

Micro Architecture Coverage Directed Generation of Test Programs

Shmuel Ur(sur@vnet.ibm.com) Yoav Yadin(yadin@tx.technion.ac.il)
IBM Haifa Research Lab

1. Abstract

In this paper, we demonstrate a method for generation of assembler test programs that systematically probe the micro architecture of a PowerPC superscalar processor. We show innovations such as ways to make small models for large designs, predict, with cycle accuracy the movement of instructions through the pipes (taking into account stalls and dependencies) and generation of test programs such that each reaches a new micro architectural state. We compare our method to the established practice of massive random generation and show that the quality of our tests, as measured by transition coverage, is much higher. The main contribution of this paper is not in theory, as the theory has been discussed in previous papers, but in describing how to translate this theory into practice in a practical way, a task that was far from trivial.

2. Introduction

Functional verification comprises a large portion of the effort in designing a processor [2]. The investment in expert time, and computer resources is huge, but so is the cost of delivering faulty products [1]. Formal verification [15][11] is not yet the answer as it cannot deal with the size of modern processors. In current industrial practice, random architectural test program generators and simulation methods are used to implement large portions of verification plans [2][4][5][6][19]. Massive amounts of test programs are generated and run through an architecture simulation model and through the design simulation model, and the results are compared. The test space is enormous, thus even a large number of tests represent only a small section of the test space. In practice, actual coverage of the global test space is unknown, and there is a lack of adequate feedback from the verification process as to the quality of the tests that are simulated.

In order to solve some of these downfalls, a new verification methodology was introduced by R. C. Ho, C. Han Yang, M. A. Horowitz and D. L. Dill in [12]. This methodology contains three basic phases: (1) describing the processor implementation control as a Finite State Machine, (2) deriving transition coverage on the FSM using methods from formal verification, (3) automatic translation of the covering tours to test vectors. This method is attractive because it is formal, mostly automatic, and produces high quality tests. Another advantage of the method is that there is an efficient use of expert designers' time in the verification process; designers focus on the control in the section that they are responsible for, and the interactions between the sections are derived automatically by the process. Unfortunately, this methodology is not applicable to modern, real-life, superscalar processors. This is due to the size of the FSM needed to describe such processors.

In [18], a similar methodology, called CDG (for Coverage Directed Generation) was suggested but not demonstrated. CDG had a few key innovations; the first was that the description of the implementation as a Finite State Machine is much smaller. The second was a modular approach which enabled the use of existing industrial tools (e.g. [2][7]). The third was that, in the last stage, test programs, i.e. regular assembler programs that can run on the hardware, are produced as opposed to test vectors.

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 99, New Orleans, Louisiana
(c) 1999 ACM 1-58113-109-7/99/06..\$5.00

This paper shows the first implementation of CDG to a superscalar state of the art PowerPC implementation[17][16]. The experiment, which is described in detail, includes modeling of parts of that processor in SMV[10], generating abstract tests from the model using CFM [7], converting the abstract tests into restrictions on architectural tests, converting the restrictions into directives for a test generation tool, generating the tests using Genesys [4], executing the tests on the real implementation and verifying, clock by clock, that the real tests executed match the abstract tests. Similar methodologies that use formal verification to drive test generation have been suggested in the past [15][13]. As the goal of this paper is to describe an experiment, the difference between the methodologies (the interested reader can refer to [18]) is not elaborated.

