Automatic Test Program Generation for Out-of-Order Superscalar Processors

Ying Zhang Ahmed Rezine Petru Eles Zebo Peng
Embedded Systems Lab, Linköping University, Sweden
{ying.zhang, ahmed.rezine, petru.eles, zebo.peng}@liu.se

Abstract—**This paper presents a high-level automatic test instruction generation (HATIG) technical that allows, for the first time, to test the scheduling unit of an out-of-order superscalar processor. This technique leverages on existing bounded model checking tools in order to generate software-based self-testing programs from a global EFSM model of the processor under test. The experimental results have demonstrated the efficiency of the proposed technique.**

Keywords: Automatic Test Instruction Generation; Bounded Model Chekcing; Software-Based Self-Testing; Out-of-Order Superscalar Processor.

I. INTRODUCTION

Hardware defects become more common as silicon designs include more nanometer scale components [1]. A promising and effective solution to uncover these defects is to use online self-testing by executing test instructions. Such a software-based self-testing (SBST) approach makes use of the functionality provided by a processor in order to apply tests for structural faults [2]. In this approach, test programs composed of test code (i.e., instructions) and test data are first downloaded into the processor under test (PUT). Then, the processor executes the instructions and tests at-speed its component under test (CUT). The resulting test responses are finally written back to memory for observation. SBST enables powerful online fault detection for processors without the need for expensive high speed automated test equipment. However, this approach has not been able to handle complex superscalar processors so far.

Mainstream SBST methods develop test programs by combining functional constraint extraction and gate-level constrained automatic test pattern generation (ATPG). They avoid test patterns that never appear during the normal operational mode (i.e., functional mode) of a processor with the help of the extracted functional constraints. Originally, SBST used manually extracted constraints to restrict the randomly generated test patterns that were applied to the components of the processor under test [2]. This approach is referred to as random patterns based SBST (RSBST). Later, a more scalable SBST technique [3] was presented for more complex processors. It applied statistical regression to extract instruction-level constraints for constrained ATPG (CATPG) [4]. In an effort to achieve more efficiency than CATPG, learning methods [5] have been used to deduce, based on simulation runs, relations between instructions, I/O data and component signals. In [6], an automatic test instruction generation (ATIG) approach was proposed where instruction-level constraints are directly mapped to the ports of the CUT and test patterns are automatically translated into test programs.

Due to the high complexities involved when dealing with gate-level implementations, many researchers shifted their attention to the register-transfer (RT) level for generating targeted instruction sequences for SBST. In [7], deterministic tests at RT-level take advantage of the inherent regularity of the functional components of a processor. It carefully designs test programs to achieve considerable high fault coverage on functional components using compact patterns. Based on these deterministic tests, systematic [8] and hybrid [9] [10] [11] SBST methods were proposed and applied successfully to test single-scalar pipeline processors. However, to the best of our knowledge, there has been no work targeting superscalar processors with out-of-order instruction execution.

Bounded model checking (BMC) has been used to generate SBST programs in the context of single-scalar pipeline processors [12]. The technique presented in [12] translates a pre-computed test pattern into an assertion. A BMC tool is then used to check if there is a sequence of instructions that can violate the assertion. If it is the case, the sequence of instructions becomes the SBST test program directly. Although BMC tools typically suffer from time-outs and state space explosion problems, they do provide a systematic way to handle complex units in the processors under test.

Both mainstream SBST and RT-level methods have their limitations when testing out-of-order superscalar processors. For instance, gate-level methods can extract functional constraints for combinational circuits, but there is still no effective way to extract functional constraints for sequential circuits. Complex sequential control units are inevitable when out-of-order executions are implemented on superscalar processors. Omitting to extract constraints for these sequential units would seriously impact the resulting fault coverage. For RT-level methods, concurrent and out-of-order execution results in a combinatorial explosion of the number of valid instruction sequences. In this context, manually generating test programs that can cover all functional paths of the sequential components is a daunting and intractable task. On the other hand, for the sequential control unit in superscalar processors, functional programs only excite control signals of functional paths, but never consider the other signals. There is therefore usually no guaranty that these functional programs can achieve a certain fault coverage level for high-quality manufacturing tests.

In this paper, for the first time, the scheduling unit of an out-of-order superscalar processor is tested with software-based self-testing programs. Our method, referred to as high-level automatic test instruction generation (HATIG), makes use of bounded model checking in order to analyze a global EFSM model of the unit under test and to generate SBST programs. First, the whole superscalar processor is modeled as a global EFSM and slicing is applied to reduce the size of the global EFSM for the scheduling unit. Second, test instruction sequences that excite certain functional paths within the scheduling unit, called leading sequences, are automatically generated using an off the shelf bounded model checking tool. Third, the sequential scheduling unit is flattened into a combinational one on which the test instruction sequences are imposed as functional constraints. ATIG is then used to generate the test programs. In this way, test generation is guided to travel through all functional paths within the scheduling unit. Our high-level ATIG approach permits to treat the scheduling unit with its large sequential depth as a combinational one. It therefore provides an effective way to detect faults in out-of-order superscalar processors. Some of these faults are particularly challenging to uncover as they require several time-frames to propagate or to get activated.

