THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, Singapore 117590

TR21/07

Specification-driven Compact Test Suite Generation for Complex Processor Pipelines

Dang T. Thanh, NGA, Abhik ROYCHOUDHURY and Tulika MITRA

November 2007

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Specification-driven Compact Test Suite Generation for Complex Processor Pipelines

Nga Dang T. Thanh
School of Computing
National Univ. of Singapore
dcsdttn@nus.edu.sg

Abhik Roychoudhury
School of Computing
National Univ. of Singapore
abhik@comp.nus.edu.sg

Tulika Mitra
School of Computing
National Univ. of Singapore
tulika@comp.nus.edu.sg

ABSTRACT

Testing of modern-day processors to achieve gate-level coverage is a complex activity. While VLSI testing methods are extremely useful, they are unaware of the micro-architectural features of the processor. Functional validation of a processor design through simulation of a suite of test programs is a common industrial practice. In this paper, we develop a high-level architectural specification driven methodology for systematic test suite generation. Our primary contributions are (1) a fully formal processor pipeline modeling framework based on Communicating Extended Finite State Machines and (2) on-the-fly exploration of the processor model to generate test program witnesses, with an aim to achieve complete state coverage. While we achieve 100% coverage, random test generation manages to cover as low as 10% of the state space with comparable sized test suite. Moreover, we achieve significant reduction in the test-suite size compared to previously studied formal approaches that rely on querying an external model checker for test generation.

1. INTRODUCTION

The increasing complexity of embedded systems (e.g., in the consumer electronics domain) is driving the dominance of high-performance processors as the design choice. The higher computational requirement of these embedded systems generally translates to the inclusion of complex but performance enhancing features, such as cache and out-of-order pipelines, in the processor micro-architecture. However, the non-trivial interactions among these performance enhancing features are the major sources of errors in the processor implementation. A significant portion of the processor design effort is thus spent in validation.

A common industrial practice in processor validation is to generate billions of random test programs at the instruction-set architecture (ISA) level. As such a test program generation process is micro-architecture agnostic, it fails to exercise the subtle micro-architectural artifacts. As an example, it is extremely unlikely that a randomly generated test program can exercise the following interaction: "In a clock cycle, an `ADD` instruction is issued to the ALU unit and a `STORE` instruction is stalled due to Read After Write (RAW) dependency with the `ADD` instruction". However, this is precisely the kind of interaction among the pipeline components that causes errors in the design.

In this work, we develop a formal framework for micro-architecture aware test program generation in processor validation. There are three major challenges in developing such a framework. First, such an automated test program generator would require a formal specification of the micro-architectural details. A Register Transfer Level (RTL) model, written in Verilog or VHDL, provides complete and accurate information about the processor implementation. However, the abstraction of micro-architectural components and their interactions are lost at RTL level. In this work, we develop a formal modeling framework based on an existing Architecture Description Language (ADL) — the Operation State Machine (OSM) model [10]. Our main contribution here is in building *a fully formal modeling framework targeted towards test generation*. Our model does not contain any custom code to describe micro-architectural behavior. Thus, it is amenable to formal analysis based test generation.

The second and central challenge is identifying *all* possible pipeline interactions to meet a certain coverage metric. Existing research on coverage driven test generation do not study this issue in details (instead they elaborate on the test program generation for a *particular* interaction [4, 8]). We develop an *on-the-fly reachability analysis method* that explores the possible pipeline states by following the evolutions of the processor pipeline. The crucial point is that (a) the reachability analysis is done at a high-level of modeling abstraction, and (b) the state space is constructed *and* traversed on-the-fly, thereby preventing blow-up.

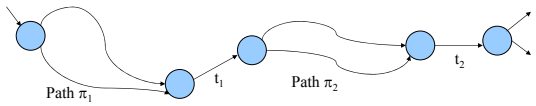

Figure 1: Test suite reduction in on-the-fly exploration.