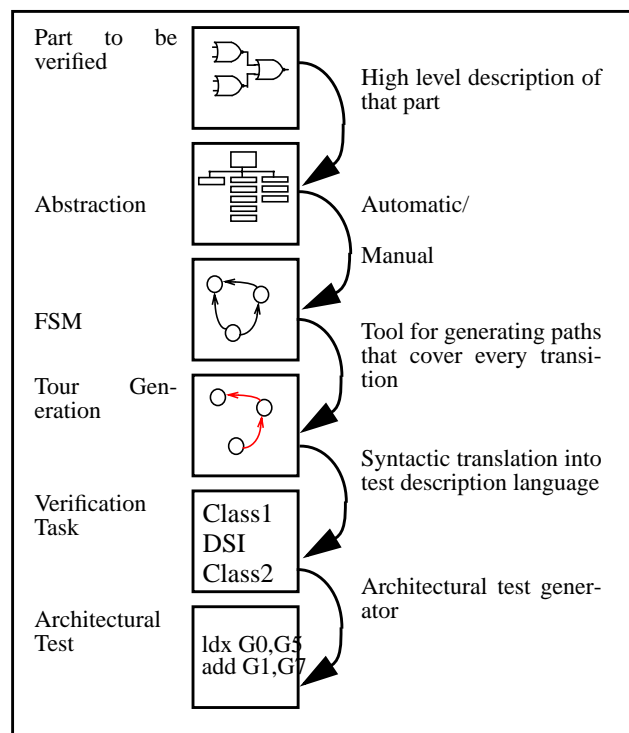


Figure 1. Steps in the methodology

The results of the experiment were very encouraging both from the theoretical and the practical aspects. We showed that, using CDG, it is possible to generate architectural tests with known micro architectural properties for a real, very large, design. We showed that, using a very small model, we can generate tests for the real pipelines that have exactly the same timing. This is a very important achievement because it lets us generate directly, and not by trial and error, cases in which the window of opportunity for an event is very narrow. We compare our results to random test generation and show its benefits. The rest of this paper is organized as follows: In section 3, we give an overview of CDG. In section 4 we describe the processor to which we apply CDG. In section 5, we describe the experiment with its many stages. Section 6 shows the results of the experiment, and we conclude in section 7.

3. Overview of CDG

Figure 1 gives an overview of the stages in CDG as applied to testing a pipeline. The stages of the process are briefly as follows: The implementation of the processor pipe control and surrounding sections are examined and abstracted. The abstracted processor control is then used to build a set of Finite State Machines (FSM's) which is used to automatically derive verification tasks. The product FSM has two important qualities. First, the machine contains states and transitions that represent complex corner cases in the actual implementation. Second, there is a simple mapping from tours on the machine to constraints on architectural test programs. The model created can be made much smaller than a model created using abstraction techniques[18].

CFSM [7] is the tool we used to automatically generate a set of tours that traverse all the edges of the FSM model. Each tour is represented by a sequence of inputs to the FSM. These input sequences are then translated automatically into a test description language that describes constraints on instruction sequences for an architectural test program generator, such as Genesys [2][4]. This is the crucial stage of the process; if the FSM model is not correctly designed, then translation of tours to constraints is impossible. Architectural test program generators, such as Genesys, read the test description language and generate many real test programs that fit the given constraints, but contain variations of the unspecified parameters of the sequence.

4. Processor Description

This work was done on a state-of-the-art PowerPC four way super-scalar processor. The segments of the processor which we modeled are (as depicted in Figure 2):

- A simple arithmetic pipe (S-Pipe) which processes fixed point instructions with an execution time of one cycle
- A complex arithmetic pipe (C-Pipe) which processes all arithmetic instructions, including S-Pipe instructions
- A dispatch buffer with two registers

4.1 Model Description

The dispatching algorithm can dispatch, at most, one instruction to a pipe per cycle. The second instruction can be dispatched only if the first instruction has been dispatched (no out of order). The instruction in the first slot is dispatched to the appropriate pipe if this pipe is not stalled. The instruction in the second slot is dispatched to the appropriate pipe if the instruction in the first slot has been dispatched, and the appropriate pipe is empty and has not stalled.

The only exception is if two S-Pipe instructions are dispatched and there is no stall. In this case, the first instruction is dispatched to the S-Pipe and the second to the C-Pipe. If there is a stall in the S-Pipe, none of them are dispatched.

The C-Pipe and the S-Pipe are both composed of three stages as sketched in Figure 2 (designated S0,S1,S2 and C0,C1,C2) the dispatch-decode-read, execution, and writeback-complete. The execution stage takes at least one cycle. All the others take exactly one cycle. Between each two corresponding stages, there is an order bit that shows which instruction was first in the program order.