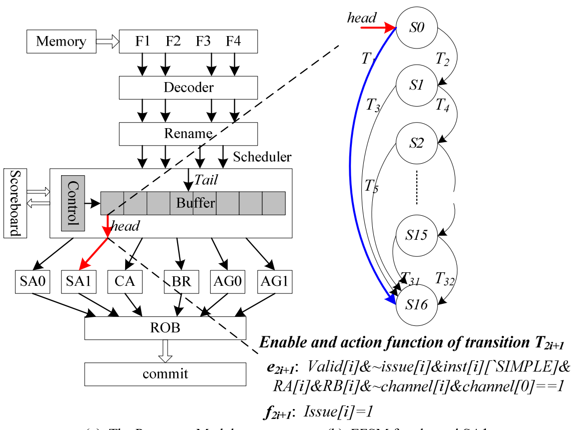
This paper is organized as follows. Section II briefly introduces the out-of-order superscalar processor we consider as a running example in this paper. It also describes the extraction of the global EFSM and the way bounded model checking is used. Section III details the three main steps of the HATIG approach. Section IV reports on the experimental results and Section V concludes the paper.

II. Background

An out-of-order superscalar processor often relies on extremely complex structures. This poses a great challenge for the functional test on the involved structures. In order to automatically generate SBST programs to test such structures, we model them using global "extended finite state machines" (EFSM) [13] and make use of bounded model checking [14].

A. *Superscalar and Out-of-order Processors*

We illustrate our approach on a 4-issue superscalar and out-of-order processor described in the Illinois Verilog model (IVM) [15]. The model is depicted in Fig.1(a). It contains 12 pipeline stages and 132 types of instructions. At each clock cycle, four instructions are fetched from memory. After the decoding and renaming steps, the instructions are stored in the buffer of the scheduling unit, starting from the tail pointer *tail*. Each instruction waits in the buffer to be issued to one of the six different channels, namely *SA1* and SA2 for the two simple ALUs, *CA* for the complex ALU including the multiply unit, *BR* for the branching unit, and *AG0* and *AG1* for the two address generating units. The control part of the scheduling unit arbitrates among the instructions that are ready to be issued. For example, if a multiply instruction *mull* has to be issued to the channel *CA*, it must first be stored in the scheduling unit and its operand registers have to be prepared. Then, if *CA* is not used by some other instruction, the scheduling unit will issue *mull* regardless of its order. To orchestrate this, an instruction sequence, namely a leading sequence, is used to initialize and to control the scheduling unit. After having been issued, instructions can be committed in order with the reorder buffer.



(a) The Processor Model (b) EFSM for channel SA1

Fig.1. An out-of-order and 4-issue superscalar processor example

This work mainly focuses on the scheduling unit which is at the core of the out-of-order capabilities of a processor. As shown in Fig.1(a), the unit checks the stored instructions from the head pointer, *head*, at each clock cycle, and the first ready instruction will obtain access to the output channel. It often requires a large number of clock cycles to initialize, issue or return the results of the stored instructions. As a result, there are many faults that need many time-frames to get activated or to propagate. This makes effective testing of the scheduling unit extremely difficult.

B. *Global EFSM*

We use extended finite state machines (EFSM) to capture a high level model of the structure of the complex superscalar processor. An EFSM is a finite state machine (FSM) extended with variables and predicates on their values. It can succinctly capture sets of values for the variables and therefore allows to greatly reduce the size of the manipulated representations. EFSMs [16] are widely used to model large systems and even whole processors. Formally, an EFSM is a seven-tuple *{S, I, O, D, E, F, T}*, where:

S is a set of symbolic states,

I is a set of input symbols,

O is a set of output symbols,

D is an *n*-dimensional space $D_1 \times \ldots \times D_n$,

E is a set of enabling functions e_i such that $e_i : D \rightarrow \{0,1\}$,

F is a set of update functions f_i such that $f_i : D \rightarrow D$, and

T is a transition relation such that $T : S{\times}E{\times}I \rightarrow S{\times}F{\times}O$.