The final challenge is the construction of a test program corresponding to each pipeline interaction identified in the previous step. Existing works generate one test program corresponding to each interaction. However it is possible for a single test program to cover more than one interaction. Figure 1 shows a schematic fragment of the state space for a processor pipeline. Transitions t_1 and t_2 are global transi-

tions possibly denoting certain interactions among pipeline components. If we generate test programs for each interaction, we will generate separate test programs for t_1 and t_2. However, when we construct/explore the state space on-the-fly we realize that the same test program (say a sequence of instructions which drives the processor along path π_1 followed by t_1, followed by path π_2, followed by t_2) can witness both t_1 and t_2. In other words, we exploit the fact that one pipeline interaction often leads to another to achieve *reduction* in the suite of test programs generated.

Organization of the paper. The rest of the paper is organized as follows. In the next section, we discuss related work. Section 3 presents our pipeline modeling. Section 4 describes the test generation method, while Section 5 discusses our experiments. Section 6 concludes the paper.

2. RELATED WORK

Test program generation for processor validation is a well-studied topic. The Genesys tool developed by IBM [1] performs pseudo-random test program generation based on an architecture/testing knowledge base. The primary focus of this work is to test the instruction set architecture.

In recent years, the focus in this area has been to generate test patterns for processor micro-architecture. Diep and Shen [3] enumerate the possible pipeline hazards, given a low-level processor specification; for each of these hazards a test program is then automatically constructed. Zhu et al. [15] take a different approach, where they synthesize directed tests from a high-level description of a processor. Their focus is to test the bypass paths in a pipelined processor. Mishra and Dutt [9] exploit test program templates for canonical events like pipeline hazards and exceptions. These test program templates can then be combined to construct test programs for more complex scenarios.

Among the formal methods driven approaches to test generation, one of the earliest works was by Ho et al. [6]. This work synthesizes FSM model from a HDL specification of the processor. One of the major difficulties here is the state-space explosion as the FSM model is constructed and stored prior to state space exploration. In subsequent works, Ur and Yadin [14] suggest using transition coverage of the state space (corresponding to the processor's FSM) to generate test programs. As pointed out in Figure 1, such a method is oblivious of the structure of the state space and may generate many redundant test programs. Moreover, unlike these works, we also validate pipeline interactions with global micro-architectural features such as cache.

In existing works on pipeline validation, Geist et al. [4] and Ko et al. [8] have used model checking tools for test program generation. The aim is to achieve a coverage of the state space by covering enough temporal properties (against which the model checker is run). The witness test program for each property is generated automatically by the model checker. However, the properties (typically light-weight temporal properties like invariants) need to be provided, and a large number of properties are required for realistic processors. The key idea in our approach is to generate the entire test suite through on-the-fly exploration of the global state space, rather then querying the global state space several times. As we do not query the global state space for property checking, we are also freed from the burden of providing the properties to the checker.

Finally, note that our work uses formal modeling and *automated* reachability analysis methods for testing processor pipelines. Thus, our aim is somewhat *different* from works on formal verification of complex processor pipelines (e.g., see [7, 12]). These approaches use a combination of model-checking and theorem-proving, and the primary focus is on enhancing automation in the formal reasoning.

3. ARCHITECTURE MODELING

Let us first present our processor micro-architecture before delving into the details of the modeling.

