UNIVERSITY OF CALIFORNIA
Santa Barbara

Enhancing the capability of constrained random test program generators via learning and test program filtering

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Vinayak Kamath Kasargod

December 2014

The Dissertation of
Vinayak Kamath Kasargod is approved:

Professor Forrest Brewer

Professor Margaret Marek-Sadowska

Dr. Eli Chiprout

Dr. Sankar Gurumurthy

Professor Li-C Wang, Committee Chairperson

September 2014

Enhancing the capability of constrained random test program generators via learning and test program filtering

by

Vinayak Kamath Kasargod

Dedicated to my parents,

Revathi Kamath and Ramesh Kamath

Acknowledgements

First off, I express my deepest gratitude to my advisor Prof. Li-C Wang. He has been more than a research advisor throughout my Phd. He has been a guide, a mentor and huge inspiration. He has helped me grow into not just a strong and independent researcher, but also into a refined, more virtuous human being. This dissertation would not have been possible without him. I would like to extend my heartfelt gratitude and appreciation for his guidance, support and encouragement.

I would like to thank my committee members, Prof. Forrest Brewer, Prof. Margaret Marek-Sadowska, Dr. Eli Chiprout and Dr. Sankar Gurumurthy for reviewing my dissertation and being on my committee. The inputs that you provided during my qualifying exams were valuable.

I would like to acknowledge the contribution of all my colleagues at AMD. I owe special thanks to my manager, Farhan Rahman, for giving me the opportunity to intern at AMD. He has been extremely supportive of my research and always been a big source of inspiration to trying out new ideas. I want to acknowledge the contribution of my colleagues, especially Steve Havlir and Todd Dukes, for having been patient and dedicating a lot of their time to helping me with my research. A big thanks to Nick Kisler, Todd Foster, Larry Smith, John King and Sarang Nadkarni for their help. I am grateful to Sankar Gurumurthy for providing me with technical help and guidance from time to time, and for agreeing to be on my committee.

I want to thank my friends Sagar, Sanjay, Ashish, Abhishek, Akella, Rakesh, Amod, Ajay, Aniruddha, Mathangi, Mahima, Bhargavi and Tejeswini - who have been with me since my undergrad days; the whole Bangalore SPIC MACAY team - Mythraeyi, Kurup, Navneeta, Supriti, Lakshmi, Chandrani and Deepak and my good old school friend Jyothsana for being just a phone call. I would also like to acknowledge my buddies from Santa Barbara/roommates at some point - Santosh, Harsh, Shashank, Madhukar, Sujay, Vineeth, Srinath, Vikram and Karthik; the 'Dance pe Chance' group - Neeraja, Ram, Palak T, Palak P, Anand and Ashwin and my friends/colleagues from Austin - Kapil, Sarang, Sheetal, Mandar, Aarti, Sid, Shoeib, Akshay and Santosh. All of you have been a tremendous support all along. My days in college and grad school would not have been the same without them. And I mean it in a good way.

My current and ex-colleagues Wen Chen, Nik Sumikawa, Po-Hsien Chang and Gagi Dramanac have been helpful beginning from the day I started my Phd. I would like to thank them for all the insightful discussions and meeting we have had over the years.

A special thanks to all my music gurus, Pt. Homnath Upadhyaya, Ms. Suniti Bergman and Ms. Manasi Joshi-Singh for giving me the gift of music. Thanks to you, I have derived immense pleasure, peace, and satisfaction through music. It has been a great companion all these years; and I am sure it that it will be in the years to come.

I want to acknowledge the love, support and encouragement that I received from my parents and siblings. Good education has always been the top-most priority for my father and I would not have made it this far if not for his determination to see me succeed. My mother and sister have been the supporting pillars of my life. I am indebted to each one of you for life for the sacrifices that you have made and for all that you have provided me with. I also grateful to my entire extended family for their unconditional love and support. A special thanks to Swati for her love, support and patience. This journey is too difficult to do solo. It is her encouragement, positivity and faith in me that kept me going despite all the hardships.

A big thanks to everyone of you who have directly or indirectly, intentionally or unintentionally, willingly or unwillingly helped me in this endeavor. I sincerely apologize in advance for any inadvertent omissions.