An EFSM can be easily extracted from an HDL description by considering each decision node as a symbolic state of the EFSM [17]. Conditions and operations on the nodes are then used to extract the corresponding enabling functions and update actions. For example, the EFSM that captures the part of the scheduling unit that arbitrates among the instructions that are ready for the channel *SA1* is shown in Fig.1(b). In this EFSM, there are 16 decision nodes corresponding to 16 cells in the buffer. The i_{th} decision node has two transitions T_{2i+1} and T_{2i} indicating if the functional path from the *(head+i)*$_{th}$ cell to the channel is excited or not. To excite T_{2i+1}, the instruction in the *(i+head)*$_{th}$ cell has to be valid but not yet issued (*valid[i]&~issue[i]*), the instruction must belong to the simple ALU group (*inst[i][`SIMPLE]*), its two operand registers must be ready (*RA[i] & RB[i]*), and the channel *SA1 should* not be used (*~channel[1]*). The resulting enabling function e_{2i+1} for the transition is presented in Fig.1(b). In addition, the *SA1* channel has lower priority than that of the *SA0* channel, so the signal *channel[0]* is applied to avoid using *SA1* before the channel *SA0* has been used. Finally, the operation *issue[i]=1* corresponds to the action f_{2i+1} of the transition, and it means that the functional path is excited and that the instruction is issued into the channel *SA1*.

Global EFSMs were originally proposed to model superscalar processors to verify all pipeline interactions [13]. In that work, every instruction in the pipeline is modeled as a single EFSM, and every component in the processor is also modeled as an EFSM. All EFSMs are then combined together as one global EFSM. In this work, we only focus on the components of the processor and directly generate instruction sequences from a global EFSM. We need not consider how instructions are pipelined. For this reason, we build on our earlier work [6] and replace the EFSMs associated to the instructions by records called extended instructions (EIR). Each of these records is associated to an instruction and contains all relevant running information for the current cycle, as follows.

$$EIR = (I,\ D_1,\ D_2,\ R,\ F,\ A,\ op)$$

An EIR record contains, as indicated above, seven elements: the instruction itself *I*, two operands D_1 and D_2, the result *R*, a flag *F*, an address *A*, and the instruction type *op*. In this way, most variables of the EFSM are EIRs instead of single variables, and the leading sequences can be generated from the modified global EFSM directly without the need for extra EFSMs to model the instructions traversing the pipeline. The obtained global EFSM includes all valid pipeline interactions. As a result, the test instruction group generated from the global EFSM will satisfy all functional constraints imposed by the processor.

C. *Bounded Model Checking*

Bounded model checking (BMC) [14] is an automatic formal verification technology that is particularly suitable for analyzing EFSMs and for finding input sequences violating specifications supplied by the user. Given a property, a bounded model checking tool will exhaustively look for sequences that violate the property and that are of lengths smaller or equal to a bound *K*. If the tool terminates without returning a sequence, then there are no sequences that violate the property within this time bound *K*.

For example, to obtain a leading sequence that issues the instruction in the $(head+i)_{th}$ cell of the scheduling unit, we can use the specification "*SPEC AG(~(*e_{2i+1}*))*" that asserts that issuing an instruction from the $(head+i)_{th}$ cell to channel *SA1* is never possible. With a sufficient time bound *K*, a BMC tool automatically finds a counter example that violates the specification and that satisfies the enabling function of the transition T_{2i+1} (recall that T_{2i+1} is the transition that issues the instruction in the $(head+i)_{th}$ cell to *SA1*). The counter example provides an input instruction sequence that we use as a leading sequence to excite the transition T_{2i+1}.

III. High-Level Test Instruction Generation For Superscalar Processors

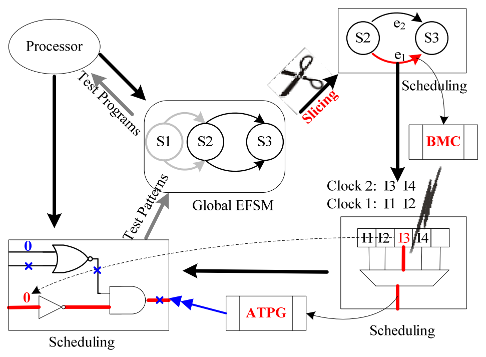
We propose a high-level automatic test instruction generation method to design SBST test programs for the scheduling unit of out-of-order superscalar processors. The method builds on both high-level information and gate-level descriptions. On the one hand, instruction-level leading sequences are extracted using a BMC tool in order to excite all functional paths in the considered scheduling unit. On the other hand, the obtained leading sequences are used to guide, at the gate level, test generation targeting structural faults. This combination enables us to achieve high fault coverage.

Applying the HATIG method on the scheduling unit of an out-of-order superscalar processor involves three main steps as depicted in Fig.2(a). In the first step, a global EFSM that models the whole processor is extracted as described in Section II. Slicing is then used to obtain a model that captures the scheduling unit while preserving the constraints imposed by the other components of the processor. In the second step, bounded model checking is applied on the obtained model in order to automatically generate leading instruction sequences that excite all functional paths of the scheduling unit. In the third step, the scheduling unit is flattened and the leading sequences generated in the previous step are mapped to the ports of the obtained circuit in order to excite the corresponding functional paths of the scheduling unit. Test patterns are then generated using ATPG and are automatically translated into test programs. A detailed description of the three steps follows.