3.1 Processor Micro-Architecture

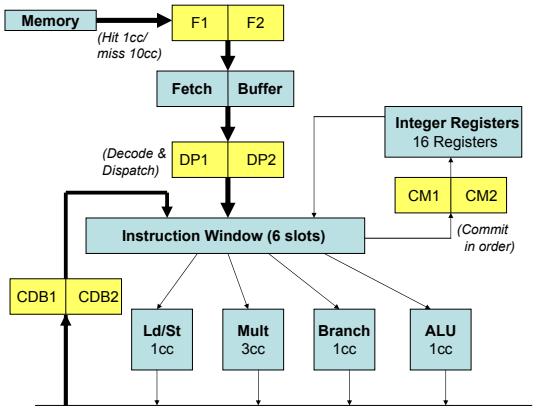

Figure 2: Processor Pipeline Structure

Our formalism is powerful enough to model different micro-architectural features for test program generation purposes. For illustration purposes, we use a 2-*way superscalar, out-of-order execution pipeline* shown in Figure 2. It is a simplified version of the SimpleScalar `sim-outorder` simulator processor model [2], which in turn is based on [13]. The pipeline consists of five stages: Instruction Fetch (`IF`), Instruction Decode and Dispatch (`ID`), Execute (`EX`), Write Back (`WB`) and Commit (`CM`). The `IF` stage fetches at most two instruction per cycle from the instruction cache. We assume 10 cycle cache miss penalty and perfect branch prediction. The `ID` stage decodes and dispatches at most two instructions per cycle to the instruction window in program order.

The instruction window forms the core of the out-of-order execution. It combines the reservation stations and the reorder buffer into one single structure and manages all the instructions in flight. An instruction remains in the instruction window from the point it is dispatched to the point it is committed. At most two instructions are issued per cycle from the instruction window to the required functional units. An instruction is issued if its operands are ready and the required functional unit is free, i.e., the instruction can be issued out of program order. If there is contention among the ready instructions for either functional unit or issue slot, the program order determines the priority. Our example pipeline has the following functional units: a single cycle ALU unit, a multi-cycle integer multiplier unit, a load-store unit and a branch unit.

Once an instruction completes execution in the functional unit, it competes for the common data bus (CDB) to write back the result and wake-up the dependent instructions in

the instruction window. This is known as the WB stage. In every cycle, two of the oldest instructions that have completed the WB stage, copy their results to the register file and free the instruction window entry. Note that the commit stage (CM) proceeds in program order.

3.2 Communicating EFSM Formalism

To construct the formal pipeline model, we draw upon and augment the central ideas in the Operation State Machine (OSM) model for architectural description languages [10]. The OSM models a processor at two levels — the operation level and the hardware level. At the operation level, OSM describes the movement of the instructions across the pipeline stages as Extended Finite State Machines (EFSM). For the hardware level, the OSM models the various hardware resources (e.g., fetch buffer, instruction window, functional units etc.) as "token managers". The operation level EFSMs progress from one state to another by acquiring/releasing tokens from/to the token managers of the resources. Currently, in the OSM model, the token management policies for the hardware resources are specified as hand-crafted executable code. The communication between the operation level EFSMs and the hardware resources are handled by executing the code corresponding to the resource managers. In other words, there is no formal modeling of the token management policies in the OSM model. We extend the OSM model to formally specify both the operation level and the hardware level processor behavior as follows.

An EFSM is a finite-state machine with variables; in our pipeline modeling we only consider finite domain variables. Each transition in an EFSM involves (a) a source and a destination state, (b) a guard on the variables (which serves as a pre-condition for the enabledness of the transition), and an action (involving assignments to the variables).

We model a processor pipeline as a collection of communicating EFSMs, with say one EFSM for the instruction window, one for the integer unit and so on. To achieve this step, we make a small change in the EFSM definition. Given a collection of EFSMs $M_1, \ldots, M_n$, any transition in an EFSM M_i contains: (a) a source state, drawn from the set of states in M_i, (b) a destination state, drawn from the set of states in M_i, (c) a guard, which is a conjunction of *guard methods* (each of these methods is a guard method for some EFSM M_j where j is not necessarily equal to i), and (d) an action, which is a sequence of *action methods* (each of these methods is an action method for some EFSM M_j where j is not necessarily equal to i). In other words, the enabledness of a transition in EFSM M_i in our pipeline model may depend not only on the variables of M_i but also on the variables of the other EFSMs. Also, the effect of a transition in EFSM M_i not only changes the variables in M_i but also the variables in other EFSMs.

