# Generating Test Programs to Cover Pipeline Interactions

Thanh Nga Dang Abhik Roychoudhury Tulika Mitra
National University of Singapore
{dangthit,abhik,tulika}@comp.nus.edu.sg

Prabhat Mishra
Univ. of Florida, Gainesville
prabhat@cise.ufl.edu

## ABSTRACT

Functional validation of a processor design through execution of a suite of test programs is common industrial practice. In this paper, we develop a high-level architectural specification driven methodology for systematic test-suite generation. Our primary contribution is an automated test-suite generation methodology that covers all possible processor pipeline interactions. To accomplish this automation, we (1) develop a fully formal processor model based on communicating extended finite state machines, and (2) traverse the processor model for on-the-fly generation of short test programs covering all reachable states and transitions. Our test generation method achieves *several orders of magnitude* reduction in test-suite size compared to the previously proposed formal approaches for test generation, leading to drastic reduction in validation effort.

## Categories and Subject Descriptors

B.1.3 [**CONTROL STRUCTURES AND MICROPROGRAMMING**]: Control Structure Reliability, Testing, and Fault-Tolerance—*Test Generation*

## General Terms

Algorithms,Verification

## Keywords

Pipelines, Automated test generation, State space exploration.

## 1. INTRODUCTION

The increasing complexity of embedded systems is driving the dominance of high-performance processors as the design choice. The higher computational requirement of these systems generally translates to the inclusion of complex but performance boosting features, such as caches and pipelines, in the processor architecture. However, the non-trivial interactions among these performance enhancing features are 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. Such a test program generation process is micro-architecture agnostic, and it fails to exercise the subtle micro-architectural artifacts.


*DAC 2009,* July 26 - 31, 2009, San Francisco, California, USA.


In this work, we develop a fully formal framework for micro-architecture aware test program generation. There are three major challenges in developing such a framework. First, an automated test program generator would require a formal specification of the architectural details. We note that architectural components and their interactions are lost at a Register Transfer Level (RTL) model. Instead, we develop a formal processor model based on communicating extended finite state machines. Our model is an extension of the Operation State Machine (OSM) model [8] proposed in the context of MESCAL Architecture Description Language (MADL) [9].

The second and central contribution is the generation of a test suite that covers *all possible pipeline interactions*. In its full generality, a pipeline interaction can be viewed simply as a transition in the global state space of the pipeline. Such a global transition may involve several pipeline resources being acquired or released by multiple in-flight instructions. Existing research on coverage-driven test generation for embedded processor pipelines [5, 6] take a restricted view of pipeline interactions — they typically enumerate and cover interactions among $k$ resources for some small value of $k$ (say $1 \leq k \leq 3$). Clearly, we cannot artificially restrict the number of participating resources in a global transition to be a small number. Moreover, even if the test generation method is scaled to include interactions among large number of resources, *it still fails to cover all pipeline interactions*. This is because a single global transition may also involve interaction among *multiple* instructions in the pipeline for the same resource rather than interaction among multiple resources (e.g., multiple instructions trying to acquire the same functional unit in a single clock cycle). The resource-centric coverage model does not capture such pipeline interactions. To achieve complete coverage of all possible pipeline interactions, both inter-resource interactions and inter-instruction interactions have to be accounted for. We construct an efficient automated method that generates the entire test-suite via a *single on-the-fly* exploration of the global pipeline state space to cover all the transitions and thereby cover all possible pipeline interactions.

Our state space explorer produces paths in the global state space leading to different pipeline interactions. Our last and final task is to generate test programs (sequence of instructions) that actually exercise all these paths. Note that a single test program can potentially cover several pipeline interactions. Suppose transitions $t_1$ and $t_2$ are global transitions denoting certain interactions among pipeline components, and $t_1$ leads to $t_2$. When we construct/explore the state space, we use the same test program to cover both $t_1$ and $t_2$. This reduces the test-suite size by *more than two orders of magnitude* compared to existing works [6].

In summary, we develop a fully formal processor model, traverse the model to cover all possible pipeline interactions, and generate a drastically reduced test suite covering all pipeline interactions.

## 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 with primary focus on testing the ISA. Subsequently, the focus has been on generating test patterns for micro-architecture. Diep and Shen [4] 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. [11] synthesize directed tests from a high-level description to test the bypass paths in a pipelined processor. Mishra and Dutt [7] exploit test program templates for canonical events like pipeline hazards and exceptions. However the template generation is not automated, and this has to be done *manually* by the designer.

