Functional and Timing Validation of Partially Bypassed Processor Pipelines

Qiang Zhu
shiyu@labs.fujitsu.com

Fujitsu Laboratories LTD., Japan
1-1, Kamikodanaka 4-Chome,
Nakahara-ku, Kawasaki 211-8588

Aviral Shrivastava
Aviral.Shrivastava@asu.edu

Department of Computer Science and Engineering,
School of Computing and Informatics,
Arizona State University, Tempe, AZ 85281, U.S.A

Nikil Dutt
dutt@ics.uci.edu

ACES Lab, CECS
School of ICS, UCIrvine, CA92617

ABSTRACT

Customizing the bypasses in pipelined processors is an effective and popular means to perform power, performance and complexity trade-offs in embedded systems. However existing techniques are unable to automatically generate test patterns to functionally validate a partially bypassed processor. Manually specifying directed test sequences to validate a partially bypassed processor is not only a complex and cumbersome task, but is also highly error-prone. In this paper we present an automatic directed test generation technique to verify a partially bypassed processor pipeline using a high-level processor description. We define a fault model and coverage metric for a partially bypassed processor pipeline and demonstrate that our technique can fully cover all the faults using 107,074 tests for the Intel XScale processor within 40 minutes. In contrast, randomly generated tests can achieve 100% coverage with 2 million tests after half day. Furthermore, we demonstrate that our technique is able to generate tests for all possible bypass configurations of the Intel XScale processor.

1. INTRODUCTION

Register bypasses or forwarding paths improve the performance of a processor by eliminating certain data hazards in pipelined processors [1]. With bypasses, additional data paths and control logic are added to the processor so that the result of an operation is available for subsequent dependent operations even before it is written to the register file. Although complete bypassing can yield the best possible performance, it incurs significant overheads on the cycle time, wiring area, and the power consumption of the processor. In embedded systems where power, area, and complexity are as critical as performance, partial bypassing is a popular approach to achieve increased performance at the cost of modest overheads [2], e.g. the popular Intel XScale microarchitecture [5] implements a partially bypassed pipeline.

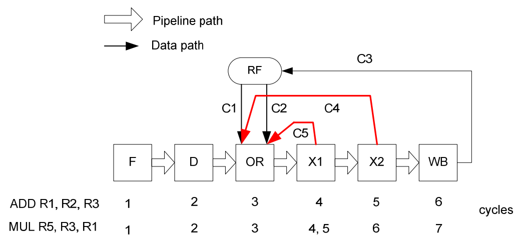

Figure 1: A simple pipelined architecture with bypasses

Figure 1 shows an example of a 6-stage, partially bypassed processor. The processor pipeline contains only two bypasses C4, and C5. The bypass C5 connects X1 pipeline stage (*Execution stage 1*) to the second operand of the OR pipeline stage (*Operand Read*), and C4 is a bypass from the X2 pipeline stage (*Execute stage 2*) to the first operand of the OR pipeline stage. Thus in the processor pipeline in Figure 1, the first operand can be read only from RF (*Register File*) or the X2 pipeline stage. A fully bypassed processor would have 6 bypasses, 2 from each of the units X1, X2, and XWB, while a non-bypassed processor will have 0 bypasses.
Since altering the bypass configuration does not affect the Instruction Set Architecture of the processor, therefore embedded processor designers often customize the bypasses between generations of a processor in order to tune the processor for the desired power/performance requirements. However, there are no existing techniques to automatically generate the test cases to verify the functional and timing correctness of a partial bypassed processor, and they have to be specified manually. Manually generating these test cases is not only a very complex and time-consuming task, it is highly error-prone.
The main challenge in generating directed test cases for a partially bypassed processor is that the test cases should verify that the bypass configuration in the implementation is exactly same as in the specification. This requires ensuring that i) bypasses absent in the specification are actually absent in the implementation, and ii) bypasses present in the specification are indeed present in the implementation. Any mismatch between the bypass configuration in the specification and the bypasses implemented can cause timing as well as functional errors. While it is absolutely necessary to detect and correct any functional faults, it is very important to correct and detect timing faults to be able to meet the power, performance constraints of the design. Existing techniques [4][12] only consider generating test cases to confirm the absence of bypasses, and thus fail to fully validate the design.
Generating tests to check for the presence and absence of bypass necessarily requires the key capability of accurate pipeline hazard detection in partially bypassed processor pipeline. In other words, given a sequence of instructions, we should be able to determine whether there will be a pipeline hazard or not? If there is going to be a hazard, then a check to detect the presence of a hazard (or a pipeline stall) should be generated, while if there is no hazard, then a check to detect the absence of a hazard (or that the pipeline did not stall) should be generated.
However existing pipeline hazard detection mechanisms use a *constant operation latency* based model and are therefore unable to accurately detect pipeline hazards in a partially bypassed processor pipeline [16]. In a partially bypassed processor pipeline, the concept of *operation latency* is ill-defined and accurate pipeline hazard detection requires not only detailed information about the structure of the pipeline, the flow of operations in the