It should be noted that even though we use guard and action methods for the sake of comfortable modeling, this *does not introduce any custom hand-crafted code in our models*. Any guard method in our model is a condition that is evaluated without side-effects to true or false. Any action method in our model is a collection of assignments that are executed simultaneously.

The communication supported in our model is a restricted version of the communication mechanism in Statecharts [5] or UML State Diagrams. In Statecharts, an action a by a state machine (denoting a system component) is broadcast to other state machines, which may then change states (provided there is an enabled transition on a from their current states). In our communicating EFSMs, there is *no broadcast communication*. In the guard/action part of a transition t in EFSM M_i (denoting a system component), we explicitly mention the other system components whose guard/action methods we execute as part of t's execution. Due to the absence of broadcast, a single step in our model always terminates (which is *not* the case in Statecharts).

What constitutes a single step in our model? Given a collection of communicating EFSMs $M_1, \ldots, M_n$, execution of a transition t in EFSM M_i in the collection involves the following (all of which are done *atomically*).

1. Evaluate each of the guard methods in t and check that they all return true.
2. Check that each of the action methods of t can be executed. This is done as follows. If an action method a_t involves assignments to variables in M_i, we say that a_t can be executed. If an action method a_t involves assignments to variables of some other EFSM M_j we check whether in the current state of M_j there exists an enabled transition t' such that (a) the guard of t' is true, and (ii) the action of t' is the action method a_t.
3. Execute all the action methods of t, thereby changing the state of M_i as well as those of all M_j whose action methods appear in t.

3.3 Modeling with Communicating EFSMs

Our pipeline modeling is a collection of communicating EFSMs. This collection can be partitioned into two groups. The first group contains EFSMs that show the advancement of an instruction across the pipeline stages. For a pipeline with maximum N in-flight instructions ($N = 8$ in Figure 2), we will have N such EFSMs executing concurrently. In the second group, we have EFSMs modeling the individual pipeline resources such as instruction cache, fetch buffer, integer unit, etc. The instructions progress by acquiring these resources. *We also model the instruction cache as a separate resource that needs to be acquired for an instruction to be fetched (this models the pipeline-cache interaction).* In our cache modeling we ensure that when a memory block is accessed, only the first instruction in the block can be cache hit/miss, but the others will result in cache hits. In summary, we model both the operation level state machines as well as the resource managers as communicating EFSMs.

In Figure 3, we show the guard/action of a sample transition in the operation level EFSM. The guard/actions refer to the branch unit and Common Data Bus or CDB (see Figure 2) meaning that this transition involves a communication between an operation level EFSM (representing an instruction residing in the pipeline), the branch unit resource EFSM and the CDB resource EFSM. Figure 3 also shows a simplified version of the branch unit resource EFSM. It has two variables: token (representing whether a branch is currently executing inside the unit) and time (representing the number of clock cycles since the currently executing branch started execution). When the operation level EFSM makes the transition from "Branch" state to "WB" state (signifying end of the EX stage of a branch instruction), it checks whether the branch unit is ready to finish by comparing the elapsed execution time with branch latency, which is a pre-defined constant (*latency* in Figure 3). This check is captured as a guard method *canRelease*.

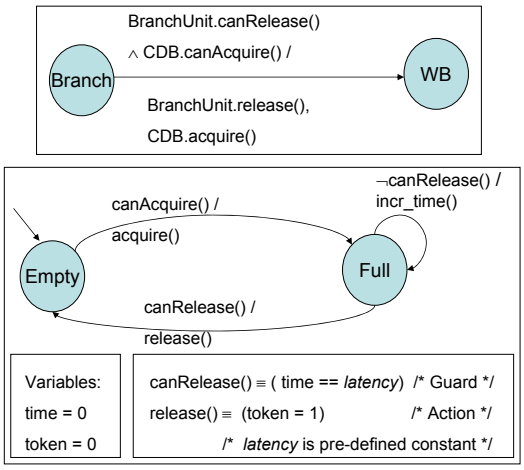

Figure 3: A simplified sample transition in the Operation level EFSM (shown with guards/actions) and a simplified Resource EFSM for the Branch Unit.