Among the formal methods driven approaches to test generation, Ur and Yadin [10] suggest coverage directed test generation. However, they model a very small portion of the processor (only handling of arithmetic instructions) at a low-level (akin to state machine like modeling as in the SMV model checker). It is not at all clear how such an approach will scale up to real-life processors. Moreover, instead of covering all global states/transitions, their test generation covers selected state variables (selected *manually*).

Geist et al. [5] and Koo et al. [6] have used model checkers to generate test programs for processor pipelines. The aim is to achieve a coverage of the state space by covering enough temporal properties. The witness test program for each property is generated automatically by the model checker. However, the designer has to provide the large number of properties required for realistic processors. Apart from automation of the entire process, our key contribution is that we generate the entire test suite through *one* exploration of the global state space. This is in contrast to approaches like [6], which query an external model checker *millions* of times.

## 3. ARCHITECTURE MODELING

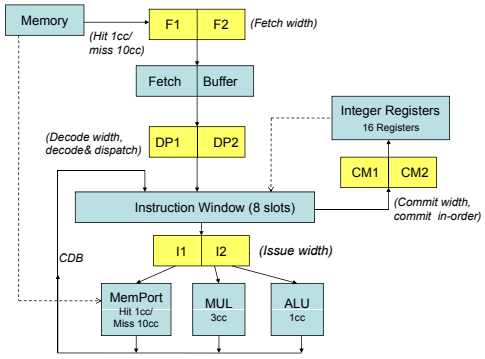


**Figure 1: Processor Pipeline Structure**

Our formalism models various micro-architectural features for test program generation purposes. For illustration, we will use a 2-*way superscalar, out-of-order execution pipeline* shown in Figure 1. The pipeline consists of the following stages: Instruction Fetch, Instruction Decode and Dispatch, Issue, Execute, Write Back, and Commit. We assume 10 cycle cache miss penalty.

To construct the formal pipeline model, we augment the central ideas in the Operation State Machine (OSM) model [8]. The OSM models a processor at two levels — the operation level and the hardware level. At the operation level, the OSM describes the movement of the instructions across the pipeline stages as Extended Finite State Machines (EFSM) [3]. 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 (1) a source and a destination state, (2) a guard on the variables (which serves as a pre-condition for the enabledness of the transition), and (3) an action (involving assignments to the variables).

At the hardware level, the OSM models various hardware resources (e.g., instruction window, functional units, etc.) as "token managers". The operation EFSMs progress from one state to another by acquiring/releasing tokens from/to the token managers of the resources. Currently, in the OSM model [8], the communication among operation EFSMs and hardware resources are handled by executing some code corresponding to the token managers. We extend OSMs to develop a fully formal processor model as follows.

The processor pipeline model consists of a collection of EFSMs. This collection can be partitioned into two groups. The first group contains **operation EFSMs** that show the advancement of instructions across the pipeline stages. For a pipeline with maximum $N$ in-flight instructions ($N = 10$ in Figure 1), we have $N$ such EFSMs executing concurrently. In the second group, we have **resource EFSMs** modeling the individual pipeline resources. The instructions progress by acquiring these resources. We also model the instruction/data cache as separate resources that needs to be acquired for an instruction/data to be fetched. This models the interaction between caches and pipeline. In our cache modeling, we ensure that when a memory block is accessed, only the first instruction/data in the block can be cache hit/miss; the remaining accesses in the block result in cache hits. Exceptions are modeled by introducing additional transitions in the corresponding EFSMs.

The pipeline is modeled as a concurrent composition of operation EFSMs and resource EFSMs. These EFSMs communicate among themselves, in particular, an operation EFSM needs to communicate with several resource EFSMs. The communication primitive needs to capture interactions among more than two components as an operation EFSM may need to communicate with *several* pipeline resources to move an instruction from one stage to another. For example, to issue an instruction to the `ALU`, the guard in the `Issue` $\rightarrow$ `ALU` transition (Fig. 2(a)) at the operation level EFSM should check for the availability of the ALU and the source registers. Thus, we cannot use a simple send-receive rendezvous communication involving two components. Moreover, a transition in an EFSM $M_i$ (representing advancement of an instruction in the pipeline) may refer to variables of other EFSMs $M_j$ (the resources).

