Coverage-driven Functional Test Generation for Processor Validation using Formal Methods

Heon-Mo Koo
hkoo@cise.ufl.edu

Prabhat Mishra
prabhat@cise.ufl.edu

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA.

***Abstract*— Functional validation is one of the major bottlenecks in processor design: up to 70% of the design development time and resources are spent on functional verification. Simulation is the most widely used form of microprocessor validation. A major challenge in simulation-based validation is how to reduce the overall validation time and resources. Traditionally, billions of random tests are used during simulation and the lack of a comprehensive functional coverage metric makes it difficult to measure the verification progress. Directed test generation is a promising approach in terms of the volume of test set but poses several challenges. One of the major challenges is how to generate directed tests for complex processor designs for efficient functional validation. This paper presents functional coverage-driven test generation techniques using formal methods. We present fault models based on the functionality of pipelined processors. These fault models are used to define the functional coverage metric to measure the verification progress. We have developed automatic test generation techniques using model checking and SAT solving methods. The experiments using MIPS processor demonstrate the feasibility and usefulness of the proposed functional validation methodology.**

I. Introduction

Functional verification is one of the major bottlenecks in microprocessor design due to the combined effects of increasing design complexity and decreasing time-to-market. An exponential increase in design complexity results from advances in the VLSI technology and availability of increasingly complex applications in the domains of communication, multimedia, networking and entertainment. To accommodate such faster computation requirements, today's processors employ many sophisticated micro-architectures including deeply pipelined superscalar architectures.

Figure 1 summarizes a study [6] of the pre-silicon logic bugs found in the Intel IA32 family of microarchitectures. This trend again shows an exponential increase in the number of logic bugs: a growth rate of 300-400% from one generation to the next. The increase in logic bugs is proportional to the increase in design complexity. The increase in design errors makes verification tasks more difficult. In addition to the growing difficulty of pipelined processor verification, time-to-market has become shorter in the embedded processor designs. A recent study [13] has shown that functional verification accounts for significant portion (up to 70%) of the overall design development time and resources. As a result, design verification of modern processors is widely acknowledged as a major bottleneck in design methodology.

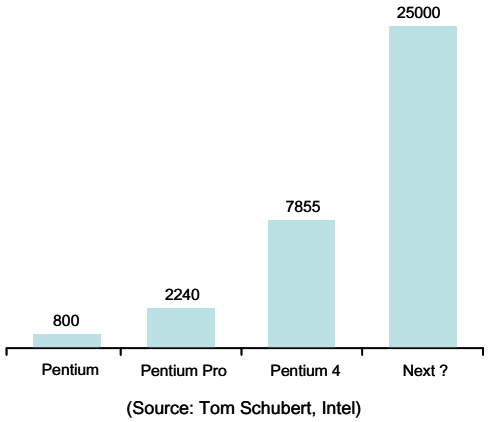

Fig. 1. Pre-silicon logic bugs per generation

A. Existing Techniques for Functional Verification

Existing verification techniques can be broadly categorized into formal methods [8], [18] and simulation-based techniques [7]. Formal methods provide the completeness of verification task by proving mathematically the correctness of designs. However, due to the state explosion problem, formal methods are generally used to verify components of design. Simulation-based validation handles complex designs but cannot guarantee complete verification. As a result, simulation-based validation is the most widely used form of processor verification.

Test programs consisting of instruction sequences are used for functional validation of processor designs. 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 handle this problem.

B. Challenges in Directed Test Generation using Formal Methods

Test generation using model checking [11] 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. This falsification can be quite effectively exploited for test generation. Instead of a desired property, its negated version is applied to the model checker to produce a counterexample. The counterexample contains a sequence of instructions 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. We present an efficient test generation technique that uses design level as well as property level decompositions to reduce test generation time and memory requirement.

As a complementary technique, SAT-based bounded model checking (BMC) has given promising results [5]. The basic idea is to restrict 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 [4]. However, finding *bound* is a challenging problem since the depth of counterexamples is unknown in general. We propose a method to determine the bound for each test generation scenario, thereby making SAT-based BMC feasible in practice.