Two types of dependencies between the S and the C pipes are described in the model. The first is the register dependency and the second is a dependency on an internal resource. Dependencies within the pipe are not described in the model since forwarding mechanisms are fully implemented between and within the pipes (see Figure 2) and therefore these dependencies do not affect timing. The dependencies between pipes which effect timing are implemented. Each pipe can be stalled either by the other pipe or by instructions with an execution time longer than one cycle.

5. Experiment

The experiment was conducted on the two arithmetic pipelines of the processor - the S and C pipes - and on parts of the dispatch unit. It contains the following steps:

1. Modeling the behavior of the S and C pipes and part of the dispatch unit as an FSM.
2. Choosing the state machine variables for which we want to check coverage (actually, the coverage refers to transitions of these variables).
3. Running CFSM [7], to get a set of abstract tests that contain all interesting transitions. Each abstract test is a path on the product machine whose starting point is the empty pipeline. It progresses according to the FSM model until a new transition is found, and continues moving according to the FSM model until the pipeline is empty again. These abstract tests contain enough information to reconstruct concrete tests that follow the same path.

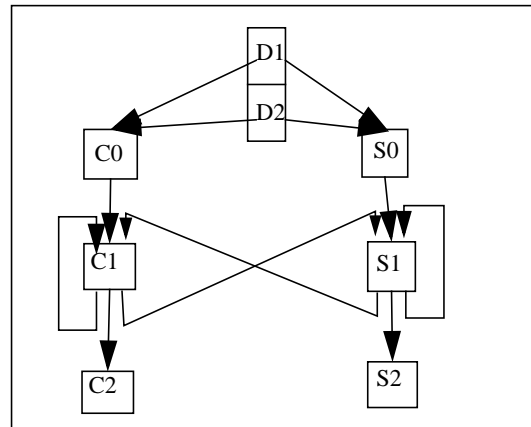


FIGURE 2. Description of the arithmetic pipes

4. Translating abstract tests into lists of instructions that follow the same path when run through the pipe. This list includes the instruction types, the order of the instructions, and information about dependencies between the instructions.
5. Building a Genesys [2] def file (a parameter file containing directives on how tests should be generated) according to these lists of instructions, and generating tests from these def files.
6. Running the tests through a behavioral simulator (texsim), that prints out a trace containing the information needed to check the values of the state machine variables in the processor in each clock.
7. The information from the simulated traces is translated back into state machine variables. For each clock, the state machine values are determined. For each test, we check whether the path it went through in the simulation is exactly the same path as we intended. When there is a miscompare, we analyze it to improve the process.

As part of the experiment we also generate random tests in order to compare CDG to random test generation.

Creation of the model is the hardest and the only manual stage in this experiment. The model has to be debugged and then tested. We believe that this process is of interest as it is different from debugging and testing for software or hardware. The problem is first to get the model to behave the way one thinks it should (the debug-

ging phase), and then to make it behave like the design (whose behavior changed during the time the experiment was run). We elaborated on this process at the end of this section.

5.1 Modeling the behavior of the pipelines as a finite state machine

The FSM model includes the following variables:

- Stage valid bits (3 per pipe)
- Instruction type - These two variables (1 per pipe) indicate how many cycles the instruction, currently in stage 0, should execute, which pipes it can go to (S & C pipes or just C-Pipe), and whether or not this is a legal instruction.
- Stall bits - Flags (1 per pipe) indicating whether or not the pipe should stall for the cycle. Possible reasons for stalls: the instruction did not finish executing, an instruction in the other pipe did not finish executing (it causes stall to avoid out of order execution), or dependencies between instructions.

```

MODULE S_pipe( d1, d2, o1, cv1, c, cs1)
VAR
sv0 : {0,1}; - The valid bit of stage 0 in the S pipe
sv1 : {0,1}; -- The valid bit of stage 1 in the S pipe
sv2 : {0,1}; -- The valid bit of stage 2 in the S pipe
is0 : {0,1}; -- Currently always 0
ss1 : {0,1}; -- The stall bit of the S pipe
scr1 : {0,1}; -- 0 if the instruction does not touch the cr
ASSIGN
sv0 := case
  ss1 : 0; -- If the pipe S is stalled no new instruction
  cs1 & (d1 = C) : 0; -- C is stalled an stalls S
  (d1 = S) | (d2 = S) : 1; -- No stalls, an S will be dispatched
  1 : 0;
esac;
init(sv1) := 0;
next(sv1) := case
  sv0 : 1; --An instruction in stage 0 will progress to 1
  sv1 & cv1 & o1 & (cs1 | c) : 1; -- Stalled by the C pipe
  1 : 0;
esac;
init(sv2) := 0;
next(sv2) := case
  !sv1 : 0; -- Nothing in sv1 -> nothing progresses.
  ss1 : 0; -- S is stalled
  1 : 1;
esac;
ss1 := case
  !sv1 : 0; -- No instruction no stall
  cv1 & o1 & (cs1 | c) : 1; -- C stals S
  1 : 0;
esac;
init(scr1) := 0;
next(scr1) := case
  sv0 & is0 = 1 : 0; -- illegal instruction
  sv0 : {0,1}; -- Randomly deciding if touching cr
  ss1 : scr1; -- stalled keep previous information
  1 : 0;
esac;

```

Figure 3. The S pipe description

- CR bits - Flags (1 per pipe) indicating whether the instruction currently in stage 1, uses the CR (Condition Register, an internal resource) or not. These flags are also used to determine dependencies.

- Order bits - Three flags indicating, for each stage, whether the instruction in the C-Pipe came before the one in the S-Pipe. These flags are used to avoid out-of-order-execution.
- Dispatch status - Theoretically, at each clock, two instructions can be dispatched. Two variables indicate if there are instructions ready to be dispatched, and which pipe they should be dispatched to.
- Dependency flag - Indicates whether or not the instructions in stages S1 and C1 share a resource. This resource can be the CR that was mentioned above, or a general purpose register.

Figure 3 shows the non-deterministic model of the S-Pipe (in SMV). It includes the 3 stage valid bits, a stall bit, the instruction type in S0 and a flag indicating whether the instruction in S1 updates the CR. The rules for sv0 (S0 valid bit) state that if there was a stall on the S-Pipe in the previous clock, sv0 will be 0 (empty stage), if the C pipe was stalled, and the first instruction in the dispatch buffer goes to the C-Pipe, sv0 will also be 0, otherwise, if one of the dispatch buffers includes an S-Pipe instruction, sv0 will be 1, and if not - sv0 will be 0.

Table 1: Sample variables from a CFSM test

Variable name	Clock number									
	1	2	3	4	5	6	7	8	9	10
d1		C	S	S						
d2		S		C						
sv0	0	1	0	0	1	0	0	0	0	0
cv0	0	1	0	0	1	0	0	0	0	0
cv1	0	0	1	1	0	1	1	1	0	0
cv2	0	0	0	0	1	0	0	0	1	0

5.2 Choosing the state machine variables for which we check coverage

Coverage was checked in each clock on the transitions of the state variables. In order to avoid covering a large number of transitions which would require many tests and more computational resources than were available, we focused on a small number of interdependent variables. Dependencies between the variables cause some combinations of values and some transitions to be illegal, further reducing the resources needed.

The variables we chose for the first experiment were the two stage 1 valid bits (for stages S1 and C1), the flag indicating whether the instruction in S1 uses the CR, and a variable indicating the type of the instruction in stage 0 of the C-Pipe. This last variable could get a number of values indicating whether the instruction can execute in the S-Pipe as well as the C-Pipe, if it is a legal instruction, and for how many clock cycles it should execute.