Finally, we should note that Figure 3 shows a simplified version of the operation EFSM transition which takes an instruction from Branch Unit to WB stage. In actual modeling, we also consider the scheduling over the CDB in the transition guard. Even if the branch unit can be released and the CDB is free, we need to enforce the writeback bandwidth *bw* (maximum number of instructions that can go on the CDB in a cycle). This is handled in our model by encoding the scheduling policy within the EFSM transition guards — an instruction transmits on the CDB if there are less than *bw* transitions "before" it (in terms of program order) that are also ready to transmit on the CDB. In other words, we let the operation EFSMs contain guards on the states of other operation EFSMs.

4. TEST PROGRAM GENERATION

We now describe test program generation from our Communicating EFSM based pipeline model. Our primary goal is specification driven generation of a compact suite of test programs based on certain coverage metric. Previous work on specification-driven test generation mostly consider the pipeline structure to define the coverage metric. For example, the coverage metric can be defined to ensure that all the components in the structural specification of the pipeline are exercised. However, given only the structural specification of the pipeline, one cannot define *behavioral* scenarios such as — in one clock cycle, one instruction (say $I1$) is issued to the ALU unit and another store instruction (say $I2$) is stalled in the instruction window as $I2$ is dependent on $I1$. Our EFSM-based pipeline specification can capture both the structure, the behavior, and the interaction among the components. Thus our coverage metric aims to exercise corner cases in the pipeline behavior as well.

So what is our pipeline coverage metric? Given the pipeline model as a collection of communicating EFSMs $M_1, \ldots, M_n$ we can compose the component EFSMs to construct a global FSM. Any state of the global FSM consists of a local state and a variable valuation for each of the component EFSMs. Then our pipeline coverage metric is defined as covering all the states in the global FSM. We aim to generate a suite of test programs that drive the processor to *all reachable states* in the global FSM (which is obtained by composing the communicating EFSMs).

4.1 Illustrative Example

We first introduce the test generation algorithm through a simple example. Suppose we have reached the global FSM state s starting with the empty pipeline state. The state s contains only two instructions in the instruction window; there are no other instructions in flight. For simplicity of exposition, let us denote the two instructions with identifiers `I1, I2`. Instruction `I1` appears before `I2` in the instruction window, which implies `I1` should appear before `I2` in program order. At this point, we construct a test program that only contains placeholders for the two instructions as follows

```
I1: <opcode> <dest> <src1> <src2>
I2: <opcode> <dest> <src1> <src2>
```

Now we need to construct all global FSM states reachable from the state s. Note that both `I1` and `I2` can proceed to any functional unit according to the operation level EFSMs. We have to construct global FSM states corresponding to each of these possibilities. Let us choose the scenario where `I1` is assigned to the ALU unit and `I2` is assigned to the MULT unit. Our test program template is now refined to

```
I1: ADD <dest> <src1> <src2>
I2: MUL <dest> <src1> <src2>
```

However, the operation level EFSMs corresponding to `I1, I2` can proceed only if their guards are satisfied. Thus our chosen scenario leads to multiple global FSM states. The guards for issuing instructions from the instruction window are: [G1] the functional unit should be free, [G2] an issue slot should be available, and [G3] the instruction should be ready. As `I1, I2` are targeting two different functional units and there are no other instructions in flight, [G1] is trivially satisfied. Similarly, [G2] is satisfied as we have 2-way issue processor. But [G3] leads to further refinement of the scenario and hence different global FSM states.