pipeline, the bypass configuration, dependent operations and the position and register information of the dependent operands. Shrivastava et al. [14] proposed the concept of Operation Tables to model partially bypassed processor pipelines. An Operation Table is a unified representation of the structure of the processor and the register information of the operations. Operation Tables of the operations in a given schedule can be combined to accurately detect all the pipeline hazards, when the schedule is executed on the given processor pipeline model. In [15], the concept of Operation Tables was used to generate code for partially bypassed processors and achieve performance improvements. Furthermore, [15] demonstrated that customizing bypasses in a processor pipeline is a very lucrative way to achieve power performance tradeoffs without modifying the instruction set architecture of a processor. Thus designers are inclined and are designing partially bypassed processor pipeline, however, challenge still remains in verifying the correctness of a partially bypassed processor pipeline.

In this paper we automatically generate test cases to verify the functionality and the timing correctness of the processor pipeline from a high level processor description. We specify the processor architecture, including the bypass configuration in a high-level Architecture Description Language (ADL) [3], and generate Operation Tables from it. We then propose a fault model for partially bypassed pipelines, and derive a coverage metric for it. Using the Operation Tables we generate 107,074 directed tests to achieve 100% coverage on the fault in the Intel XScale processor pipeline within 40 minutes. In contrast, randomly generated tests can achieve 100% coverage after 2 million tests with half day. Furthermore, we change the bypass configuration of the XScale and demonstrate that approach can be used to generate test cases for all bypass configurations in a reasonable amount of time.

2. RELATED WORK

2.1 Partial Bypassing

Bypasses have been widely used in pipeline processor design to improve the performance of a processor [1][9]. However, the performance improvement due to full bypassing may be accompanied by a significant increase in the cycle time, chip area, energy consumption, wiring congestion, and design complexity. Partial bypassing has therefore been proposed to remove several low utilization bypasses from a design in order to reduce the power, cost and area of the design without significantly affecting the performance [10]. PBExplore [14] is a framework to explore the power-performance tradeoffs of bypass configurations, and ultimately design the bypass configuration of a processor. In PBExplore, authors describe a retargetable compiler generated code for the given bypass configuration using *Operation Tables* [15][16]. The executable generated is then simulated on a cycle-accurate simulator and a power simulator, also parameterized on the same processor description to estimate the performance and the power consumption of the processor with the given bypass configuration. This accurate evaluation of each bypass configuration enables the designer to choose and implement the appropriate bypass configuration. However, no method has been proposed to verify the design of a partially bypassed processor.

2.2 Processor Pipeline Test Generation

The field of test generation for processor pipeline verification has been extensively explored.

2.2.1 Test Generation for Instruction Set Architecture

Early works concentrated on generating test patterns for the instruction set architecture of a processor. Aharon et al. [7] and Fine et al. [8] proposed a test program generation methodology for testing the instruction set architecture of processors. However, the instruction set description does not capture the bypasses in a processor. Furthermore the presence/absence of bypasses do not affect the instruction set architecture of a processor. As a result, these approaches can not generate directed tests for testing the processor bypasses.

2.2.2 Test Generation for the Microarchitecture

The next generation of works on processor pipeline test generation focused on generating tests for the microarchitecture of processors. Shen et al. [11] extract an abstract FSM model from the processor HDL description. However, they generate tests from implementation, and not a specification. As a result if the implementation has a different but correct bypass configuration, it will not be able to detect an error in the specification. Furthermore, it will not be possible to generate directed tests for bypasses, as it is very difficult to isolate the bypasses in a HDL description of a processor.