5.3 Running CFSM, to get abstract tests for the FSM

Given an FSM model and the list of coverage variables, CFSM outputs an abstract test (a traversal on the state machine) for each possible transition. Table 1 shows an example of a partial abstract test. By following the dispatch (variables d1 and d2) one can see that the first instruction is of type C, the second and third of type S and the fourth of type C. By following the valid bits, one can check where each instruction was at each clock.

5.4 Translating CFSM paths into lists of instructions

Each CFSM path is translated into a list of instruction types. This list is based on the values of the valid stage, instruction type and CR variables in each clock. Also, the list includes information about dependencies between instructions, which is based on the CR flags and dependency variables.

```
dep 1 2 instr1 - C3.CR instr2 - S0.CR
    instr3 - S0.NCR instr4 - C4.CR
```

Figure 4. Example of a list of instructions

Figure 4 shows an example of an instruction list derived from the abstract test in Table 1. For example, C3.CR designated an instruction that can go only to the C-Pipe, executes in one cycle and touches the CR.

5.5 Creating concrete tests

The lists of instruction types and dependencies are used to specify the input to an architectural test generator.

The architectural test generator that we use (Genesys) takes the specification and outputs test programs. It creates full instructions with all their resources so that the test can be executed on a real processor. It also creates the initial and final memory content. It is important to use a good architectural test generator and to create a number of test programs for each specification. CDG ensures the generation of test programs that reach many control configurations. It is useful to reach every one of these configurations a number of times with different data. The reason is that some bugs can be discovered only by specific combinations of control and data configuration (for example dividing a small number by a large number while there is a register dependency between the divide instruction and a load instruction). Figure 5 shows an example of a test program matching the instruction list in Figure 4.

```
9836 * EA=DBA000000109C0F4 sld G13,G1,G19
1815 * EA=DBA000000109C0F8 addc. G9,G13,G3
2B02 * EA=DBA000000109C0FC addi G2,G11,0x2B02
B9D7 * EA=DBA000000109C100 mullw. G28,G8,G23
```

Figure 5. Example of a test

5.6 Running tests on a simulator

The tests were simulated using texsim. A trace (see Figure 6) which contains the values of the state variables at each stage is created.

5.7 Generating random tests

In order to correctly evaluate the experiment results, coverage of the experiment tests was compared to coverage of randomly generated tests. The random tests are approximately twice as long as the experiment tests, contain the same set of instructions, and the same restrictions on cache miss and interrupts. We used Genesys register interdependency options to make sure that the tests generated will have more register dependencies than at random, since some of our coverage tasks included interdependencies. The test generation directives that we used were as focused on the coverage model as we could make them without aiming for specific coverage tasks.

The random tests were also run on the simulator, and their coverage was compared to the coverage of the tests generated using CFSM.

5.8 The FSM model debugging/testing process

The FSM model, written in SMV, was debugged by writing temporal rules using rulebase[14]. The rules were used to check that the model behaves as expected. The FSM model was tested by comparing the timing of the abstract tests created to the timing of concrete tests executed on the actual design. We explain the debugging/testing process by showing how the design mistakes that existed in the FSM model were found

5.8.1 Debugging the FSM Model

Symbolic model checking is used for the debugging of the FSM model in exactly the same way that it is used for testing hardware[14]. This is the only way to debug the model as simulation is not available for SMV. Temporal rules are written to check that the model behaves as expected.

Example of rule:

- formula { AG({counter > 0 [+], counter = 0 , 1} (cv2)) }

This rule state that if a variable named *counter* was greater than zero, then in the clock immediately after it becomes zero the valid bit of stage C2 will be on. We expect this rule to be true as *counter* greater than zero indicates an instruction in stage C1. Once *counter* became 0 the instruction finished executing and moves from stage 1 to stage 2 of the C pipe..

```
clock 15
COValid S0Valid
instr_num=1,instr=0x7c2d9836,stage=m0(2, 7),
mnem=sld(no. 380, grp 3), source_gpr=40001000,
target_gpr=00040000, source_fpr=00000000,
target_fpr=00000000
instr_num=2,instr=0x7d2d1815,stage=r0(1, 4),
mnem=addc(no. 3, grp 2), source_gpr=10040000,
target_gpr=00400000,source_fpr=00000000,
target_fpr=00000000
end of clock

clock 16
ROValid
instr_num=2,instr=0x7d2d1815,stage=r1(1, 5),
mnem=addc(no. 3, grp 2), source_gpr=10040000,
target_gpr=00400000,source_fpr=00000000,
target_fpr=00000000
instr_num=1,instr=0x7c2d9836,stage=m1(2, 8),
mnem=sld(no. 380, grp 3), source_gpr=40001000,
target_gpr=00040000, source_fpr=00000000,
target_fpr=00000000
M12Stall R12Stall
end of clock
```

Figure 6. Part of a trace

5.8.2 Testing the FSM Model

The SMV model is used to generate abstract tests in which the value of each variable in each clock is predicted. The abstract tests are converted to concrete tests, as described above, and are executed on the real implementation. The timing and results of the abstract and concrete tests are compared. Every bug in the SMV model causes a mismatch between the abstract and concrete tests. We examine the tests that are not created correctly and check the reason. If the problem was in our modeling we fix the SMV model, else we fix the implementation, and rerun the process.

First iteration

In the first iteration, 56 abstract tests that cover all the transitions over three variables were generated. Five concrete tests were generated for each abstract test for a total of 280 tests. We generated more than one concrete test for each abstract test in order to overcome transitory recreation problem such as those created by cache-misses (which change timing).

The main cause for errors was a bug in our SMV model. Certain assembler instruction did not update the Condition Register in our model while in the implementation it did. This caused dependencies and stalls in the real pipe, but not in the model's pipes, which threw our tests off the right track.

Out of the 56 abstract tests, 42 were reconstructed successfully. Out of these 42, 27 were reconstructed successfully on the first attempt. The number of times each abstract test was reconstructed successfully (out of 5 attempts) appear in Table 2.

Second Iteration

After correcting the bugs found on the first iteration we ran the entire experiment using four variables in the cover set. The results (summarized in Table 2) were not as good as expected with our method being only slightly better than random testing. Upon further investigation, in particular looking at the tests that where not recreated correctly, we have found the following problems:

1. Cache misses, which are not modeled in the SMV model, can happen. If a cache miss occurs, the timing of the concrete and abstract tests will not match. We tried to lower the probability of a cache miss, but we cannot reduce it altogether.
2. We have implemented incorrectly, in the SMV model, dependencies between a long instruction in the C pipe and an instruction in the S pipe when the dependency is on a machine resource. Unlike dependency on an architectural resource, which delays long instructions in the S pipe by one cycle, dependency on the CR is resolved while the instruction in the C pipe is being executed and therefore no delay is needed.
3. We found a bug in Genesys and a bug in CFSM which were fixed.

6. Experimental Results

The final iteration was conducted on the same variables as the previous one but after fixing the FSM model. There were a total of 240 paths to reconstruct. Out of the 240 paths, 200 were reconstructed successfully at least once (Table 2).

Due to technical problems with CFSM, which have been fixed since, too many tests were created. The number of tests should be at most, the number of transitions as each test covers a new transition. After the bug fix, and after adding a test set compaction component [3], 137 tests covered the 218 transitions

Figure 7 shows the coverage obtained by CDG tests compared to the coverage obtained by random tests. The short random tests were 5 instructions long, and the long random tests were 10 instructions long. The average length of a CDG test was 5-6 instructions.

The CDG tests reached 100% transition coverage after 800 tests, even though not all the tests were reconstructed correctly. Long random tests reached 96% coverage after over 4800 tests. Getting the last tasks done is hard as can be attested by the fact that the last 1500 tests discovered only two new tasks.

The CDG coverage graph includes all five attempts to reconstruct the abstract tests. First we checked the coverage obtained by the first attempt to reconstruct the abstract tests, then we added the coverage obtained by the second attempt, etc. The graph shows that using CDG, taking only the first attempt to reconstruct the abstract tests, kept the coverage above one transition per test. Using all five attempts to reconstruct the tests kept us above one transition per five tests.