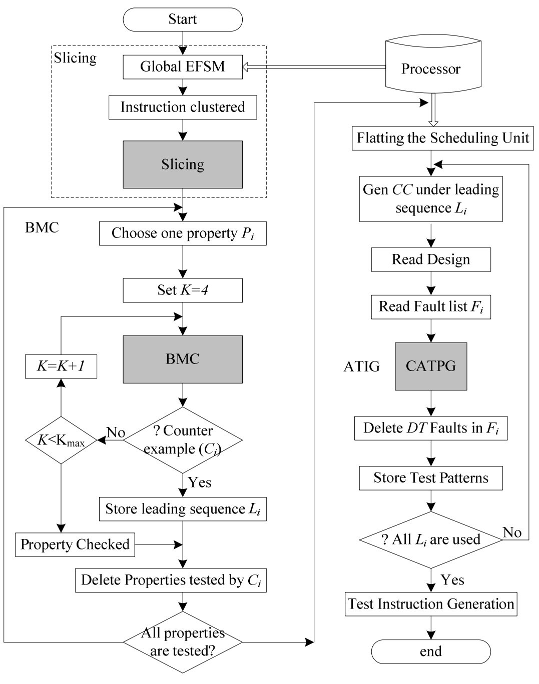
A. *Slicing on the Global EFSM for the Scheduling Unit*

To reduce the design space, we apply slicing techniques [18] before using BMC on superscalar processors. To illustrate this, the IVM processor contains approximately 50,000 bit registers [15]. Considering that it takes at least 12 clock cycles to complete one instruction, giving the whole IVM as input for the BMC tool would force the latter to exhaustively explore a state space of at least $(50000)^{12}$ in size. This explosion in the size of the state space to be explored typically exceeds the capabilities of existing model checkers.

Slicing away details that are irrelevant to the function of the scheduling unit greatly reduces the size of the manipulated model [18], thereby enabling the application of a BMC tool. The global EFSM is first extracted from the HDL description of the processor as described in Fig.2(b). Since the scheduling unit issues instructions to five different executing units, the 132 instruction types are clustered into six instruction groups: *ADD, SLL, MUL, CMOV, BEQ*, and *LD*. The conditional move instructions are clustered in an independent group *CMOV* because they are fetched only one instruction per cycle with the three other concurrent instructions being forced to be invalid.



(a) The framework of HATIG



(b) The flow chart of HATIG

Fig. 2. High-level automatic test instruction generation

In this work, the variables appearing in the enabling functions e_i of the transitions of the scheduling unit, such as the variables corresponding to the valid and to the register ready signals, are defined as chosen variables. The transitions and states that do not affect these variables are said to be non-contributing. A description of the EFSM resulting from slicing away non-contributing variables and transitions is shown in Fig.3. In addition to the scheduling unit, transitions of other components of the global EFSM are preserved by the slicing step because of their contribution to the chosen variables. These include transitions in the fetch stage, in the executing unit, and in the scoreboard. Transitions in the fetch stage are kept because they force the valid bits of the instructions that follow a *CMOV* or *BEQ* instruction to be invalid. Also, the feedback transitions in the executing unit are retained because they are used to delete instructions stored in the scheduling unit. In a similar manner, parts of the scoreboard unit that store the register ready signals are also preserved by the slicing step. These additional transitions correspond to the functional constraints imposed by components other than the scheduling unit. The size of the input and output EIRs of the resulting model is strongly reduced and the new f_i actions only contain the contributing variables. In other words, slicing greatly reduces the size of the model for the scheduling unit without violating the sequential constraints imposed by the other components of the processor.

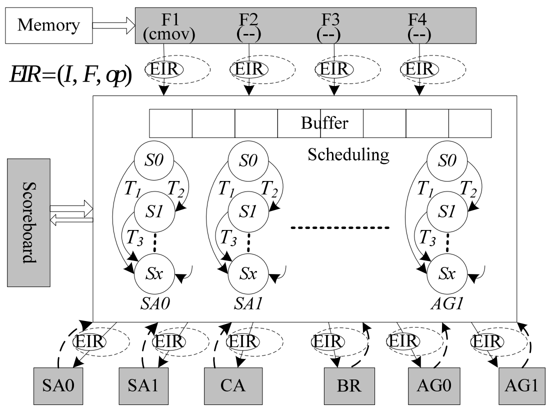


Fig.3. The sliced global EFSM targeting the scheduling unit

We use Table I to give an idea of the reductions achieved by the slicing step. The sliced global EFSM only contains 149 state variables and 1477 combinational variables. Compared to the scheduling unit of the original EFSM, the sliced EFSM has only 7.5% as many state variables and 1% as many combinational variables. This significant reduction alleviates state space explosion problems when generating, in the following step, leading sequences using BMC tools.