Iwashita et al. [12] and Ur et al. [13] describe the processor microarchitecture in a high-level description. They then transform the pipeline description into a FSM model. The states relate to the units and the different types of instructions that it can hold. They then generate paths to cover every transition in the FSM. However, all three approaches cannot scale with the microarchitectural complexity and are therefore unusable for any realistic microarchitecture. Furthermore, they cannot generate directed tests for bypasses, because they abstract away the microarchitectural components and generate tests from a state machine, from where it is not possible to identify and isolate bypasses.

2.2.3 Directed Test Generation

Realizing that the number of tests required for complete testing a microprocessor is very large, and even impractical, recent approaches have focused on generating directed tests. Directed tests verify certain microarchitectural feature or property, and provide a quantitative handle on the coverage of the tests. Mishra et al. [4] generate directed tests from a high-level processor description in the EXPRESSION ADL [3]. However, they do not model bypasses in their ADL description, and are therefore unable to generate test cases to verify a bypassed microarchitecture. Further, they propose to find out the set of operation sequences which can cause the stall (*Stall Set*) in the pipelined architecture, and check the correctness of the functional results. They do not generate tests for the (*Activate Set*), which implies they do not test for the presence of bypasses. However, any inconsistency between the bypasses and the bypass control logic may result in a timing or functional error.

To conclude, none of the previous approaches model a partially bypassed processor pipeline and cannot generate directed tests to test a partially bypassed processor pipeline in a realistic machine.

In this paper, we explicitly model partial bypasses in the architecture description language (ADL). We use Operation Tables to provide the capability to generate directed tests to verify for both the presence as well as the absence of the bypasses. We then propose fault models to exhaustively test the bypass configuration. However since the test cases are prohibitively large, dependent on the implementation style, we propose a constrained fault model to drastically reduce the number of tests. Our experimental results to generate tests for the Intel XScale microarchitecture demonstrate that our approach can be utilized to automatically generate test cases and verify the bypass configuration of a realistic processor in a reasonable time.

3. OUR APPROACH

Figure 2 outlines our approach for test generation for a partially bypassed processor. We describe the processor microarchitecture at a high level of abstraction in an Architecture Description Language (ADL). We develop fault models for partially bypassed processor pipelines, and define coverage metrics using these fault models. The test generator takes the processor description and the fault model/coverage metric as an input and generates directed test to cover the fault model, and verify the partially bypassed processor pipeline.

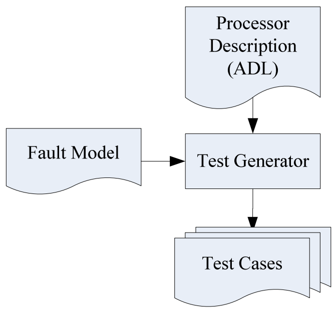

Figure 2: ADL driven bypass test generation methodology

3.1 Processor Description

We describe the partially bypassed processor microarchitecture at a high level of abstraction. Figure 3 shows the 7-stages pipelined processor architecture at our level of representation. F, D, OR, X1, X2 are the pipeline stages in the processor.

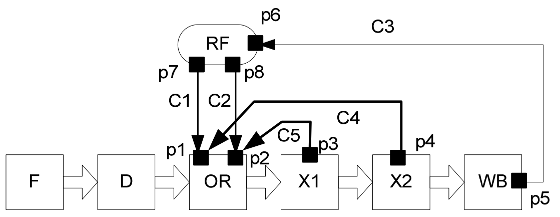

Figure 3: An example of a bypassed architecture

The flow of operations is explicitly modeled in the pipeline. Each pipeline unit contains a list of operations that it supports, and the time they spend in the unit. The path of each operation in the pipeline can then be derived, and is represented by the block arrows in Figure 3. Pipeline units can read/write operands using read/write ports. A port may be connected to other ports in the Register File (RF), or other pipeline units via explicit directed connections. Bypasses are modeled simply as a connection between a write port on a pipeline unit and a read port on the OR pipeline unit. Thus the first operand in OR can be read from pipeline unit X2 via bypass C4, but not from X1. A more detailed, graphical model of the processor description in the ADL is presented in [17].

3.2 Operation Tables

An *Operation Table* (OT) models the execution of an operation in the processor. As defined in Table 1, an OT is a DAG (Directed Acyclic Graph), whose nodes contain information about the pipeline unit in which the operation is being executed, and the operands are being read, written, and bypassed in that execution cycle. The edges of the DAG define a temporal ordering on the nodes.