# correctly reconstruct ed tests (out of 5)	After debugging	After initial testing	In the final experiment
0	14	194	40
1	4	20	32
2	10	50	76
3	15	37	47
4	9	28	32
5	4	11	13

Table 2: Quality of test reconstruction improves with testing/debugging

6.1 Resources used

Generation of all the abstract tests using CFSM took about 1 minute on a RS/6000 Model 250 workstation. In comparison, generation of the concrete tests and the tessim simulation took about two minutes per test. Therefore, the computer resources used for CDG were about one day on a Model 250 workstation while about a week's cpu time was used for random test generation. This compression is misleading as the main resource used by CDG is expert time. It took us about three man months to execute this experiment. We assume that doing something similar again after the methodology is understood, will take about a man month.

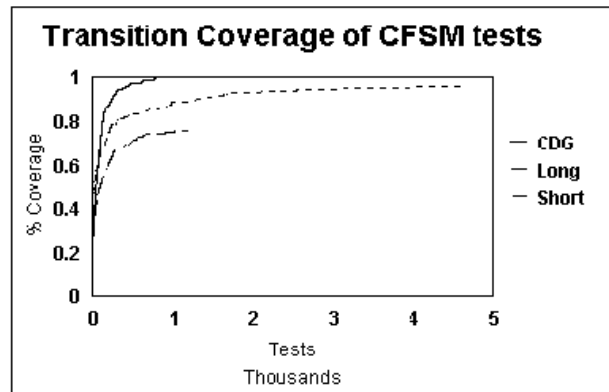


Figure 7. Comparing CDG with random tests

6.2 Analysis of the transitions not covered by random generation

We have looked at the 8 transitions not covered with the random biased test generator. Half of these transitions require, as a partial requirement, the following two-cycle scenario:

1. An instruction in the S-Pipe that stalls an instruction in the C-Pipe due to a dependency on the CR. The instruction in the C-Pipe is of the type that executes in one cycle
2. The instruction in the S-Pipe moves to the third stage. The S and C pipes receive two new instructions. The new S instruction is also dependent on the CR. Some transitions, which have been covered by random generation, are of this kind. It is our estimate that full coverage can be reached using the random test generator but will require a very large number of tests.

7. Conclusions

In this paper, we described an experiment in which we combined the power of a model checker with an architectural random test generator to produce tests that have very desirable micro architectural properties (as explained in the paper), desirable architectural properties (Section 4.5), and can be executed in a standard simulation environment.

When coverage is done using a random test generator, the shape of the coverage graph (tasks vs. tests) is always the same: initially, coverage progresses rapidly and then it starts to slow down until eventually it flattens out. When the coverage goal is "easy" or the test generator is "good", flattening will happen after most of the tasks have been covered.

The advantage of using formal methods which process the FSM of the model, is that there are no longer hard cases. Every new test covers at least one new task. The so-called hard scenarios, those that require a specific window of opportunity which happens very rarely with random generation (we have seen examples where it took three man months and hundreds of millions of cycles to recreate a test for a known bug), pose no problem to CDG. Models that, in the past, required generation of millions of tests in order to cover thousands of tasks, will require only thousands of tests with CDG.

We have seen that the advantage over random test generation increased with the size of our SMV model, and with the number of tasks to cover. An advantage of our modeling strategy [18] is that we can model very large machines. It is our estimate that we can make a model four times the size (one that includes all the pipes, branch prediction and interrupts) before we have to start worrying about the state explosion problem. In larger models, the benefit over random test generation will make CDG very attractive.

Another advantage over random architectural test generation is the fact that the SMV model can be used as a reference model for performance. In this experiment we not only checked that the results of the tests were correct, but also that the execution of the tests was done in the right timing. This is very important as many of the performance bugs (e.g. an unnecessary stall) never cause incorrect results.