Curriculum Vitæ

Vinayak Kamath Kasargod

Education

2014	Doctor of Philosophy in Electrical and Computer Engineering, University of California, Santa Barbara (Expected).
2012	Master of Science in Electrical and Computer Engineering, University of California, Santa Barbara.
2008	Bachelor of Technology, National Institute of Technology Karnataka - Surathkal.

Professional Experience

2013 - 2014	Co-op, Advanced Micro Devices, Austin.
2011 – 2013	Research Assistant, University of California, Santa Barbara.
2008 – 2010	IC Design Engineer II, LSI Research India Pvt. Ltd., Bengaluru

Awards/Recognition

2014	Executive Spotlight Award, Advanced Micro Devices, Austin.

Publications

- Vinayak Kamath, Wen Chen, Nik Sumikawa, Li C Wang, ”Functional Test Content Optimization for Peak-Power Validation - An Experimental Study”, 2012 IEEE International Test Conference (ITC), Anaheim, CA, November 2012.
- Vinayak Kamath, Farhan Rahman, Li C Wang, ”Analyzing efficacy of constrained test program generators - A case study”, International workshop on Microprocessor Test and Verification (MTV) 2013, Austin, TX, December 2013.

Abstract

Enhancing the capability of constrained random test program generators via learning and test program filtering

Vinayak Kamath Kasargod

Functional verification of RTL is one of the primary and most time consuming tasks of microprocessor design. However, designs cannot be completely verified due to their large size and strict time-to-market restrictions. Formal verification and simulation-based verification both sacrifice completeness for utility. While formal verification is relegated to the verification of a part or an abstraction of the design, dynamic verification reduces complexity by restricting possible input sequences. Being more scalable, simulation-based verification has been the mainstay of functional verification.

A majority of simulated tests are created from separate, automatic, random stimuli generators based on user templates. The generated stimuli, usually in the form of assembly programs, trigger architectural and microarchitectural events. The quality of applied tests is periodically evaluated based on coverage points defined in a verification plan. On one hand, the use of randomization inevitably leaves some redundancy in the generated tests. On the other hand, the effectiveness of a generator depends on the fact that test templates are user defined. Because of these limitations, it is challenging to develop a new generator that out-performs an existing generator in every aspect. As a result, over the years multiple test generators are developed and retained, incurring tremendous overhead in maintaining the software infrastructure.

In this thesis we propose a novel methodology to improve the effectiveness of one test generator with respect to another. First, we evaluate the effectiveness of using a legacy test generator used at AMD and quantify its verification performance. We explore the differences in the design and capabilities between the legacy test generator and AMD's latest in-house x86 ISA-based test generator. We then proceed to gather experimental evidence to support our understanding. Functional coverage measurements based on an existing verification plan confirm our findings. With the exception of two cases, we find the latest test generator to have a far superior capability for verifying the features tested. The two exceptions are due to its design limitations. We propose to utilize external test filters to overcome these limitations.

We develop a test filtration approach that is independent of the test generator, to filter out ineffective tests prior to RTL simulation. We achieve this by using ISA simulation traces. We find that using a combination of ISA simulation traces and microarchitectural models is necessary to cover a wider range of coverpoints. Our work shows that by using implementation specific details for extrapolating test behavior information present in simulation traces, we can compensate for microarchitecture agnostic test generation and consequently improve the effectiveness of a test generator without modifying its design. The proposed approach expedites coverage closure by providing precise control over random test behavior. Experimental results based on the latest AMD multi-core processor design are presented to demonstrate the feasibility of our proposed approach.

Contents

List of Figures

List of Tables

Chapter 1

Functional verification of digital integrated circuits

1.1 Introduction

Functional verification of an integrated circuit is demonstrating that the design implementation is the same as the specification by checking the functional equivalence of two versions of the design. The two versions may be representations of the design at a different level of abstraction, such as behavioral and structural RTL. It can also be different versions of the design at the same level of abstraction; for example, one may be a retimed version of the other. Non-conformance of feature sets, protocols or performance parameters to the specification are regarded as functional defects or bugs. Some bugs can be fixed by software workarounds, while others may require expensive silicon revisions. A robust verification methodology is necessary at the unit, chip and system level to ensure a defect free silicon on the very first revision. Overall, functional verification takes up 30% to 50% of the development time and resources in chip design[22]. Some estimates even put it as high as 70%[21].