Table 1: Operation Table Definition

OperationTable	:= {otCycle}
otCycle	:= unit ros wos bos dos
ros	:= ReadOperands {operand}
wos	:= WriteOperands {operand}
bos	:= BypassOperands {operand }
dos	:= DestOperands {regNo}
operand	:= regNo {regConn }

Table 2 shows the OT of the operation *ADD R1, R2, R3* on the processor pipeline shown in Figure 3. Operation Tables have been used to accurately detect all pipeline hazards when a given schedule of instructions executes on a given processor pipeline, even in the presence of partial bypassing. In this work, we use the concept of OTs to automatically generate test sequences to verify the functional and timing correctness of a partially bypassed processor pipeline.

Table 2: Operation Table of ADD R1 R2 R3

```
1 F
2 D
3 OR
          ReadOperands
              R2
                  C1, C4
              R3
                  C2, C5
          DestOperands
              R1, RF
4 X1
          WriteOperands
              R1
                       C5
5 X2
          WriteOperands
              R1
                       C4
5 WB
          WriteOperands
              R1
                  C3
```

3.3 Bypass Test

A bypass test is an ordered sequence of operations, which *will try* to use a bypass. A bypass test necessarily contains two operations, a Bypass Producer Operation (*BPO*), and a Bypass Consumer Operation (*BCO*). A BPO of a bypass is an operation, which can generate a bypass value from the operation in the unit at the source-end (write port) of the bypass. A BCO of a bypass is an operation that can receive a bypass value from the operation in the unit at the destination-end (read port) of the bypass.

```
// Part 1. Initialize the registers
ADDI R2 R0 3
ADDI R3 R0 5
ADDI R6 R0 5

// Part 2. Excite the bypass from X1 to OR
MUL R1 R2 R3
ADD R6 R6 R6
ADD R5 R1 R3

// Part 3. Check timing and functional correctness
if (stall) JUMP ERROR
if (R5 != 15) JUMP ERROR
SUCCESS;
```

Figure 4: A bypass test to verify the presence of the bypass C5

In order to exercise a bypass, the operations, BPO and BCO should be correctly separated. For example, in Figure 1, to exercise the bypass C5, the difference between the schedule times of the BPO and BCO should 1 if the BPO is an ADD operation, but it should be 2, if the BPO is a MUL operation. This is because MUL takes two cycles to execute in X1, and it can bypass the result only after it has finished execution.
Figure 4 shows an outline of a directed test case to verify the presence of bypass C5. It comprises of 3 steps: i) Initializing the register values (R2 = 3, R3 = 5). ii) Exciting the bypass C5. The operation *MUL R1 R2 R3* can write the value R1 in the bypass C5, two cycles after it is issued, and that is exactly when we have scheduled the operation *ADD R5 R1 R3*, which can use the value of R1 as the first operand through the bypass C5. The operation *ADD R6 R6 R6* is an independent operation, which will ensure separation of the two operations without any stall. iii) Check the timing and functional correctness of the execution.

3.4 Fault Model

Bypass design consists of two parts, first is the data path that actually connects a write port of any unit to a read port in the OR pipeline stage, and second is the control logic to enable transfer of values between operations, and the corresponding pipeline stall logic. Functional and timing faults can be due to incorrect implementation in the data path or in the control logic.

3.4.1 Fault model for the presence of bypasses

The implementation of a bypass that is present in the specification is erroneous if on exercising the bypass, the output result is incorrect, or if a pipeline stall occurs.

To find such faults, we define an *Activate Set* for a bypass *b*, ACT_b as the set of all possible operation sequences that activate the bypass *b*. The activation set of all the bypasses $ActivationSet = \cup_{\forall b} ACT_b$. Let us assume a sequence of operations ops_{act} can cause an activation *act* (i.e., $act \in ActivationSet$). Let val_{act} denote the result of computing the operation sequence ops_{act}, and N_{stall} represent the number of stalls. The val_{act} has n components ($\cup_{k=1}^{n} \mathrm{val}_{\mathrm{act}}^{k}$). In the fault-free case, all the destinations will contain correct values, i.e., $\forall k\ dest_k = val_i^k$ and no stall occurs. Under a fault, at lease one of the destinations will have incorrect value, or an unexpected stall occurs, i.e., $(\exists k\ dest_k \neq val_i^k)$ or $(N_{stall} \neq 0)$.

3.4.2 Fault model for the absence of bypasses

Similarly the implementation of a bypass that is absent in the specification is erroneous if on exercising the bypass, the output result is incorrect, or if a pipeline stall *does not* occur.