C. Coverage-driven Functional Test Generation

Figure 2 shows the overall flow of the functional coverage-driven test generation process. The process begins by generating a list of functional faults in the processor design under validation. One of these faults is selected for test generation. In our approach, a functional test generator, e.g., model checking or SAT solver produces a test program for this fault. Once the test program is generated, fault simulation is performed to determine all of the faults that are detected by that test program. These detected faults are removed from the functional fault list. Another undetected fault is selected from this list and the process begins again. This loop is exited when all of the faults are detected.

This paper consists of three parts. First, we define functional pipeline interaction behaviors using a graph-based model of the processor. The negated versions of desired behaviors, called functional faults, are converted into temporal logic properties. Functional faults are used to define a functional coverage and generate test programs. Second, we formally specify the processor architecture in model checking language. The properties are decomposed into local properties and the formal processor model is partitioned into components

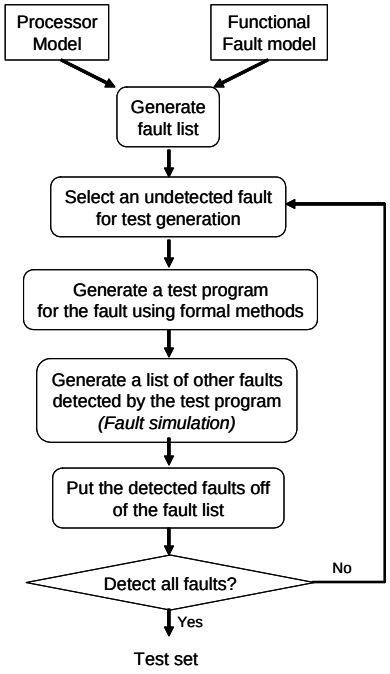

Fig. 2. Functional test generation flow

accordingly in order to avoid the state explosion problem in formal methods. Applying the decomposed properties and partitioned components to a model checker, counterexamples for individual components are generated and merged together at every clock cycle to form a counterexample for the entire processor. The counterexample contains a sequence of instructions to exercise the desired pipeline behavior. Finally, SAT-based bounded model checking makes it efficient to generate functional tests for large partitioning by restricting search space of counterexamples. We have successfully applied the formal-method-based test generation to MIPS processors.

II. Relate Work

A variety of techniques for generating test programs have been developed for architectural micro-architectural validation of pipelined processors. In [19], the processor model is described as a finite state machine (FSM) and the 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 [25] have proposed an RTL abstraction technique that creates an abstract FSM model while preserving clock accurate behaviors. Ur and Yadin [26] have also used abstraction of processor model to generate test programs for micro architectural validation of a superscalar PowerPC processor. Adir et al. [1] has 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 [9], [11] has been successfully used in software and hardware verification as a test generation engine [14], [22]. Model of design described in model checking language 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. [5] introduced bounded model checking (BMC) combined with satisfiability solving. The recent developments in SAT-based BMC techniques have been presented in [4], [10], [24]. 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 [15], [20], [23], [27] 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], [3]. The capacity increase of BMC techniques has become attractive for industrial use. An Intel study [12] 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. [16] have used BMC as test program generator for mapping pre-computed module-level test sequences to processor instructions.

III. Generation of Functional Faults

A functional coverage metric is necessary to evaluate the progress of functional validation. Several coverage metrics are commonly used during functional validation such as code coverage, toggle coverage, etc. However, these coverage metrics do not have a direct relationship with the design functionality. For example, none of the existing coverage metrics determines if all possible interactions of stalls are tested in a pipelined processor. Therefore, we need a coverage metric based on the functionality of pipelined processors. We define a pipeline interaction fault model using graph-based modeling of pipelined processors. The fault model is used to define a functional coverage as well as generate functional faults described in temporal logic properties. We use the functional coverage to measure the validation progress by reporting the faults that are not covered by a given set of test programs.

A. Functional Fault Model and Functional Coverage

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 [17] as shown in Figure 3. 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.