Despite advances in verification methodologies, tools and the availability of faster and larger compute clouds, verification is only barely keeping up with System on a Chip (SOC) design. Transistor count in SoCs is roughly doubling every two years, as projected by Moore's Law(see Fig. 1.1). The complexity of designs is increasing as more and more IPs are being integrated onto SoCs. Advances in Computer Aided Design (CAD) have enabled logic to grow at an exponential rate. A net result of all this is an increase in the size and complexity of verification space[8].

The cost of detecting and fixing bugs increases as we move down the design cycle. Post-silicon debugging is notoriously tedious and the cost of design re-spins is prohibitive. While the chips are expected to meet the prescribed quality standards, they are bound by cost and aggressive time-to-market constraints. They have to communicate with other chips and memories on a board and comply with myriad standards. Consumer products such as mobile and gaming console processors are designed under very tight schedules with approximately 18 to 24 months from design to mass production of multi-billion gate designs, even with logic reuse. Product release windows are very narrow and missing it can severely affect product sales. All these factors force verification teams to make the most efficient use of available resources to deliver a high quality design.

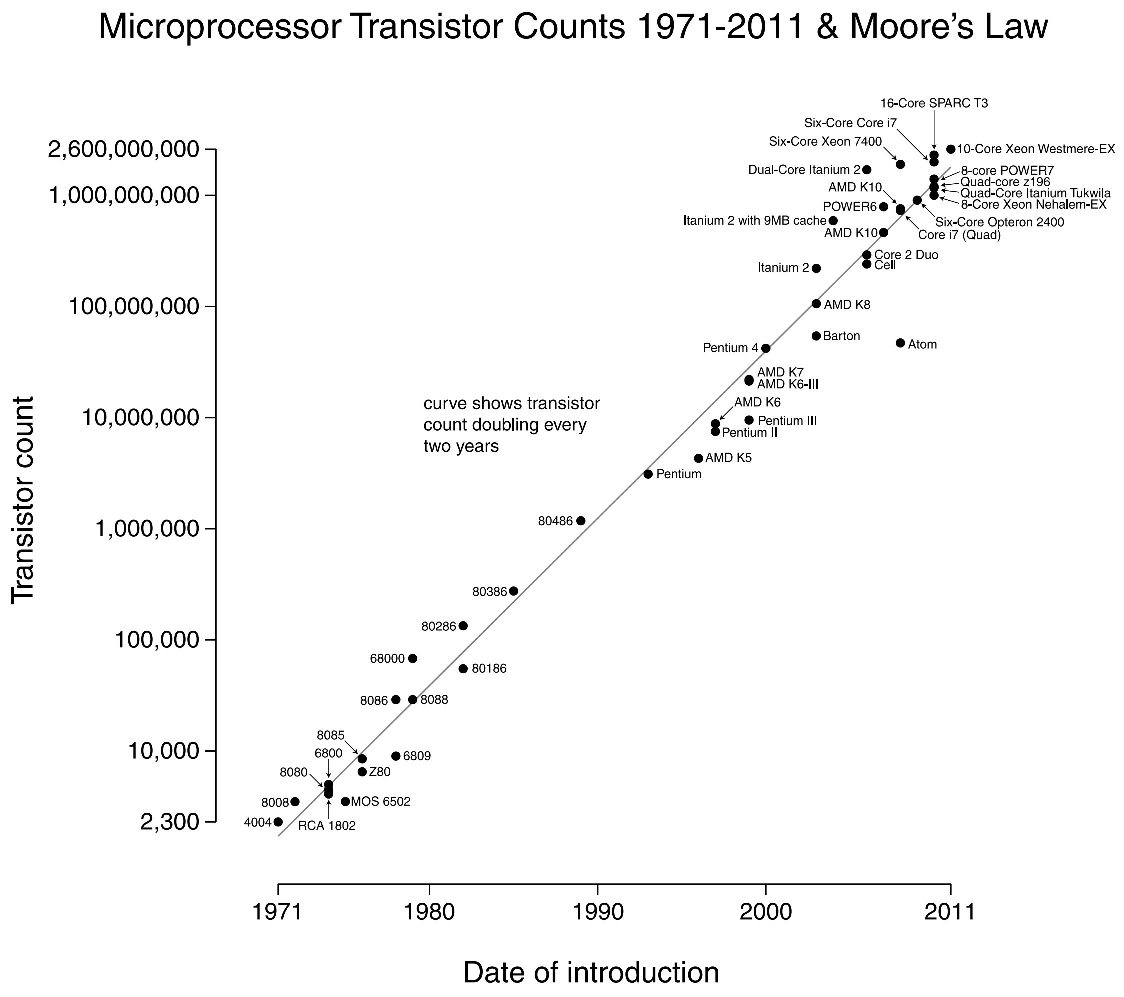


Figure 1.1: Plot of microprocessor transistor count over the years since 1971 *Source: "Transistor Count and Moore's Law - 2011" by Wgsimon*[13]