To find these faults we define a stall set for the OR unit (SS_{or}) as all possible ways to stall the OR unit. A sequence of operations ops_{ss} is a sequence which can cause a stall *ss* in OR unit ($ss \in SS_{or}$). Let val_{ss} denote the result of computing the operation sequence ops_{ss}. The val_{ss} has n components ($\cup_{k=1}^{n} val_{ss}^{k}$). In the fault-free case, all the destinations will contain correct values, i.e., $\forall k\ dest_k = val_i^k$ and a stall occurs. Under a fault, at lease one of the destinations will have incorrect value or no stall occurs, i.e., $(\exists k\ dest_k \neq val_i^k)$ or $(\mathrm{N_{stall}} = 0)$.

3.5 Coverage Metric

Given the fault model for the presence and the absence of the bypasses, we now estimate how many test sequences will be there to exhaustively test functionality and the stall logic of the processor pipeline. The number of unique test sequences to excite a bypass is $N_{possible}^{b} = n_{BPO}^{b} \times n_{BCO}^{b}$, where n_{BPO}^{b} is the number of different BPO operations of the bypass *b*, and n_{BCO}^{b} is the number of different BCO operations of bypass *b*.

Thus, the total number of possible test sequences is the number of all combinations of BPOs and BCOs shown as follows,

$$N_{possible} = \sum_{b \in B} N_{possible}^{b} = \sum_{b \in B} n_{BPO}^{b} \times n_{BCO}^{b} \quad (1)$$

where *B* is the set of all the bypasses that are present in the processor pipeline. Suppose the set of all the operations in a processor is *O*. Further suppose that operation $o_i \in O$ has $o_i.nd$ destinations, then $n_{BPO}^{b} = \sum_{o_i \in O} o_i.nd$ Similarly if operation $o_i \in O$ has $o_i.sd$ register sources, then $n_{BCO}^{b} = \sum_{o_i \in O} o_i.sd$ ·
Thus, formula (1) can be transformed to formula (2) as follows.

$$N_{possible} = \sum_{b \in B} \sum_{o_i \in O} o_i.nd \times o_i.sd \quad (2)$$

which is bounded by $|B| \times |O|^2 \times nd \times sd$, where $nd = \underset{o_i \in O}{MAX}\ o_i.nd$ and $sd = \underset{o_i \in O}{MAX}\ o_i.sd$ ·

The coverage metric derived from this fault model is $C_{presence} = \sum_{b \in B} N_{try}^{b} / N_{possible}^{b}$, where N_{try}^{b} is the number of unique operation sequences that excite the bypass *b* have been tried. This coverage metric provides a quantitative estimate of the exhaustive test coverage that has been achieved.

3.6 Test Generation

```
TestGenerate()
01: for each bypass b ∈ B
02:    for each operation bco ∈ BCO(b)
03:       for each operation bpo∈ BPO(b)
04:          // generate tests for (b, bpo, bco)
05:          dopnds = bpo.OT.opndsThatWriteTo(b.srcPort);
06:          sopnds = bco.OT.opndsThatReadFrom(b.destPort);
07:          for each dopnd ∈ dopnds
08:             for each sopnd ∈ sopnds
09:                t1 = bpo.OT.getWriteCycle(dopnd, b);
10:                t2 = bco.OT.getReadCycle(sopnd, b);
11:                d = t1-t2;
12:                GenBypassTest(b, bpo, dopnd, bco, sopnd,
d);
13:             end for
14:          end for
15:       end for
16:    end for
17: end for
```

Figure 5: Directed test generation for the fault model

Figure 5 shows the algorithm to generate tests for our bypass fault model. The first set of loop (lines 01-03) find out the BPOs and BCOs for each bypass b. B is the set of all bypasses present in the processor. Next, bypass tests need to be generated to test this combination of *bpo* and *bco* trying to activate the bypass *b*.

```
SetBCOBPORegisters (b, bpo, dopnd, bco, sopnd, d))
01: depReg;
02: regsToInit;
03: // set registers of bpo
04: for each opnd ∈ bpo.destOperands U bpo.srcOperands
05:    if (opnd.type = REGISTER)
06:       r = getNewRegister();
07:       bpo.setOpnd(opnd, r);
08:       if (opnd = dopnd) depReg = r;
09:       regsToInit += r;
10:       end if
11:    end if
12: end for

13: // set registers of bco
14: for each opnd ∈ bco.destOperands U bco.srcOperands
15:    if (opnd.type = REGISTER)
16:       if (opnd = sopnd) r = depReg
17:       else r = getNewRegister();
18:       end if
19:       bpo.setOpnd(opnd, r);
20:       regsToInit += r;
21:    end if
22: end for

23: // set registers of nop
24: for each opnd ∈ nop.destOperands U nop.srcOperands
25:    if (opnd.type = REGISTER)
26:       r = getNewRegister();
27:       nop.setOpnd(opnd, r);
28:       regsToInit += r;
29:    end if
30: end for
```