We propose **communicating EFSMs** for this purpose where the guard and the action in an EFSM transition may involve guards and actions of other EFSMs. Formally, given a collection of communicating EFSMs $M_1, \ldots, M_n$, we define any transition in $M_i$ with: (1) a source state, drawn from the set of states in $M_i$, (2) a destination state, drawn from the set of states in $M_i$, (3) 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 (4) 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 $i$). 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. In other words, the ownership of a variable by an EFSM is for the convenience of the designer. Strictly speaking, these variables function as shared variables where a variable in $M_j$ can be assigned as an effect of a transition in $M_i$.

Given a collection of communicating EFSMs $M_1, \ldots, M_n$, the execution of a transition $t$ in $M_i$ performs the following atomically.

1. Check that all the guard methods in $t$ are true.

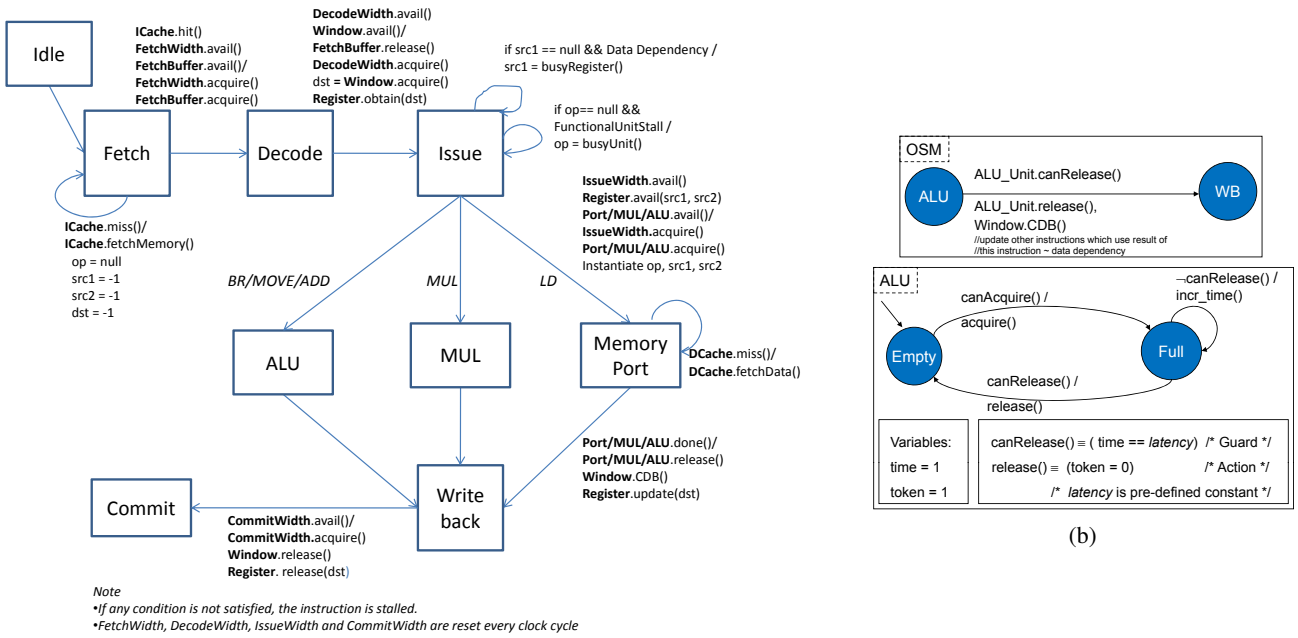


**Figure 2: (a) Operation EFSM for the pipeline, (b) A simplified sample transition in the Operation EFSM (shown with guards/actions) and a simplified Resource EFSM for the ALU Unit.**

2. Check that all the action methods of $t$ can be executed. If an action method $a_t$ involves assignments to variables in $M_i$, $a_t$ can be executed. If $a_t$ involves assignments to variables of another EFSM $M_j$, we check whether in the current state of $M_j$ there exists an enabled transition $t'$ such that (i) the guards of $t'$ are true, and (ii) the action method $a_t$ is the action corresponding to $t'$.

3. Execute all the action methods of $t$, changing the state of $M_i$ as well as those of all $M_j$ whose action methods appear in $t$.