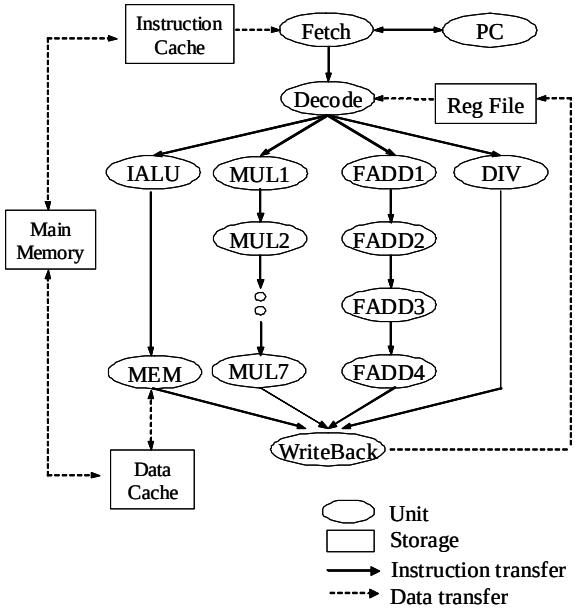

Fig. 3. Graph Model of the MIPS processor

Using the graph model shown in Figure 3, 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.

We need one test program to activate each fault. The functional coverage (FC) is defined as follows:

$$\text{FC} = \frac{\text{faults detected by test programs}}{\text{total detectable faults in the fault model}} \quad (1)$$

B. Conversion of Functional Faults into Properties

The main overhead in using model checking is to correctly describe temporal logic properties from desired behavior of designs. This is mainly due to the semantic gap between them. Since interaction faults are semantically explicit and a processor model is described in structure-oriented modules in general, interaction faults can be converted into temporal logic properties. 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)* is *G($\neg p_i$)* that 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"*.

IV. Test Generation using Formal Methods

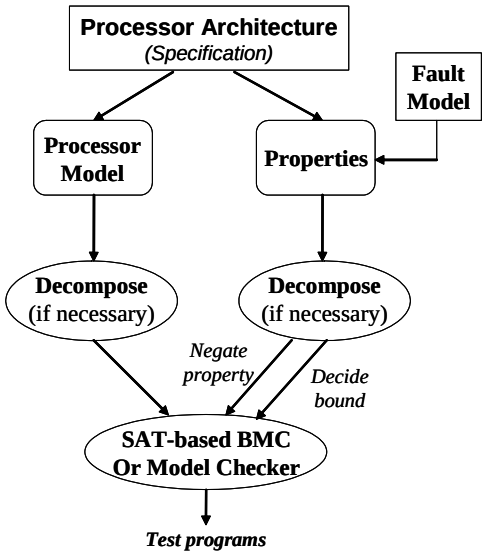

Fig. 4. Test Program Generation Methodology

Figure4 shows our test generation methodology. Processor model and properties are generated from the architecture specification. We use pipeline interaction fault model to define functional coverage. Temporal logic properties are created from pipeline interaction faults. We use design and property decompositions to improve the performance of test generation. We determine bound for each property for applying to formal methods such as SAT-based BMC or model checking, thereby, reducing test generation time and memory requirement compared to using the maximum bound for all properties. Processor model, negated property, and the bound are applied to SAT-based BMC or model checker to generate a test program. Based on the coverage report, more properties can be added, if necessary. We use design and property decompositions to further improve the performance of test generation.

A. Design and Property Decomposition

In this section, we describe how to decompose a processor model and the properties (already negated) for efficient model checking. Decomposition of a design plays a central role in the generation of efficient test programs. Ideally, the design should be decomposed into components such that there is very little interaction among the partitioned components. For a pipelined processor, the natural partition is along the pipeline boundaries, e.g., functional units. In other words, the partitioned pipelined processor can be viewed as a graph described in Section III-A where *nodes* consist of units (e.g., fetch, decode etc.) or storages (e.g., memory or register file) and *edges* consist of connectivity among them. Typically, instruction is transferred between units, and data is transferred between units and storages. This graph model is similar to the pipeline level block diagram available in a typical architecture manual.