Figure 6: The function *SetBCOBPORegisters* sets the register numbers of the operands of BPO and BCO

A bypass test will be generated for each destination operand *dopnd* of *bpo* that can write into the bypass b, and each source operand *sopnd* of *bco* that can read from bypass *b* (Lines 08-09). We use Operation Tables to find out *dopnd* (Line 05) and *sopnd* (Line 06). We use the Operation Tables of *bpo* to find out the cycle in which *dopnd* writes into the bypass *b*, and the Operation Table of *bco* to find out the cycle (*t2*) when *sopnd* reads from the bypass *b* (Lines 09-10). The required separation between the *bpo* and *bco* should be *d=t1–t2* (Line 11). Finally, we call the function *GenerateBypassTest* that will generate one directed test sequence.
The function **GenBypassTest** will generate a test sequence to test the given bypass *b*. The key function here is to set the register values of the *bco*, *bpo* and *nop* operations. This functionality is achieved by the function **SetBCOBPORegisters,** described in Figure 6. The function has to make sure that the dependent operand of the *bpo*, *dopnd* and *bco*, *sopnd* should be the same register. This is achieved using the variable *depReg* in Figure 6. This function should also ensure that there are no other data hazards. Therefore it should provide fresh registers for all the other operands of operations. After setting the operands, Part 2 of the test shown in Figure 4 can be easily generated.
The function *SetBCOBPORegisters* in Figure 6 returns the set of registers that need to be initialized, *regsToInit* in Part 1 of the bypass test in Figure 4. The stall checking sequence, i.e., Part 3 of bypass test in Figure 4 is an important part of the test sequence to verify the timing the correctness of the bypass configuration, i.e., to check the occurrence of stalls. Figure 7 shows an example of a generated stall checking sequence for Intel Xscale [5] architecture. The stall checking sequence is very architecture specific, and requires hardware support. The Intel XScale processor contains hardware counters that count the number of stalls. These counters can be reset and read, enabling us to count the number of stalls in a sequence of instructions.

```
mrc p14, 0, r5, c2, c0, 0
cmp r5, #0
bne ERROR_S
…
```

Figure 7: Stall check sequence for the Intel XScale

4. EXPERIMENTS

To demonstrate the usefulness of our approach, we apply it to the partially bypassed Intel XScale [5] processor, whose pipeline diagram is shown in Figure 8. XScale processor implements the ARM instruction set, and is a popular embedded processor for wireless and handheld devices. XScale has three execution pipelines, the X pipeline (units X1, X2, and XWB), the D pipeline (units D1, D2, and DWB), and M pipeline (units M1, M2 and MWB). We assume that the 7 pipeline stages X1, X2, XWB, M2, MWB, D2 and DWB *can* bypass to all the 4 operands in the RF stage. Thus there are $7 \times 4 = 28$ different bypasses that are possible.

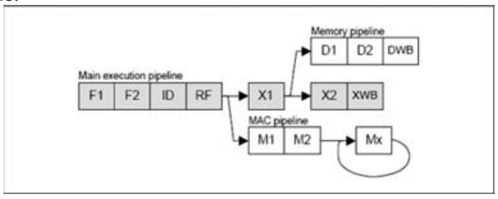

Figure 8: XScale 7-stage super-pipeline

We describe the ARM ISA and the XScale microarchitecture in the EXPRESSION processor-Architecture Description Language (ADL), and automatically generate Operation Tables from there. We use the Operation Tables and our scheme of automated test generation described in Section 3 to generate test sequences to test the bypass and stall logic of the processor. To fully demonstrate the applicability and capability of our approach we present three sets of experiments.