TABLE I
MODEL SIZE OF THE SCHEDULING UNIT AND THE SLICED GLOBAL EFSM

Variables	State Var. (bits)	Com. Var. (bits)
The Complete Scheduling Unit	1976	148666
The Sliced Global EFSM	149	1477
Percentage (EFSM/The S Unit)	7.5%	1.0%

B. *Leading Sequences Generated using BMC*

We use an off the shelf bounded model checker (the SMV tool [19]) in order to automatically generate sequences that excite certain functional paths in the EFSM of the scheduling unit resulting from slicing. From now on, we mean the sliced global EFSM when we mention the global EFSM. We refer to the instruction sequences obtained in this step as the leading sequences. As an example, assume the global EFSM contains three *CMOV* instructions initially. In addition, suppose the instruction in the 11_{th} cell of the buffer has to be issued into channel *SA1* when *head* points to the 3_{rd} cell. Observe that the instruction is at the $(head+i)_{th}=(3+8)_{th}=11_{th}$ cell. To excite this functional path, the transition T_{2*8+1} (the T_{2i+1} of the 8_{th} node) in the EFSM of channel *SA1* has to be excited, so that the specification sent to the BMC tool to be checked is "*SPEC AG(~(*e_{17} *& (head==3)))*". The tool generates a sequence that violates the property, hence exciting the considered functional path (here within a time bound of 4 clock cycles).

The situation is illustrated in Fig.4. The scheduling unit has three *CMOV* instructions in the initial state C_0, where the second *CMOV* is just issued, and the branch instruction *BEQ* is the input instruction in C_0. The head and tail pointers of the buffer of the scheduling unit are respectively represented by a light and a dark pointer. The fetch unit forces the instructions concurrent to a *BEQ* instruction to be invalid because the instructions following a *BEQ* instruction are often uncertain. In the first cycle C_1, the feedback of the first *CMOV* instruction arrives, so the later is deleted from the buffer and the head pointer *head* advances to the 0_{th} cell. In the same cycle, the third *CMOV* is issued and four load instructions *LD*s become the input instructions in C_1. In the second cycle C_2, the second *CMOV* instruction is deleted and *head* moves to the 1_{st} cell while two *LDs* and two *ADD*s instructions are input. In the third cycle C_3, *head* points to the 2_{nd} cell after deleting the *BEQ* instruction, and the first couple of *LD* instructions are issued into the channels *AG0* and *AG1*. At this point, the second pair of *LD* instructions has to wait since there are no available channels. In the fourth cycle C_4, *head* moves to the 3_{rd} cell. The second pair of *LD* instructions is issued and the third pair of *LD* instructions has to wait. The channels *SA0* and *SA1* are available and the pair of *ADD* instructions is issued before the third pair of *LD* instructions regardless of their previous order. At this point, *head* points to the 3_{rd} cell and the *ADD* instruction in the 11_{th} cell is issued into the channel *SA1*. The functional path (in red in the figure) is successfully excited by this sequence. The third cycle, namely C_3, where the leading sequence has initialized the scheduling unit, so that the functional path can be excited in the next cycle, is called the ready cycle C_R. The input instructions from the initial cycle to the ready cycle define the leading sequence.

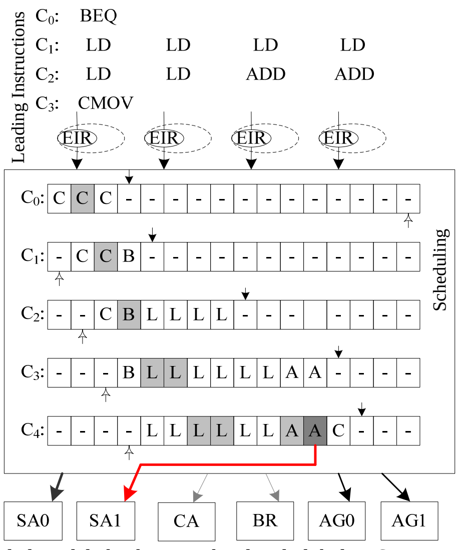

Leading Instructions
C0: BEQ
C1: LD LD LD LD
C2: LD LD ADD ADD
C3: CMOV
EIR
Scheduling
C0: C C C - - - - - - - - - - - - - -
C1: - C C B - - - - - - - - - - - - -
C2: - - C B L L L L - - - - - - -
C3: - - - B L L L L L L A A - - - -
C4: - - - - L L L L L L A A C - - -
SA0
SA1
CA
BR
AG0
AG1

Fig.4. Bounded model checking on the sliced global EFSM

We use the algorithm in Fig.2(b) to generate all leading sequences for all the functional paths. We consider that a functional path depends on both the transition of the EFSM and on the value of the head (or tail) pointer. We assume that if all transitions of the EFSM are covered whatever the value of the head (or tail) pointer is, then all of the functional paths are excited. The algorithm of Fig.2(b) therefore iterates through all pairs of transitions from the global EFSM and values for the head pointer. For each such property, the BMC tool is initially started with a time bound *K* equal to 4. If a sequence is found, then the property, as well as any additional properties excited by this sequence, is removed from the property set. If no sequences are found for the considered bound, the analysis is repeated after having increased K up to K_{max} = 16. If a property does not result in a leading sequence within the maximal time bound K_{max}, it is deemed to hold for any bound and is removed from the property set. We report in Table II on both maximum and average bounds *K*, execution times in seconds and consumed memory in Bytes. SMV was able to efficiently check all properties and did not timeout on any of them.