The operation EFSM for our processor pipeline appears in Figure 2(a). In Figure 2(b), we show the guard and action of a sample transition in the operation level EFSM. This transition involves a communication between an operation EFSM and the resource EFSMs for the ALU unit and the instruction Window. Figure 2(b) also shows a simplified version of the ALU EFSM. It has two variables: token (representing whether an instruction is currently executing inside the unit) and time (representing the number of clock cycles since the currently executing instruction started execution). When the operation EFSM makes the transition from `ALU` state to `WB` state (signifying end of the `EX` stage of an arithmetic or logic instruction), it checks whether the ALU unit is ready to finish. This check is captured as a guard method *canRelease*.

# 4. TEST PROGRAM GENERATION

We now describe our test program generation method.

## 4.1 Coverage Metric

Previous works on specification driven test generation mostly consider pipeline structure to define the coverage metric. For example, the coverage metric can be defined to ensure that all the pipeline components 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 $I1$. We seek to capture behavioral scenarios in our test suite.

Given the pipeline model as a collection of communicating EFSMs $M_1, \ldots, M_n$, we compose the EFSMs to construct a **global FSM**. Any state of the global FSM consists of a local state and variable valuations for each of the component EFSMs. A transition in the global FSM constitutes a pipeline interaction. Note that a single transition in the global state space may involve progress (or lack thereof) of one or more instructions. In particular, a transition can include (1) the progress of one or more instructions in their operation EFSMs, which in turn requires cooperation from the respective resource EFSMs, and/or (2) progress in one or more resource EFSMs without any change in the operation EFSMs. An example of the latter is when an instruction is executing on the multiplier unit with multi-cycle latency. Here, the number of cycles left to complete execution is a variable associated with the multiplier EFSM. The multiplier EFSM will make a local self-transition and decrease the variable value; the operation EFSM of the corresponding instruction remains in the same state with same variable valuations.

Clearly, if we cover all the reachable states and transitions in the global FSM, we can cover all possible pipeline interactions. This is our coverage metric. That is, our generated test suite must cover all reachable states and transitions of the global FSM.

## 4.2 An Example

We now describe the test generation algorithm through a simple example. The example illustrates the main features of our method.

Suppose we have reached the global FSM state $s$ illustrated in figure 3. State $s$ has only four active instructions, three of which (`I1, I2, I3`) are in the instruction window and one is in the fetch queue (`I4`). We assume that the operation level EFSMs of the inactive instructions are in the idle state. Instruction `I1` has already been instantiated as a load instruction and it is now in the "execute" state (operation EFSM state of the instruction `I1`). This load instruction causes a cache miss; so the data cache (the DCache unit in Fig. 3) is currently in the "miss" state with elapsed time equal to 1. The other two instructions in the instruction window (`I2` and `I3`) are still in the "issue" state; they have not been instantiated to any operation class. Two functional units, ALU and MUL, are in "empty" state. The register resource is in the "partial" state with