4.1 Directed Test Generation

We generated test sequences to test the partially bypassed XScale processor pipeline. A total of 80,516 test sequences were generated for verifying the presence of bypasses, and 26,558 test sequences were generated to verify for the absence of the bypasses. The test generation time was approximately 40 minutes on a 3.2 GHz Pentium-4 processor, 1G MB Memory PC. Since many full processor-level RTL simulators operate at about 1 instruction per second, we estimate that our generated tests can verify a detailed RTL model of a processor in about a day. This shows that our technique can verify the bypass and stall logic of a real processor pipeline in a reasonable amount of time.

4.2 Comparison with random test generation

To demonstrate the goodness of our fault model and coverage metric, we randomly generated dependent operation sequences that are separated by a random number of NOPs and measured their coverage. Note that the maximum number of NOPs is less than maximum execution time of the operation in the none-bypassed architecture. Figure 9 shows the results which compare our direct test generation with the randomly test generation. The random test generation spent about half day to generate 2 million tests to achieve 100% coverage for our fault model, while our method of directed test generation can achieve 100% coverage using about 107,074 tests within 40 minutes.

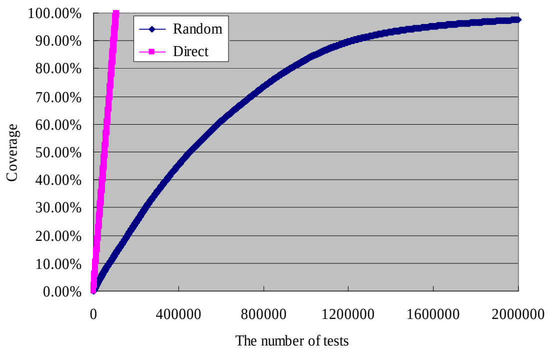

Figure 9 Randomly V.S. Directed Test Generation

4.3 Other Bypass Configurations

To demonstrate that our technique can successfully generate tests for various bypass configurations, we change the bypass configuration in the Intel XScale and generate tests for each of them. The Intel XScale has 28 different bypasses. This means that there can be up to 2^{28} different bypass configurations. While our technique can generate test sequences for all these cases, here we present two important feasible set of explorations.
First we automatically generate test sequences by varying the bypass sources. We vary whether bypasses exist from a pipeline unit. Since there are 7 units that can generate a bypass value, therefore there can be $2^7 = 128$ bypass configurations. We number the bypass configurations using a 7-bit number where the bits from the right to the left identify whether the bypasses from units X1, X2, XWB, M2, MWB, D2 DWB are present or not. Figure 10 plots the number of test sequences generated to verify for the presence and the absence of the bypasses. We note that the number of test sequences required to achieve 100% coverage across all bypass configurations is bounded to less than 120,000 tests.

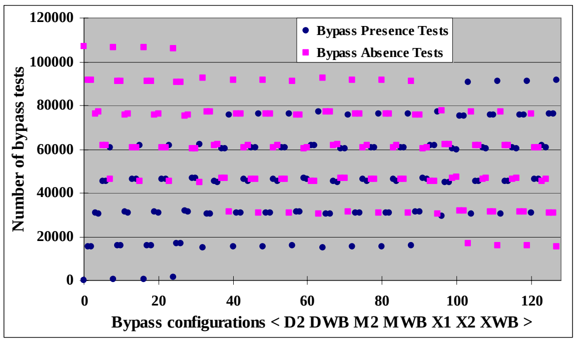

Figure 10: Number of tests while exploring bypass sources

In the second set of experiments, we vary the bypass destinations. We vary whether the bypasses exits to the ports in the RF unit. Since there are 4 ports in the RF unit, there are 2^4 = 16 bypass configurations.

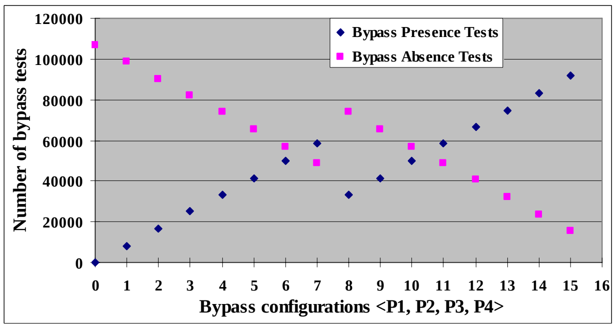

Figure 11: Number of tests while exploring bypass destinations

As before we define a bypass configuration by a 4-bit number, with the bits from right to left identifying whether the bypasses to port1, port2, port3 and port4 of the RF unit are present or not. Figure 11 plots the number of test sequences required to verify the presence and absence of bypasses in these bypass configurations. Again, we note that the number of directed test sequences generated by our approach is bounded to less than 110,000 to achieve 100% coverage of the fault model.