Finally, it was possible to generate the leading sequences for all of the considered functional paths except for some redundant ones. Indeed, some functional paths, called redundant paths, are never excited during the operations of the processor. For example, the functional paths where the instruction in the $(head)_{th}$ cell is issued into channel *SA1* are never excited because *SA0* is not used when the $(head)_{th}$ cell is pointed to by *head* and *SA0* has a higher priority than *SA1*. The instruction in the $(head)_{th}$ can therefore not be issued to the *SA1* channel. For these redundant paths, the algorithm of Fig.2(b) will reach the maximal bound $K_{\max}$ before removing them from the property set. In Table III, 672 sequences are generated to cover 2048 of a total of 2080 functional paths in the scheduling unit. This corresponds to about 98.5% of functional coverage.

TABLE II
THE COST FOR BOUNDED MODEL CHECKING

Value	Max	Average
Time Bound K	15	7
Time (s)	5.38	3.80
Memory (Bytes)	278,497	185,216

TABLE III
FUNCTIONAL TEST COVERAGE FOR THE SCHEDULING UNIT

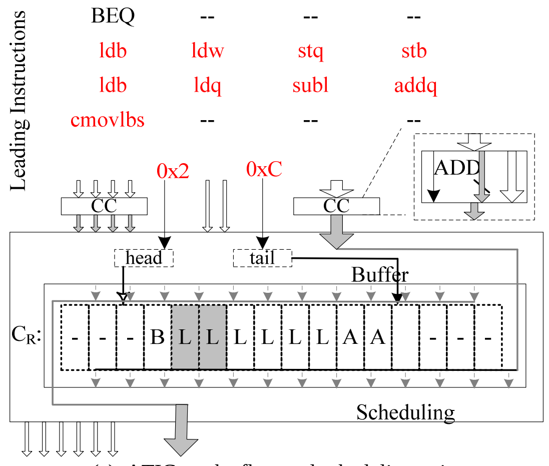
Total Functional Path	2080
Leading Sequence	672
Excited Functional Path	2048

C. *ATIG Guided by Leading Sequences*

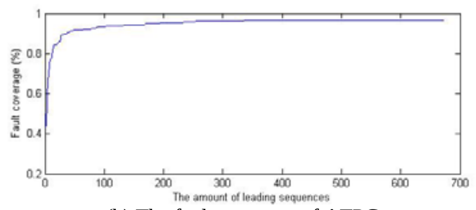
We can completely test the functionally testable faults within the scheduling unit by guiding the test generation using leading sequences for all functional paths. More specifically, we generate the test patterns just at the ready cycle when the considered leading sequence has appropriately initialized the scheduling unit in order to excite the related functional path. We detail this process in the following.

First, the scheduling unit is flattened by removing its inner registers and by transforming the inputs, respectively outputs, of the removed registers into new primary outputs (PO), respectively primary inputs (PI), of the unit. This is illustrated in Fig.5(a). The values of the inner registers in the ready cycle are then imposed on these PIs resulting in a constrained circuit (CC). A constrained circuit involves three parts. First, the old PIs for the four input instructions are constrained by the input instructions of the ready cycle, while the new PIs for the 16 cells of the buffer are constrained by the part of the leading sequence stored in the buffer. For example, the PIs for the 11th cell in Fig.5(a) are constrained by the second ADD instruction of the leading sequence. This instruction type *op* belongs to the simple ALU group (ADD). The controlling signals are set based on the instruction type *op*, and the rest of the signals are left unconstrained and can be searched by ATPG. Second, the PIs of the chosen variables (i.e., the variables appearing in the enabling function used to generate the functional path) are set according to their values at the ready cycle (the sequence generated by the BMC tool constrains the variable values in each cycle). Third, the PIs of the remaining registers are left unconstrained and can also be searched by ATPG. In this way, the flattened unit is at a similar state as the one of the scheduling unit at the ready cycle. If ATPG is applied to the constrained unit, all detectable faults along the excited functional path can be tested by changing the instruction types or the values of the unconstrained signals.