There are two drawbacks to our methodology; the first is that there is a need for an expert for the modeling effort in the first stage of the process. The second is that, in order for the abstract tests to be translatable to concrete tests, we need to model the timing accurately on the entire processor. This is not hard to accomplish for tests on the dispatch algorithm, pipes, and branch prediction. However tests containing the cache controller, or other parts with complex timing, will be much harder to model accurately.

Bibliography

1. B. Beizer, "The Pentium Bug, an Industry Watershed", *Testing*

- Techniques Newsletter On-Line Edition*, September 1995
2. A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molco, G. Shurek "Test Program Generation for Functional Verification of PowerPC Processors in IBM", In proceeding of *ACM/IEEE Design Automation Conference* 1995
 3. E. Buchnik, S. Ur. "Compacting Regression Suites On-The-Fly" *APSEC*, December 1997
 4. Y. Lichtenstein, Y. Malka, A. Aharon "Model Based Test Generation for Processor Design Verification", In *Innovative Applications of Artificial Intelligence (IAAI)* AAAI Press 1994
 5. A. M. Ahi, G.D. Burroughs, A.B. Gore, S.W. LaMar, C.R. Lin, A.L. Wieman, "Design Verification of the HP9000 Series 7000 pa-risc Workstations", *Hewlett-Packard-Journal* num. 8 vol. 14 August 1992
 6. A. Chandra, V. Iyengar, D. Jameson, R. Jawalker, I. Nair, B. Rosen, M. Mullen, J. Yoor, R. Armoni, D. Geist, Y. Wolfsthal "AVPGEN - A Test Case Generator for Architecture Verification", *IEEE Transactions on VLSI Systems* 6(6) June 1995
 7. D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y. Wolfsthal "Coverage Directed Generation Using Symbolic Techniques", *FMCAD Conference November 96*
 8. G. J. Holtzman, "Design and Validation of Computer Protocols", Prentice Hall, Englewood Cliffs, NJ 1991
 9. K.L. McMillan "Symbolic Model Checking" Kluwer Academic Press, Norwell MA 1993
 10. K.L. McMillan "The SMV System DRAFT", Carnegie Mellon University, Pittsburgh PA 1992
 11. A.K. Chandra, V.S. Iyengar, R.V. Jawalekar, M.P. Mullen, I. Nair, B.K. Rosen "Architectural Verification of Processors Using Symbolic Instruction Graphs", In *Proceedings of the International Conference on Computer Design*, October 1994
 12. R. C. Ho, C. Han Yang, M. A. Horowitz, D. L. Dill "Architecture Validation for Processors" In *ACM ISCA* 1995
 13. H. Iwashita, S. Kowatari, T. Nakata, F. Hirose "Automatic Test Program Generation for Pipelined Processors", In *Proceedings of the International Conference on Computer Aided Design*, November 1994
 14. I. Beer, M. Yoeli, S. Ben-David, D. Geist and R. Gewirtzman, "Methodology and System for Practical Formal Verification of Reactive Systems", *CAV94 Conference*, LNCS818, pp 182-193
 15. T. A. Diep, J. P. Shen "Systematic Validation of Pipeline Interlock for Superscalar Microarchitectures" In *Proceedings of the 25th Annual International Symposium on Fault Tolerance*, June 1995
 16. C. May, E. Silha, R. Simpson, H. Warren editors "The PowerPC Architecture", Morgan Kaufmann, 1994
 17. S. Weiss, J. E. Smith "POWER and PowerPC", Morgan Kaufmann, 1994
 18. D. Lewin, D. Lorenz, S. Ur "A Methodology for Processor Implementation Verification", *FMCAD conference November 96*
 19. A. Hosseini, D. Mavroidis and P. Konas "Code Generation and Analysis for the Functional Verification of Microprocessors", In *Proceeding of the 33rd Design Automation Conference*, June, 1996.
 20. M. Kantrowitz and L.M. Noack, "I'm Done Simulating: Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor", In *Proceeding of the 33rd Design Automation Conference*, June, 1996.