Each property consists of temporal operators (G, F, X, U) and Boolean connectives ($\wedge$, $\vee$, $\neg$, and $\rightarrow$). There are various combinations of temporal operators and Boolean connectives where decompositions are not possible e.g., $F(p \wedge q) \neq F(p) \wedge F(q)$ and $G(p \vee q) \neq G(p) \vee G(q)$. In certain situations, such as pUq, $F(p \rightarrow F(q)$, or $F(p \rightarrow G(q)$, decompositions are not beneficial compared to traditional model checking. The following combinations allow simple property decompositions.

$$G(p \wedge q) = G(p) \wedge G(q) \qquad F(p \vee q) = F(p) \vee F(q)$$
$$X(p \vee q) = X(p) \vee X(q) \qquad X(p \wedge q) = X(p) \wedge X(q)$$

If we introduce the notion of clock (time step) in the property then more decompositions are allowed as shown below[1]. Note that the left and right hand side of the decomposition are not logically equivalent but they produce functionally equivalent counterexamples.

$$G((clk \neq t_s) \vee (p \vee q)) \approx G((clk \neq t_s) \vee p) \vee G((clk \neq t_s) \vee q)$$

For the proof of the above equation, let's put $A = G((clk \neq t_s) \vee (p \vee q))$ and $B \vee C = G((clk \neq t_s) \vee p) \vee G((clk \neq t_s) \vee q)$. A failure triggered by the latter $B \vee C$ always causes a failure that can be obtained using the former C. If U is the universal set, then A is a subset of $B \vee C$, which is a subset of U. Therefore, if we find a counterexample inside $U - (B \vee C)$ then it is always a counterexample of A.

Although we only use a few decomposition scenarios, it is important to note that these scenarios are sufficient for generating the properties where pipeline interactions are considered. Moreover, the property decomposition is dependent

[1]The clk variable is used to count time steps, and t_s is a specific time step.

on the design decomposition. For example, consider a design which has two partitions: d_1 and d_2. We cannot decompose a property into two sub-properties p_1 and p_2, if it is not possible to apply p_1 and p_2 to the partitions d_1 and d_2. In other words, if p_1 contains variables from both partitions, it is not possible to apply it to one partition of the design. However, we can change the partition based on the properties. For example, a property may not be decomposable based on a module level partitioning but it may be decomposable based on a pipeline path level partitioning. For example, in the graph model shown in Figure 3, the integer-ALU pipeline path PP_{IALU} = {Fetch, Decode, IALU, Mem, WriteBack} is treated as one of path level partitions.

B. Deciding 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 3, 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.

C. Test Generation by Merging Local Counterexamples

Algorithm 1 presents our test generation procedure using design and property decompositions. The basic idea of the algorithm is to apply the components of the properties (sub-properties) to appropriate modules and compose their responses to construct the final test program. This algorithm accepts design M and properties P as inputs and produces the test programs. It uses two lists to maintain the current (TaskList) and future (FutureList) tasks. Both lists have exactly the same structure. Each entry in the list contains a collection of sub-tasks that is applicable to a particular module. Therefore, each list can have up to *n* entries where *n* is the number of modules (or partitions) in the design. The tasks in the TaskList need to be performed in the current time step (clock cycle). The tasks in the FutureList will be performed in the next clock cycle. Initially both lists are empty.

For each property P_i, the algorithm generates one test program. Each property consists of one or more sub-properties based on their applicability to different modules or partitions in the design as discussed in Section **??**. The algorithm adds the sub-properties to the TaskList based on the module to which this property is applicable. The algorithm iterates over all the tasks (sub-properties) in the TaskList. It removes an entry (say k'th location) from the TaskList. In general, this entry can be a list of sub-tasks (due to simultaneous output requirements from multiple children nodes) that need to be applied to module M_k. These subtasks are composed to create the intermediate property P_i^k. The property P_i^k is applied to the module M_k using a model checker. The model checker generates a counterexample. The generated counterexample is analyzed to find the input requirements inp_k for the module M_k. If these are primary inputs then they are stored in PrimaryInputs list; otherwise for each parent node M_r, where inp_k is applicable, extract the output requirements for M_r. This output requirement is added to the r'th entry of the FutureList. Finally, if the tasks for the current timestamp is completed (TaskList empty), FutureList is copied to the TaskList and this process continues until both the lists are empty. This implies we have obtained the primary input assignments for all the sub-properties. These assignments are converted into a test program consisting of instruction sequences.