Verification is a process that is never really complete. The goal of verification is to ensure that the design is error-free. This however cannot be proved. Verification can only detect the presence of errors, but not its absence. Given enough time, verification can always find errors. For a robust verification environment, the returns on spending more time and resources are diminishing. The ability and confidence to declare verification as complete can only come with experience.

1.2 Functional verification methodology

Functional verification begins with writing a verification plan, followed by implementing the verification environment, device bring-up and device regression[36]. Fig 1.2 illustrates the components of the functional verification plan. It consists of four parts:

1. Stimulus generation
2. Design simulation environment
3. Response checking
4. Coverage measurement

The verification plan enumerates what must be verified and how. It describes the scope of the verification problem and serves as the functional specification for the verification environment. What must be verified is captured in the functional, code and assertion coverage requirements of the coverage measurement section of the verification plan. How the device is to be verified is captured in the top

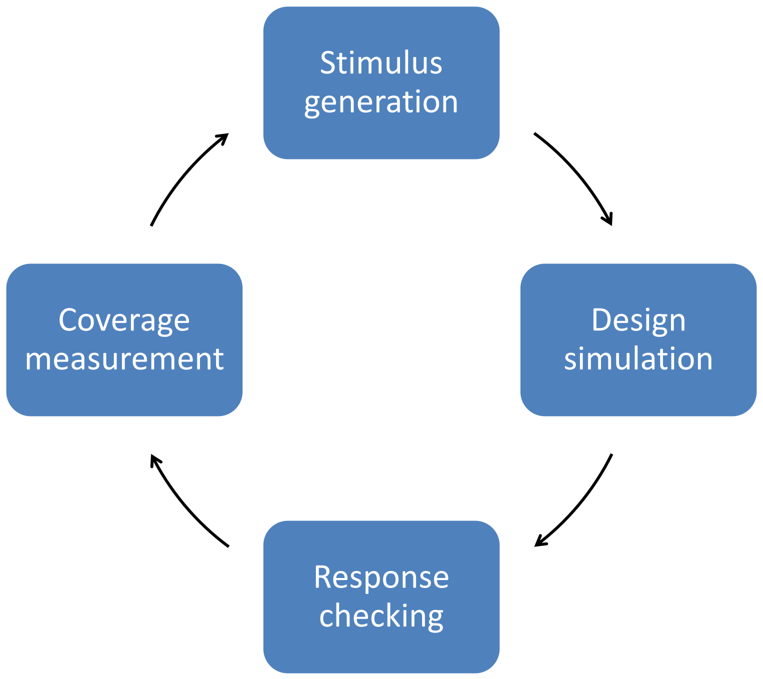


Figure 1.2: Functional verification process

and detailed-design section of each of the three aspects of the verification plan: coverage measurement, stimulus generation and response checking. The stimulus generation defines the setup needed to generate the stimuli required by the coverage section. The response checking describes the mechanisms to be used to compare the response of the design to the expected, specified response.