As there is no other in flight instruction, the state where `I1` is not ready is not reachable from state s. There is no assignment of the variable values that can make [G3] false for `I1`. So we can choose arbitrary register identifiers for `I1`.

```
I1: ADD r0, r1, r2
I2: MUL <dest> <src1> <src2>
```

Now [G3] can be either true or false for `I2`. So we have one complete global FSM state corresponding to [G3] being true for `I2`. The test program template is now complete.

```
I1: ADD r0, r1, r2
I2: MUL r3, r4, r5
```

The other global FSM states corresponds to [G3] as false. The constraint [G3] is false can be satisfied if one or both the source registers of `I2` have dependency with the destination register of `I1`. So we get three more reachable FSM states from s (these will lead to three other test programs apart from the one shown above).

Note that the program with two instructions we generated above uses registers that have not been pre-loaded. So, to make the test program *executable*, we need to introduce instructions in the beginning to load registers from memory. Furthermore, we need NOP instructions to ensure that the pipeline is empty when the two instructions in our synthesized program start executing.

Algorithm 1 Constructing the suite of test programs

Input — $M_1, \ldots, M_n$: the operation and resource EFSMs for processor pipeline;
Output — Test program suite;

$s_0 \leftarrow$ Initial state of $M_1, \ldots, M_n$;
$testPgm(s_0) \leftarrow null$; $Visited \leftarrow \{s_0\}$;
call $TGen(s_0, M_1, \ldots, M_n)$;

Algorithm 2 Recursive procedure $TGen$ for constructing test programs from a pipeline state

Input — $M_1, \ldots, M_n$: the operation and resource EFSMs for processor pipeline, s: a global pipeline state
Output — Test programs starting from state s;

for all operation EFSM step $s \rightarrow s'$ **do**
/* these steps involve resource EFSMs as well */
if $s' \in Visited$ **then**
output $testPgm(s)$;
else
$Visited \leftarrow Visited \cup \{s'\}$;
$testPgm(s') \leftarrow Refine(s, s', testPgm(s))$;
call $TGen(s', M_1, \ldots, M_n)$;
end if
end for
for all resource EFSM step $s \rightarrow s'$ **do**
/* these steps do not involve operation EFSMs */
if $s' \in Visited$ **then**
output $testPgm(s)$;
else
$Visited \leftarrow Visited \cup \{s'\}$; $testPgm(s') \leftarrow testPgm(s)$;
call $TGen(s', M_1, \ldots, M_n)$;
end if
end for

4.2 Algorithm

In our approach, (i) construction of the global FSM, (ii) traversal of the paths of the global FSM, and (iii) test program generation of the individual paths — all of these are fused into a single step. We do not construct and store the global FSM in advance. Instead, we construct it *on-the-fly*, as we are traversing the paths.

Our test program generation scheme is shown in Algorithms 1 and 2. The set of visited states in the global state space is maintained via the $Visited$ set (we implement it as a hashtable for efficient access). The path $\pi = s_0 \rightarrow s_1 \ldots \rightarrow s_m$ from the initial state s_0 to the current state s_m is maintained implicitly via the procedure invocation stack. For each state s_i $(0 \leq i \leq m)$ in π, we maintain $testPgm(s_i)$ which is the sequence of instructions driving execution along the path π up to state s_i. The operation $Refine(s, s', testPgm(s))$ refines the test program for state s depending on the operation EFSM transition $s \rightarrow s'$.

While exploring the global FSM corresponding to a processor model we always maintain a path π from the initial state to the "current" state. Each state along this path π is associated with a "partially instantiated" test program where the opcodes/operands of some of the program instructions may be uninstantiated (this was shown in our illustrative example, refer Section 4.1). These uninstantiated opcodes/operands are *lazily* instantiated as the instructions proceed along the pipeline stages. Having partially instantiated test programs allows us to maintain *one* test program for each state s, rather than maintaining a large set of concrete test programs (each of which leads to s). If a generated test program (*i.e.*, output in Algorithm 2) contains any partially instantiated instruction, it means that any uninstantiated opcode/operands can be instantiated arbitrarily.