Algorithm 1: *Test Generation*
Inputs: i) Processor model M as a composition of modules
ii) Set of global properties P where each property is decomposed into multiple module level properties
Outputs: Test programs to verify the pipeline interactions.
Begin
TaskList = ϕ; FutureList = ϕ; TestPrograms = ϕ
for each property P_i in set P
for each sub property P_i^j
TaskList[j] = P_i^j /* P_i^j is applicable to M_j */
endfor
PrimaryInputs = ϕ
while TaskList is not empty
$items$ = RemoveEntry(TaskList)
P_i^k = ComposeRequirements($items$);
Apply P_i^k on module M_k using model checker
inp_k = input requirements for M_k from counterexample
if inp_k are not primary_inputs
for each applicable parent node M_r of M_k
out_r = Extract output requirements for M_r
FutureList[r] = FutureList[r] ∪ out_r
endfor
else PrimaryInputs = PrimaryInputs ∪ inp_k
endif
if TaskList is empty
TaskList = FutureList; FutureList = ϕ
endif
endwhile
$test_i$ = GenerateTest(PrimaryInputs).
TestPrograms = TestPrograms ∪ $test_i$
endfor
return TestPrograms
End

For illustration, consider a simple property P_1 to verify a multiple stall scenario consisting of IALU (3rd module) and DIV (15th module) nodes in Figure 3 at clock cycle 5. This property can be decomposed into two sub-properties P_1^3 (IALU not stalled in cycle 5) and P_1^{15} (DIV not stalled in cycle

5). This implies that TaskList will have two entries before entering the while loop: TaskList[3] = P_1^3 and TaskList[15] = P_1^{15}. At the first iteration of the while loop P_1^3 will be applied to M_3 (IALU) using model checker; generated counter example will be analyzed to find the output requirement for the Decode unit (2nd module in Figure 3) in clock cycle 4; the requirement will be added to FutureList[2]. During second iteration of the while loop P_1^{15} (TaskList[15]) will be applied to M_{15} (DIV); generated counter example will be analyzed to find the output requirement for the decode unit in clock cycle 4; the requirement will be added to FutureList[2]. At this point, the TaskList is empty and the FutureList has only one entry with two requirements which is copied to the TaskList. At the third iteration of the while loop, these two requirements are composed into an intermediate property and applied to M_2 (Decode) that generates requirements for Fetch node. Finally, the fourth iteration applies the corresponding property to the Fetch unit that generates the primary input assignments. These assignments are converted to a test program.

V. A Case Study

We performed various test generation experiments to validate the pipeline interactions by varying interactions of functional units and decompositions of design and properties. We excluded illegal interactions based on the fact that their negated properties could not generate any counterexample. In this section, we present our experimental setup followed by test generation example using horizontal and vertical decompositions. Next, we compare our test generation technique with UMC-based test generation method as well as BMC using the maximum bound.

A. Experimental Setup

We applied our methodology on a simplified MIPS architecture [17], as shown in Figure 3. We chose the MIPS processor since it has been well studied in academia and there are HDL implementations available for the processor that can be used for validation purposes. Additionally, the MIPS processor has many interesting features, such as fragmented pipelines and multi-cycle functional units that are representatives of many commercial pipelined processors such as TI C6x and PowerPC.

For our experiments, we used Cadence SMV [21] as a model checker and zChaff [23] as a SAT solver. We used 16 16-bit registers in the register file for the following experiments. All the experiments were run on a 1 GHz Sun UltraSparc with 8G RAM.

B. 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"*. The property *F(Decode.stall* $\wedge$ *IALU.exe* $\wedge$ *FADD3.exe)* is generated from the interaction. Its negation will be *G(*$\neg$*Decode.stall* $\vee$ $\neg$*IALU.exe* $\vee$ $\neg$*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 I where Decode unit is in stall due to the read-after-write(RAW) hazard by FADD instruction.