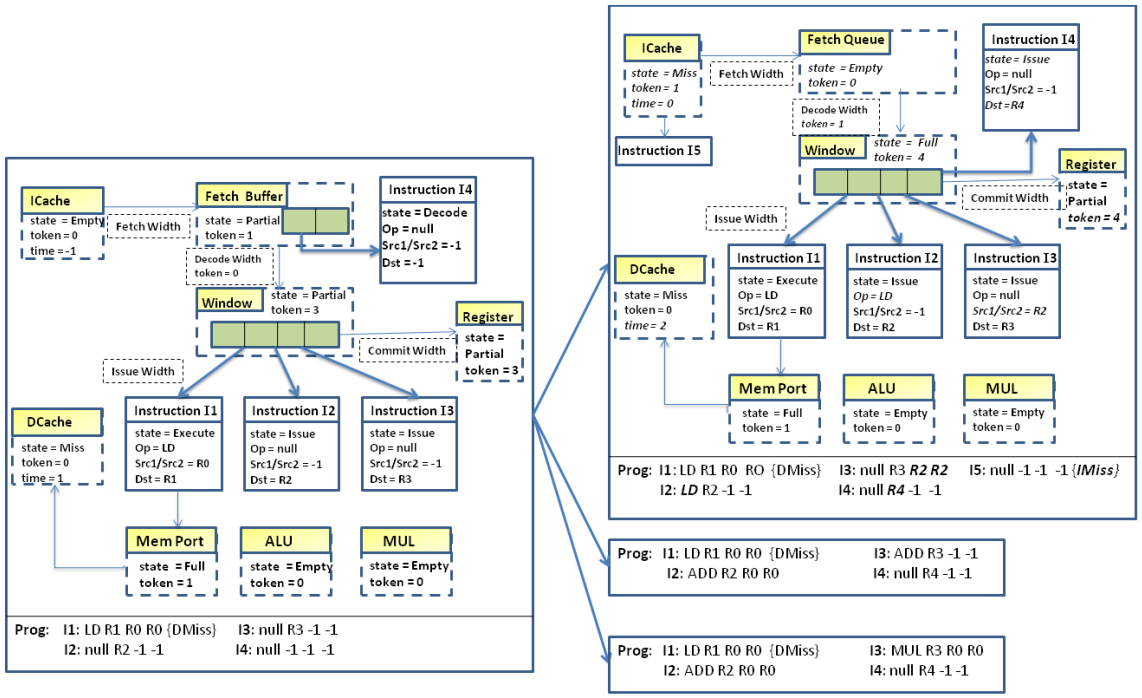


**Figure 3: Global FSM states and transitions.**

three tokens currently being held by three instructions in the window (register will be in "full" state when all the 16 registers are being used). The last instruction `I4` is in the "decode" state and is holding one token from the fetch queue. The fetch queue is in the "partial" state with one free slot available. As the instruction window and the fetch queue preserve the program order, we can deduce program order of `I1, I2, I3` and `I4`. At this point, the partially instantiated test program is as follows. Note that an uninstantiated opcode is written as `null` and an uninstantiated operand is written as `-1`.

```
I1: <op>LD   <dst>R1  <src1>R0 <src2>R0 // Dmiss
I2: <op>null <dst>R2  <src1>-1 <src2>-1
I3: <op>null <dst>R3  <src1>-1 <src2>-1
I4: <op>null <dst>-1  <src1>-1 <src2>-1
```

Now we construct all global FSM states reachable from the state $s$ shown in Figure 3. We process the operation level EFSMs in the order of the pipeline stages they are in, i.e., first the operation level EFSMs in the commit state, followed by write-back state, execute state, and so on. So we first process operation EFSM of instruction `I1`. For `I1` to make progress in the operation level EFSM, the resource EFSM of the memory port unit has to move to "completion" state, which in turn requires the data cache EFSM to move to "completion" state. As data cache needs 10 clock cycles to load data from memory, `I1` remains in the "execute" state. However, the data cache EFSM updates its elapsed time to 2 units.

Next, we check guard conditions in all possible transitions from the "issue" state in the operation level EFSMs of `I2` and `I3`. Several transitions can be satisfied. For purpose of illustration, let us choose the scenario where `I2` is instantiated as a load. `I2` will be stalled in the "issue" state due to resource contention for the memory port unit with instruction `I1`. For operation EFSM of `I3`, we choose the transition where it is also stalled in the "issue" state due to data dependency with `I2`. Accordingly, the opcode of `I2` and source register of `I3` are updated in the test program template.

Now we proceed to the operation level EFSM of instruction `I4`, which is in the "decode" state. The guard condition for a transition from "decode" to "issue" requires tokens from both the decode width and instruction window. Both of these conditions are satisfied. So `I4` returns one token to the fetch buffer, obtains new tokens from decode width and instruction window, and also gets a token for its destination register. The newly obtained register is recorded in the register resource by increasing it token count.

Finally, we progress the operation level EFSMs currently in the "idle" state. Suppose we fetch one instruction from the instruction cache and this instruction causes a cache miss. Consequently, one operation level EFSM moves from "idle" state to the "fetch" state. A new instruction `I5` is appended to our test program. The instruction cache transitions to the "miss" state with zero elapsed time.

This completes one of the global transitions from the global state $s$ and a new global state is reached as illustrated in figure 3. Our test program template is now refined to:

```
I1: <op>LD   <dst>R1  <src1>R0 <src2>R0 // DMiss
I2: <op>LD   <dst>R2  <src1>-1 <src2>-1
I3: <op>null <dst>R3  <src1>R2 <src2>R2
I4: <op>null <dst>R4  <src1>-1 <src2>-1
I5: <op>null <dst>-1  <src1>-1 <src2>-1 // IMiss
```

Clearly we can choose different transitions from the current state of `I2`, `I3`, `I5`. The combination of different local transitions leads to different global states, i.e., different pipeline interactions.