Inherent Reduction in Generated Test Suite. Our on-the-fly exploration of the state space helps reducing the total number of test programs as shown in Figure 1. If one uses an external model checker as a query engine [14, 8], we will construct two temporal logic properties to check for the (un)reachability of t_1 and t_2. By checking for the reachability of transitions t_1 and t_2 separately (as would be done in separate runs of model checking) — we would unnecessarily generate separate witness paths for transitions t_1, t_2 without realizing that transition t_1 *leads to* transition t_2. This redundancy will be avoided in our approach, leading to reduction in test suite size, as shown by our experiments.

5. EXPERIMENTAL EVALUATION

We present an evaluation of our automated formal specification based test program generation framework in this section. We are particularly interested in the following metrics: (1) state space coverage of a random test generation method in comparison to our 100% guaranteed coverage test generation framework, (2) test suite reduction as we aim to cover multiple states through one test program, and (3) the scalability of our test generation algorithm.

5.1 Test Program Generation Framework

We build our test generation framework on top of the SimpleScalar architectural simulation toolset [2]. The SimpleScalar instruction set architecture (ISA) is a superset of MIPS ISA. As mentioned earlier, the out-of-order superscalar processor pipeline that we model in this work is a simplified version of the SimpleScalar `sim-outorder` simulator processor model [2], which in turn is based on [13].

Our test suite generation framework consists of three main components: *Formal Specification* of ISA and micro architecture for the target processor, *State space exploration*, and target ISA-compatible executable *test program construction*. We specify the SimpleScalar ISA in MESCAL Architecture Description Language (MADL) [11]. This includes static properties of the instructions such as operation semantics, assembly syntax, and binary encodings. The current version of MADL can only support the operation level of the OSM model. The hardware level, including the token managers and the structural hardware components, is not part of MADL (as the OSM model itself lacks formalization of the internal behaviors of the token managers). Instead, we rely on Extensible Markup Language (XML) to specify the operation level and hardware level communicating EFSMs in our specification formalism. These communicating EFSMs correspond to the SimpleScalar out-of-order processor pipeline shown in Figure 2.

The parser in our test generation framework reads in the ISA (specified in MADL) and the micro-architecture (specified in XML), and creates a mapping between the two. Next, the state space exploration module is invoked to traverse and identify all reachable states in the global FSM (which is obtained by composing the communicating EFSMs).

Recall that a partially instantiated test program $\mathcal{P}$ is maintained as we traverse a path through the global FSM state space. However, the instructions in $\mathcal{P}$ only contain the opcode and register operands information. The test program construction (1) creates assembly language instructions for $\mathcal{P}$ by consulting the target ISA specification, and (2) prepends additional instructions in the test program to load legal data from memory to the live registers in $\mathcal{P}$. Fi-

Processor Config.	# States $\lvert\mathcal{S}\rvert$	# Tests. $\lvert\mathcal{T}\rvert$	Reduction $\frac{\lvert\mathcal{S}\rvert-\lvert\mathcal{T}\rvert}{\lvert\mathcal{S}\rvert}$
In-order, superscalarity=1	14,512	6835	53%
In-order, superscalarity=2	33,136	20,175	39%
Out-of-order, superscalarity=1	135,765	71,715	48%
Out-of-order, superscalarity=2	331,071	220,878	34%

Table 1: Compactness of our generated test suite for different processor configurations. The length of test program is limited to the maximum number of in-flight instructions in the pipeline (in this case 8).

Test Pgm. Length	Coverage (Ours)	Coverage (Random)	Time (Ours)
2	100%	66%	0s
3	100%	48%	1s
4	100%	31%	5s
5	100%	20%	28s
6	100%	10%	3m 50s
7	100%	9.4%	10m 3s
8	100%	10%	18m 48s

Table 2: (i) Quality of randomly generated test suites, and (ii) runtime of our approach, on an out-of-order 2-way superscalar processor with cache.

nally the assembly level test program is executed on the SimpleScalar simulator to ensure correctness.