TABLE I
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

C. Results

Table II 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. Similarly, the entry *module* implies that the test generation uses module-level decomposition. 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

TABLE II
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
	Module	0.40	2.24	0.42
2	Whole	**X**	7.42	0.65
	Group	**X**	4.31	0.43
	Module	0.57	6.41	1.38
3	Whole	**X**	7.74	0.70
	Group	**X**	5.72	0.52
	Module	0.86	6.41	1.45
4	Whole	**X**	8.79	0.75
	Group	**X**	6.98	0.64
	Module	1.12	7.63	1.97
5	Whole	**X**	9.29	0.89
	Group	**X**	8.31	0.62
	Module	1.50	9.03	2.18
6	Whole	**X**	9.58	1.05
	Group	**X**	9.04	0.68
	Module	1.86	10.70	2.50

X: Not Applicable.

that a counterexample was not found due to "*Out of Memory*" problem.

As expected, Table II shows that the test generation time grows with the increase of the number of module interactions. UMC can be used only with module level decompositions while SAT-based BMC can be used without decomposition. Bound for each property reduces approximately 90% of the test generation time compared to using BMC with maximum bound. An interesting observation is that UMC with module level decomposition provides better performance than SAT-based BMC. This is because the time to unfold the model and convert it to SAT problem is more than the time to search a counterexample.

VI. Conclusion

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 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 efficient directed test generation techniques for validation of pipelined processors using formal methods as well as search restriction techniques. Our methodology made three important contributions. First, it presented pipeline interaction faults and they are converted into temporal logic properties in order to be applied to formal methods. Second, it developed a procedure for determining *bound* for each property. Finally, it developed a method for decomposing design and properties in the context of SAT-based BMC and UMC. Our experimental results using MIPS processor demonstrated that the proposed test generation technique is a promising approach. It is important to note that due to state space explosion problem naive UMC-based test generation failed in many scenarios.

In this paper, we used pipeline interaction fault model for test generation. Our future work includes applying our technique for test generation using other fault models such as stuck-at and FSM transition faults. Since the number of interactions (directed tests) can be still extremely large, we plan to develop a test compaction technique to reduce the number of test programs for functional validation of pipelined processors.

References

[1] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[2] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 254–268. Springer, 2005.

[3] N. Amla, R. Kurshan, K. McMillan, and R. Medel. Experiment analysis of different techniques for bounded model checkings. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 2619 of *LNCS*, pages 34–48. Springer, 2003.

[4] A. Biere, A. Cimatti, and E. M. Clarke. Bounded model checking. *Advances in Computers*, 58, 2003.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[6] Bob Bentley. High level validation of next-generation microprocessors. In *Proceedings of High Level Design Validation and Test (HLDVT)*, pages 31–35, 2002.

[7] R. E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM (JACM)*, 38(2):299–328, 1991.

[8] P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE Computer*, 21(7):8–19, 1988.

[9] E. M. Clark and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[10] E. M. Clarke, A. Biere, R. Ramimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design (FMSD)*, 19(1):7–34, 2001.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[12] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. of Intl. Conference on Computer Aided Verification (CAV)*, LNCS, pages 436–453. Springer, 2001.

[13] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proc. of Design Automation Conference (DAC)*, pages 286–291, 2003.

[14] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 146–162, 1999.

[15] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[16] S. Gurumurthy, S. Vasudevan, and J. A. Abraham. Automated mapping of pre-computed module-level test sequences to processor instructions. In *Proc. of Intl. Test Conference (ITC)*, 2005.

[17] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Sanfrancisco, CA, 2003.

[18] C. Kern and M. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[19] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. In *Proc. of Design Automation Conference (DAC)*, pages 816–821, 2001.

[20] J. P. Marques-Silva and K. A. Sakallh. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[21] K. L. McMillan. *SMV Model Checker, Cadence Berkeley Laboratory*. http://embedded.eecs.berkeley.edu/Alumni/kenmcmil/smv, October, 2002.

[22] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 182–187, 2004.

[23] M. H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of Design Automation Conference (DAC)*, pages 530–535, 2001.

[24] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *Intl. Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

[25] J. Shen and J. A. Abraham. An RTL abstraction technique for processor microarchitecture validation and test generation. *Journal of Electronic Testing: Theory and Applications*, 16(1-2):67–81, 2000.

[26] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. In *Proc. of Design Automation Conference (DAC)*, pages 175–180, 1999.

[27] H. Zhang. SATO: An efficient propositional prover. In *Proc. of International Conference on Automated Deduction (CADE)*, volume 1249 of *LNCS*, pages 272–275. Springer, 1997.