Note that the program with the 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. We also make sure that the data for the instruction `I1` (above) will not be present in the cache. Furthermore, we need NOP instructions to ensure that the pipeline is empty when our synthesized program starts execution.

## 4.3 Algorithm

Algorithm 1 is our test generation 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 generate the test suite *on-the-fly*, as we are traversing the FSM.

We start with the initially empty pipeline state $S_0$. As mentioned before, a global state consists of states and variable valuations for the operation and resource EFSMs. If we can have at most $x$ in-flight instructions, then we have $O_1, \ldots, O_x$ operation EFSMs — all in idle state in the beginning. Similarly, we have $R_1, \ldots, R_y$

resource EFSMs — all in empty state in the beginning. We search the global state space recursively starting with the initial state.

**Algorithm 1** Constructing the suite of test programs - Schematic

**Require:** $O_1, \ldots, O_x$ operation EFSMs, $R_1, \ldots, R_y$ resource EFSMs;
**Ensure:** Test program suite $testSuite$;

```
S_0 ← Initial state of O_1, ..., O_x and R_1, ..., R_y;
testPgm(S_0) ← null; testSuite ← null; Visited ← {S_0};
call TGen(S_0);

function TGen(S: Global state, testPgm: Test program){
  for all combinations of enabled local transitions in EFSMs {
    Let t_1, ..., t_x be the chosen transitions in O_1, ..., O_x;
    Let t'_1, ..., t'_y be the chosen transitions in R_1, ..., R_y;
    /* check feasibility of global transition consisting of local transitions */
    tempS ← S;
    for each operation EFSM O_i in reverse order of pipeline stages {
      if guard of t_i is true and action of t_i is feasible in tempS {
        tempS ← update(tempS, O_i, t_i);
      }
    }
    for each resource EFSM R_i {
      Let t'_i = s → s';
      if R_i is not in state s in tempS continue;
      if guard of t'_i is true and action of t'_i is feasible in tempS {
        tempS ← update(tempS, R_i, t'_i);
      }
    }
    if (tempS = S) continue; /* global transition is not feasible */
    testPgm(tempS) ← refine(S, tempS, testPgm(S));
    if (tempS ∈ Visited){
      testSuite ← testSuite ∪ {testPgm(tempS);
    } else { /* new global state */
      Visited ← Visited ∪ {tempS}; TGen(tempS);
    }
  }
}
```

The recursive routine TGen (see Algorithm 1) identifies all reachable global transitions starting with a global state $S$. A global transition consists of one or more local transitions in the operation and resource EFSMs. So we choose all feasible combination of local transitions out of state $S$. Given a combination of local transitions, we give priority to the moves of the operation EFSMs and among them the ones that are in the commit stage first, followed by those in the write-back stage, execute stage and so on. For two operation EFSMs in the same pipeline stage, we give priority to the one corresponding to the earlier instruction in program order. It is *necessary* to consider operation EFSMs in this order. As an operation EFSM makes a transition, it forces some resource EFSMs to make transitions as well. This may enable the next operation EFSM to make a transition. For example, in register bypassing, when an instruction `I1` moves from ALU to the write-back stage, another instruction `I2` dependent on `I1` can move from issue stage to execution. In hardware, these two moves happen in a single clock cycle; we achieve the same effect by advancing the operation EFSM of `I1` followed by that of `I2` in the *same* global transition.

If the transition $t_i$ of operation EFSM $O_i$ is feasible, then we apply the action method of $t_i$ on the global state to create a temporary global state $tempS$ through the $update$ function. Note that this action method may change the state of some resource EFSMs as well. For the next operation EFSM, the feasibility of the local transition is checked on this temporary global state $tempS$ rather than on the original global state.

Once all the operation EFSMs make their transitions, we move on to the resource EFSMs. The resource EFSMs may make independent transitions without affecting the operation EFSMs. However, the transitions of the operation EFSMs may disable some of the possible transitions of the resource EFSMs. Hence we check that the resource EFSM $R_i$ is in the same local state in both $tempS$ and $S$. The transition of the resource EFSM may enable transition of the operation EFSMs in the next global state.

