*, 2004.

[2] N. Amla et al. An analysis of SAT-based model checking techniques in an industrial environment. *CHARME*, 2005.

[3] A. Biere et al. Bounded model checking. *Advances in Computers*, 2003.

[4] A. Biere et al. Symbolic model checking without BDDs. *TACAS*, 1999.

[5] E. M. Clarke et al. *Model Checking*. MIT Press, 1999.

[6] F. Copty et al. Benefits of bounded model checking at an industrial setting. *CAV*, 2001.

[7] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engineering Notes*, 1999.

[8] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. *PDATE*, 2002.

[9] S. Gurumurthy et al. Automated mapping of pre-computed module-level test sequences to processor instructions. *ITC*, 2005.

[10] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[11] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. *DAC*, 2001.

[12] H. Koo and P. Mishra. Functional Test Generation using Property Decompositions for Validation of Pipelined Processors. *DATE*, 2006.

[13] M. H. Moskewicz et al. Chaff: Engineering an efficient SAT solver. *DAC*, 2001.

[14] M. R. Prasad et al. A survey of recent advances in SAT-based formal verification. *Intl. Journal on Software Tools for Technology Transfer (STTT)*, 2005.

[15] J. Shen et al. An RTL abstraction technique for processor microarchitecture validation and test generation. *Journal of Electronic Testing: Theory and Applications*, 2000.

[16] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. *DAC*, 1999.

[17] www-cad.eecs.berkeley.edu/ kenmcmil/smv. *Cadence SMV*.