These results demonstrate that we can *successfully, automatically, and efficiently* generate bypass tests to test a partially bypassed processor pipeline.

5. SUMMARY

Partial bypassing is an attractive option in embedded processors to achieve power, performance and complexity tradeoffs. While previous approaches have demonstrated the need and usefulness of partially bypassed processors, and even have suggested techniques to explore and design them, no method has been suggested to verify their correctness. Existing test generation schemes are unable to generate tests for partially bypassed processors, chiefly because i) they cannot model partial bypassing in processors, and consequently, and ii) they do not generate test cases to verify for the presence of bypasses. Specifying bypass tests by hand is not only a time consuming and cumbersome task, it is also highly error-prone. In this paper, we presented a fault model for partially bypassed processors and derived coverage metric. We then propose a directed test generation scheme, which needs 40 minutes to generate 107,074 tests to fully cover the faults in a partially bypassed Intel XScale processor. In contrast, random test generation scheme can achieve 100% coverage after 2 million tests with half day, demonstrating the goodness of our fault model and coverage metric. Finally, we vary the bypasses in the Intel XScale processor and show that we can generate test sequences for all bypass configurations, demonstrating the efficacy of our approach. In the next step, we will confirm the effect of our approach by randomly inject bugs into implementation, and see the difference between fault models based on specification and implementation.

6. REFERENCES

[1] P. Hennesy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publisher, 1990.

[2] E. McLellan. *The Alpha AXP Architecture and 21064 processor.* IEEE Micro, June 1993.

[3] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Nutt and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retarget- ability. In Proceedings of Software and Compilers for Embedded Systems (SCOPES), 2001.

[4] P. Mishra, N. Dutt, *Functional Verification of Programmable Embedded Architectures- A Top-Down Approach*, Springer, 2005.

[5] http://www.intel.com/design/intelxscale/273436.htm, *Intel XScale Microarchitecture Programmers Reference*, 2001.

[6] *Intel XScale Microarchitecture for the PXA255 Processor User's manual,* pages 88-95, 2003.

[7] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, and G. Shurek. *Test Program Generation for Functional Verification of PowerPC Processors in IBM.* In Proceedings of Design Automation Conference (DAC), pages 279-285, 1995.

[8] S. Fine and A. Ziv. Coverage Directed Test Generation for Functional Verification using Bayesian Networks, In Proceeding of Design Automation Conference (DAC), pages 286-291, 2003.

[9] S. Muchnick, *Advanced Complier Design and Implementation.* Morgan Kaufmann Publishers, 1998.

[10] K. Fan, N. Clark, M. Chu, K. V. Manjunath, R. Ravindran, M. Smelyanskiy, and S. Mahlke. *Systematic Register Bypass Customization for Application-Specific Processors.* In Proc. of ASSAP, 2003.

[11] Shen, J. and J. Abraham, *An RTL Abstraction Technique for Processor Micorarchitecture Validation and Test Generation*, J. Electronic Testing: Theory&Application 16 (1999), pp. 67–81.

[12] H. Iwashita, S. Kowatari, T. Nakata and F. Hirose. *Automatic test pattern generation for pipelined processors.* In Proceedings of International Conference on Computer-Aided Design (ICCAD), pages 580-583, 1994.

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

[14] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau., *PBExplore: A Framework for Compiler-in-the-Loop Exploration of Partial Bypassing in Embedded Processors,* In Proceedings of the International Conference on Design Automation and Test in Europe, DATE 2005, pages 1264-1269, 2005.

[15] S. Park, A. Shrivastava, E. Earlie, A. Nicolau, Y. Paek, N. Dutt., *Automatic Generation of Operation Tables for Fast Exporation of Bypasses in Embedded Processors,* In Proceedings of the International Conference on Design Automation and Test in Europe, DATE 2006, pages 1197-1202, 2006.

[16] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. *Operation tables for scheduling in the presence of incomplete bypassing.* In CODES+ISSS '04, pages 194–199, New York, NY, USA, 2004. ACM Press.

[17] P. Mishra, A. Shrivastava, N. Dutt, *Architecture Description Language (ADL)-Driven Software Toolkit Generation for Architectural Exploration of Programmable SOCs*, In Proceedings of the 41st Annual Conference on Design Automation, DAC '04. 2004.