While exploring the global FSM corresponding to a processor model, the set of visited states in the global state space is maintained via the $Visited$ set (we implement it as a hash table for efficient access). The path $\pi = S_0 \rightarrow S_1 \ldots \rightarrow S_m$ from the initial global 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 $\pi$, we maintain a test program $testPgm(S_i)$ which is the sequence of instructions driving execution along the path $\pi$ from empty state $S_0$ up to state $S_i$. The test program at $S_i$ may be "partially instantiated" where the opcodes/operands of some of the program instructions may be un-instantiated (this was shown in our illustrative example). These un-instantiated 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 $S_i$ while exploring a path $S_0 \rightarrow \ldots \rightarrow S_i$. Similarly, when we backtrack during state space exploration — say from state $S_{i+1}$ we backtrack to its parent state $S_i$ and then to another successor $S'$ of $S_i$ — we can readily use the (partially instantiated) test program of $S_i$ to construct the test program for $S'$.

Given a reachable global transition $S \rightarrow tempS$, the $refine$ function refines the test program at $S$ to create the test program at $tempS$. If the destination state $tempS$ has already been visited, we output the test program corresponding to it; otherwise we continue state exploration from $tempS$. If a generated test program contains any partially instantiated instruction, it means that any uninstantiated opcode/operands can be instantiated arbitrarily.

The following properties hold for our test generation algorithm.

THEOREM 1. *Algorithm 1 visits all global transitions reachable from the initial state $S_0$.*

Note that for any global state $S$ reachable from $S_0$, Alg. 1 invokes a recursive call $TGen(S)$. Further, all outgoing transitions from any reachable global state are explored.

THEOREM 2. *The $testSuite$ size from Alg. 1 is bounded by the number of global transitions reachable from the initial state $S_0$.*

This can be proved by showing that each test program generated covers at least one new reachable global transition. As observed from experiments, our test-suite size is *much smaller* than the number of global transitions (which in turn is much smaller than the test-suite size produced by existing approaches like [6]).

# 5. EXPERIMENTAL EVALUATION

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 have modeled an Alpha 21264 like superscalar processor in the SimpleScalar architectural simulation framework [2]. We specify the ISA in MESCAL Architecture Description Language (MADL) [9]. 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. 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

| Processor Configuration | # States $\|\mathcal{S}\|$ | # Transitions $\|\mathcal{T}\|$ | # Test Programs $\|\mathcal{TP}\|$ | Avg. Test Program Length | Test Gen. Time |
|---|---|---|---|---|---|
| In-order, superscalarity=1 | 42,754 | 95,086 | 13,232 | 16 | 1m30s |
| In-order, superscalarity=2 | 43,590 | 124,256 | 15,334 | 17 | 1m44s |
| Out-of-order, superscalarity=1 | 220,039 | 522,088 | 84,401 | 17 | 8m26s |
| Out-of-order, superscalarity=2 | 310,993 | 851,041 | 116,260 | 20 | 12m24s |

**Table 1: Statistics for our test generation algorithm for different processor configurations: reachable states, transitions, test programs, length of test programs (in number of instructions), and test generation time.**

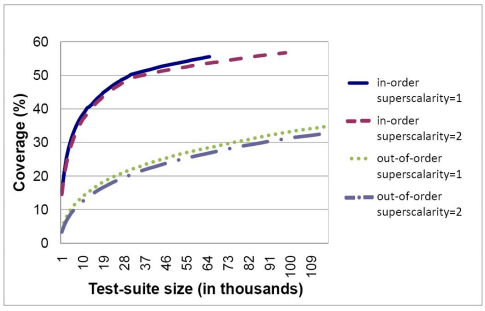


**Figure 4: Coverage from random test generation. Our coverage is 100%.**

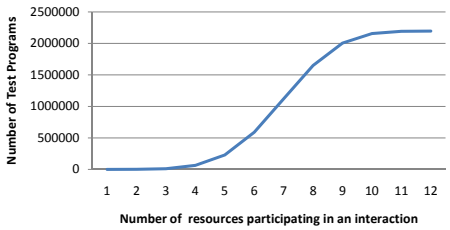


**Figure 5: Test-suite size from existing formal approach [6].**

module is invoked to traverse and identify all reachable states in the global FSM (obtained by composing the communicating EFSMs). Recall that a partially instantiated test program is maintained as we traverse a path through the global FSM state space. Once a path with at least one previously unexplored global FSM state is formed, our test program construction component creates an *assembly level program* that can exercise this path. Thus, a test-suite is formed. We also validated our test-suite by successfully *executing* the generated test programs.