5.2 Results

Let us now proceed to present the experimental results. Our formal specification-based test generation framework is guaranteed to achieve 100% test coverage. Let $\mathcal{S}$ be the set of reachable states in the global FSM state space. Clearly, our method explores all the states in $\mathcal{S}$. Moreover, as we generate a test program corresponding to each path in the state space (as opposed to one test program per state), we achieve significant reduction in the size of the test suite. In other words, one test program in our method can possibly cover multiple states. Let $\mathcal{T}$ be the set of test programs generated by our approach. Then $(\frac{|\mathcal{S}|-|\mathcal{T}|}{|\mathcal{S}|} \times 100)$ is the test suite size reduction. Table 1 shows this reduction ranges from 34–53% for different processor configurations — in-order (or not), superscalar (or not). We have bounded the length of a test program to N where N is the maximum number of in flight instructions in any clock cycle ($N = 8$ in our experiment). Clearly, introducing more than N instructions in a test program cannot exercise any new interaction among the instructions. Recall that the test program construction phase introduces additional load/NOP instructions at the beginning of a test program. These additional instructions are excluded while counting the "length" of a test program.

Next, we compare random test generation with our test generation method. For this comparison, we choose the most complex among the four processor configurations — out-of-order and 2-way superscalar. Let $|\mathcal{T}_n|$ ($|\mathcal{S}_n|$) be the number of test programs (reachable states) generated by our approach for a bound n on the length of each program. Then we randomly generate $|\mathcal{T}_n|$ test programs. We allow these test programs to proceed through our global FSM and identify the set of reachable states $\mathcal{R}_n$ visited by the random test programs. Then $(\frac{|\mathcal{R}_n|}{|\mathcal{S}_n|} \times 100)$ shows the coverage percentage of the random test generation approach. Our method is guaranteed to achieve 100% coverage. The coverage percentage of random method diminishes quickly to as low as 9% with increasing values of n as shown in Table 2. The runtime of our on-the-fly exploration approach is less than 19 minutes for covering all the 331,071 states of a complex out-of-order 2-way superscalar processor with a direct-mapped instruction cache (32 bytes per cache line). All experiments were conducted on a 2.6 GHz Intel Pentium IV machine with 1 GB of main memory.

6. DISCUSSION

In this paper, we have presented a fully formal processor modeling framework inspired by recent work on Architecture Description Languages. We have used our processor models for systematic test generation to cover all possible pipeline states for out-of-order superscalar processors with instruction cache.

In future, we will extend our approach to model the pipeline interaction with data cache and branch prediction. Moreover, we plan to enhance the efficiency of our approach by optimizing the data structures and the implementation of our test generation method.

7. REFERENCES

[1] A. Aharon, D. Goodman, M. Levinger, C. Metzger, M. Molco, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *DAC*, 1995.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[3] T.A. Diep and J.P. Shen. Systematic validation of pipeline interlock for superscalar microarchitectures. In *FTCS*, 1995.

[4] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. Coverage-directed test generation using symbolic techniques. In *FMCAD*, 1996.

[5] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[6] R.C. Ho, C. Han Yang, M.A. Horowitz, and D.L. Dill. Architecture validation for processors. In *ISCA*, 1995.

[7] R. Jhala and K.L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, 2001.

[8] H-M. Ko and P. Mishra. Functional test generation using property decompositions for validation of pipelined processors. In *DATE*, 2006.

[9] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *DATE*, 2004.

[10] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *DATE*, 2003.

[11] W. Qin, S. Rajagopalan, and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES*, 2004.

[12] S. Ray and W.A. Hunt. Deductive verification of pipelined machines using first-order quantification. In *CAV*, 2004.

[13] G. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3), 1990.

[14] S. Ur and Y. Yadin. Micro-arachitecture coverage directed generation of test programs. In *DAC*, 1999.

[15] Q. Zhu, A. Shrivastava, and N. Dutt. Functional and timing validation of partially bypassed processor pipelines. In *DATE*, 2007.