The algorithm in Fig.2(b) is proposed to test all functionally testable faults using the leading sequences generated with the help of the BMC tool. Given a new leading sequence, the algorithm uses the leading sequence to generate a constrained circuit and imposes it on the flattened unit. It then loads the constrained unit as well as the list of all possible faults and applies constrained ATPG on the resulting unit. It removes all of the detected faults from the fault list, and stores the test patterns. The algorithm continues until there are no remaining leading sequences to be used. This method can effectively detect all functionally testable faults in the scheduling unit because the leading sequences already excite all functional paths and because ATPG excites all testable faults along each functional path. As depicted in Fig.5(b), fault coverage increases in relation to the number of used leading sequences. In our experiments, the method achieved 96.6% fault coverage during test pattern generation. Finally, each test pattern generated from the constrained ATPG application is first translated as a group of test instructions using the ATIG method proposed in [6]. The test instructions from *tail* to *head* in the buffer are set using the leading sequence as shown in Fig.5(a). The test program is obtained after inserting observing instructions and a number of *CMOV* instructions. The *CMOV* instructions are inserted at the end of the test sequence to make the scheduling unit return to its initial state without using an external reset signal.



(a) ATIG on the flattened scheduling unit



(b) The fault coverage of ATPG

Fig.5. HATIG on the flattened scheduling unit and ATPG fault coverage

IV. EXPERIMENTAL RESULTS

We applied the HATIG approach to generate test programs for the IVM processor [15]. We evaluate the resulting fault coverage on the scheduling unit. We use random testing with the same number of instructions and sequential ATPG for comparison. We introduced three modifications to the IVM processor. First, the buffer sizes of the scheduling unit and the reorder buffer are reduced to 16 cells as we encountered out-of-memory problems when synthesizing the original IVM. Second, since we focus on the scheduling unit, we bypass both the rename unit and the data cache. WAR and WAW data dependencies are resolved on software level. Third, we delete the redundant segments directly from the RT-level description of the scheduling unit. Examples of such segments are the PC_TAG signals that do not affect the outputs of the processor.

TABLE IV
THE SBST GENERATION TIME OF HATIG

	Time (s)
Bounded Model Checking	2551.0
ATPG on the Flattened Unit	530.7
Test Program Generation Time	70.0
Total	3151.7

With our HATIG method, the slicing step greatly reduces the size of the global EFSM and the SMV tool only takes 2551.0s to generate the leading sequences for all the functional paths. In the test pattern generation step, the sequential ATPG problem on the scheduling unit is turned into a combinational one. As a result, the ATPG algorithm does not need to explore several time frames to excite or to propagate the faults and it takes 530.7s to generate all test patterns. The translation of the test patterns into test programs takes 70.0s. Overall, the whole SBST generation time for the scheduling unit is 3151.7s, as shown in Table IV.

In the HATIG method, the leading sequences guide the test generation through each inner segment of the scheduling unit. This allows the HATIG method to effectively test complex sequential units. Table V shows that the test programs generated by the HATIG method achieve 90.0% of fault coverage for the scheduling unit during fault simulation on the whole processor. The test programs take 90184 instructions and 87780 clock cycles.

TABLE V
FAULT COVERAGE WITH DIFFERENT METHODS ON THE SCHEDULING UNIT

	Inst Num.	Gen Time (s)	Exec Time (cycles)	Fault Cov. (%)
Sequential ATPG	--	240h	--	0.2
Random program	90184	4.71	67957	50.7
HATIG	90184	3151.7	87780	90.0

For comparison, we generate random test programs with the same amount of instructions. This approach only achieves 50.7% of fault coverage. Recall that the IVM processor can issue 4 instructions at each clock cycle with 132 types of instructions. As a result, the number of possible input combinations of the scheduling unit is very large, and the random programs only cover a small proportion of that combination. What is worse, the responses of the excited faults do not propagate to the output immediately, and they are easily changed by other instructions. Compared with random programs, the HATIG method uses structural test patterns by applying constrained ATPG on the flattened scheduling unit. This allows the HATIG method to avoid having to test all possible input combinations of the unit under test. Furthermore, observing instructions are inserted just after the generated test instruction sequence, and test responses can be stored into memory at once. As a result, the HATIG method effectively tests the scheduling unit, and achieves about 40% higher fault coverage when compared to randomly generated programs with the same number of instructions.

As shown in Table V, we have also applied a state of the art sequential ATPG on the scheduling unit, and it takes 240h on an 8-cpu computer and only achieves 0.2% of fault coverage. The reason for this is that faults in sequential units often require several clock cycles to be excited or to propagate. In addition, there are many loops in such large circuits and, without proper guidance, the ATPG tool ends up in an endless search among many possible combinations and paths. Compared with the sequential ATPG method, the HATIG method generates leading sequences using bounded model checking in order to ensure that each leading sequence activates a functional path within the sequential unit. Each leading sequence is then imposed on the flattened circuit of the scheduling unit and only then is the ATPG applied to detect the faults on the remaining parts of the unit. In other words, the leading sequences guide ATPG across the inner segments of the scheduling unit. This allows the HATIG method to achieve very high fault coverage.