We consider four different processor pipeline configurations. The most complex pipeline supports out-of-order and 2-way superscalar execution. We restrict superscalarity to one (i.e, scalar pipeline), and also study in-order pipelines (with superscalarity = 1 or 2).

Table 1 presents the various statistics on our test generation algorithm. We report the number of reachable global states/transitions. Note that the number of reachable states and transitions are only a fraction of the total number of global states and transitions (which are in the order of $10^{20}$). The number of test programs generated by our method are given in the next column. As discussed earlier, we cover all reachable states and transitions via our test program suite. However, as a generated test program covers several global states/transitions, the test suite size is much smaller than the total number of global states/transitions.

The average length of our test programs vary between 16–20 instructions, which is *pretty small*. The runtime of our test program generation algorithm is very reasonable. Even for the most complex processor, we can generate all the test programs to cover 116K transitions within 12 minute 24 seconds (on a 2.6 GHz Intel Pentium IV machine with 1 GB of main memory).

Next, we evaluate random test generation methods. We identify the set of states $\mathcal{R}$ visited by the random test programs among all reachable states $\mathcal{S}$. Then $(\frac{|\mathcal{R}|}{|\mathcal{S}|} \times 100)$ is the coverage percentage. Our method is guaranteed to achieve 100% coverage. The coverage percentage of random method (Y-axis in Figure 4) diminishes quickly to as low as 31% with increasing complexity of processor.

Finally, we compare the quality of our test suite with that of the most recently proposed fault model [6] for processor pipeline interactions (see Figure 5). Assume that there are $n$ resources each with an average of $r$ activities. The work of [6] *estimates* the total number of pipeline interactions as $\sum_{k=1}^{n} {}^nC_k \times r^k$, and generates test programs for these interactions with the help of a model checker. We choose the simplest architecture with in-order execution and superscalarity = 1 for the comparison. The X-axis of Figure 5 shows $k$, the restriction on the number of functional units participating in a pipeline interaction (this number is imposed by [6]), and the Y-axis shows the number of test programs for varying $k$. Given $k$, we find which of the $\sum_{i=1}^{k} {}^nC_i \times r^i$ interactions are actually encountered in the global state space; only the number of test programs corresponding to these feasible interactions appears in the Y-axis of Figure 5. We see that [6] generates more than 2 million tests (Fig. 5), whereas we require 13,232 tests (Table 1).

# 6. CONCLUSION

In this paper, we have presented a fully formal processor modeling framework and used our processor models for systematic test generation. Our test generation method is (i) efficient, (ii) covers *all* pipeline interactions, (iii) produces executable test programs of short length and, (iv) produces a compact test-suite which is less than 1% in size compared to test-suites produced by existing approaches. Thus, our approach greatly reduces validation effort.

*Acknowledgments.* This work was partially supported by an A*STAR (Singapore) grant R252-000-258-305.

# 7. REFERENCES

[1] A. Aharon et al. 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] K.T. Cheng and A.S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *DAC*, 1993.
[4] T.A. Diep and J.P. Shen. Systematic validation of pipeline interlock for superscalar microarchitectures. In *FTCS*, 1995.
[5] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. Coverage-directed test generation using symbolic techniques. In *FMCAD*, 1996.
[6] H-M. Koo and P. Mishra. Functional test generation using design and property decomposition techniques. *ACM TECS*, 8(4), 2009.
[7] P. Mishra and N. Dutt. Specification-driven directed test generation for validation of pipelined processors. *ACM TODAES*, 13(2), 2008.
[8] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *DATE*, 2003.
[9] W. Qin, S. Rajagopalan, and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES*, 2004.
[10] S. Ur and Y. Yadin. Micro-arachitecture coverage directed generation of test programs. In *DAC*, 1999.
[11] Q. Zhu, A. Shrivastava, and N. Dutt. Functional and timing validation of partially bypassed processor pipelines. In *DATE*, 2007.