Finally, it is worth observing that it is not necessary to achieve comparable fault coverage in functional tests as in full-scan tests in the context of a complex microprocessor architecture. Indeed, some faults will never be activated in functional mode. More specifically, there are two types of faults in the scheduling unit. First, the processor structure imposes some constraints on the signals of that unit. For example, as explained in Section III.B, the instruction at the *(head)*$_{th}$ cell cannot be issued to the *SA1* channel. As a result, the faults under that path cannot be tested under normal operations of the processor. Second, the instruction set architecture also imposes some constraints on the signals of the unit under test. For example only the jump/branch instructions are issued to the BR channel. These instructions do not use registers for their second operand, so that the register ready signal for the second operand are always set to be true.

V. CONCLUSIONS

In this work, the scheduling unit in an out-of-order superscalar processor is effectively and efficiently tested using a new method called HATIG. First, the processor is modeled as a global EFSM on which slicing is applied to reduce the size of the resulting representation and to target the scheduling unit. Second, bounded model checking is applied on the resulting EFSM in order to generate leading instruction sequences that activate all functional paths of the scheduling unit. Third, the flattened scheduling unit is constrained by these leading sequences and ATPG is applied on the constrained units in order to detect all functionally testable faults. The obtained test patterns are then turned into test programs using ATIG. The experimental results show that HATIG achieves 90% of fault coverage for the scheduling unit during fault simulation on the whole processor.

REFERENCES

[1] K. Constantinides, T. Austin, “Using Introspective Software-Based Testing for Post-Silicon Debug and Repair”, *DAC’09*, pp.537-542.

[2] L. Chen, S. Dey, “Software-Based Self-Testing Methodology for Processor Cores”, *IEEE Trans on CAD*, Vol. 20, 2001, pp. 369-380.

[3] L. Chen, S. Ravi, A. Raghunathan and S. Dey, “A Scalable Software-Based Self-Test Methodology for Programmable Processors”, *DAC’2003*, pp. 548-553.

[4] R. S. Tupuri and J. A. Abraham, “A Novel Functional Test Generation Method for Processors Using Commercial ATPG,” *ITC’1997*, pp. 743-752.

[5] Charles H.-P. Wen, Li-C Wang, Kwang-Ting Cheng, “Simulation-Based Functional Test Generation for Embedded Processors”, *IEEE Trans on Computers*, Vol. 55, 2006, pp.1335-1343.

[6] Y. Zhang, H. Li, X. Li, “Software-Based Self-Testing of Processors Using Expanded Instructions”, ATS’2010, pp.415-420.

[7] N. Kranitis, A. Paschalis, D. Gizopoulos, G. Xenoulis, “Software-Based Self-Testing of Embedded Processors”, *IEEE Trans on Computers*, Vol. 54, 2005, pp.461-475.

[8] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, S. Ravi, “Systematic Software-Based Self-Test for Pipelined Processors”, *IEEE Trans on VLSI Systems*, Vol. 16, 2008, pp. 1441-1452.

[9] N. Kranitis, A. Merentitis, D. Gizopoulos, “Hybrid-SBST Methodology for Efficient Testing of Processor Cores”, *IEEE Design & Test of Computers*, Vol. 25, 2008, pp. 64-75.

[10] C-H Chen, C-K Wei, T-H Lu, H-W Gao, “Software-Based Self-Testing With Multiple-Level Abstractions for Soft Processor Cores”, *IEEE Trans on VLSI Systems*, Vol. 15, 2007, pp. 505-516.

[11] T-H Lu, C-H. Chen, K-J Lee, “Effective Hybrid Test Program Development”, *IEEE Trans on VLSI Systems*, 2009.

[12] S. Gurumurthy, S. Vasudevan, J.A. Abraham, “Automated Mapping of Pre-Computed Module-Level Test Sequences to Processor Instructions”, *ITC’05*, pp.294-303.

[13] D. Thanh, A. Roychoudhury, T. Mitra, P. Mishra, “Generating Test Program to Cover Pipeline Interactions”, *DAC'2009*, pp.142-147.

[14] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving", Journal of Formal Methods in System Design, Vol.19, 2001, pp.7-34.

[15] http://www.crhc.illinois.edu/ACS/tools/index.html

[16] K.T. Cheng, A.S.Krishnakumar, “Automatic Functional Test Generation Using the Extended Finite State Machine Model”, *DAC’93*, pp.86-91.

[17] A.Y. Duale, M.U. Uyar, “A Method Enabling Feasible Conformance Test Sequence Generation for EFSM Models”, *IEEE Trans. on Computers*, pp.614-627.

[18] M. K. Ganai, and A. Gupta, "Accelerating High-level Bounded Model Checking", *ICCAD'2006*, pp.794-801.

[19] http://w2.cadence.com/webforms/cbl_software/index.aspx