Once the design under test (DUT) is integrated into the verification environment, simple directed tests are run to eliminate all trivial and show-stopper bugs to bring the setup to a stable state. At this stage, bugs are easy to find; so there is no reasons to bombard the design with elaborate tests. The initial simulations exercise an extremely narrow path through the behavioral space of the device. The

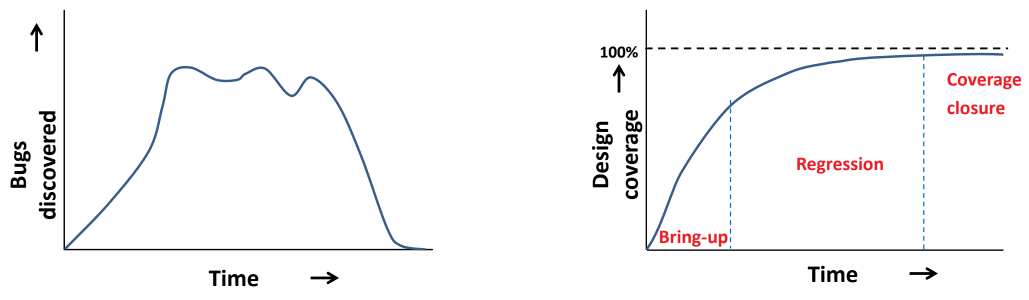


Figure 1.3: Representative plot of bug discovery rate (left) and design functional coverage (right) over the lifetime of a project

tests are commonly handwritten as they are functionally restricted in order to make it easy to diagnose a failure. By the end of design bring-up, both, the DUT and the verification environment can be subjected to a steady stream of stimuli to identify functional bugs. This is followed by extensive regressions where the design is thoroughly exercised to detect the (re-)introduction of bugs into the design implementation. This process is interleaved with means of coverage measurement. This serves as a feedback mechanism for further testing and provides a sense of verification completion. This is an extended phase that continues all the way to design tape-out.

Fig. 1.3 illustrates a typical bug discovery and coverage chart. The number of bugs discovered in a week progressively increases during device bring-up as the quality of the stimulus is improved through better test generator constraints. After the design has been rigorously tested for an extended period, the rate begins to drop. This indicates improving design health. Bugs become increasingly hard to find towards the end of a project cycle. Much like bug discovery, design coverage growth is significantly higher during bring-up. Once all of architectural and elementary design features have been tested, coverage growth begins slowing down. Coverage closure is a combination of waiving unnecessary coverage from the verification plan and creating directed tests to cover known design corners.

1.3 Optimizing functional test content

Ideally, in order to maximize the utilization of simulation cycles, we have to eliminate all stimuli which do not excite any new design behavior. To be useful, every simulated test should trigger a previously unseen design behavior. Estimating test quality without running a detailed (RTL) simulation, however, is a challenging task. Architectural coverage, for example, provides a coarse estimate of micro-architectural behavior. Since architectural simulation lacks timing information and microarchitecture information, two tests with identical architectural coverage can behave differently on RTL.

In practice, tests are generated using constrained test program generators which attempt to provide a desirable spread of stimulus across the test space. More often than not, there is no filtering mechanism to evaluate test behavior before simulation. A significant fraction of generated tests do not provide any new coverage. Since tests provide coverage beyond what is measured by the verification plan, this is considered a cost of random verification.

1.3.1 Avoiding duplication of test generation effort

It is a common practice to have multiple sources of random tests. When working on the same instruction-set architecture (ISA) for decades, newer and better random test program generators are developed from time to time. While the older test generators are trusted and proven, the newer ones offer better coverage and control. As opposed to substituting an existing tool, older tools are used as a supplement. Even though this intuitively gives a sense of better verification quality, the benefits of this verification overhead are seldom evaluated. In this dissertation, we tackle this issue by analyzing two in-house test generators (exercisers) to understand their strengths and weaknesses. We substantiate our findings with RTL simulation data from a recent x86 ISA based microprocessor core.

1.3.2 Generating targeted tests for coverage closure

Parts of the verification plan that remain untested are targeted during the coverage closure phase. This is the last leg of design verification. Exercisers are rerun with iteratively improved test templates until the required coverage is achieved. Because the probability of occurrence of these tests, which we regard as novel tests, is very low (less than 1%), large batches of tests are simulated. In cases where this fails, verification engineers resort to writing tests manually.

We suggest two methodologies to filter novel tests, independent of the test generator. The filters are specific to the targeted behavior. In the first approach, we use ISA simulation trace values directly to estimate test behavior. This allows us to filter only simple behaviors that are architecturally defined. In the second approach, we extrapolate trace data using a microarchitectural model for a closer estimation of test simulation behavior. We demonstrate the ability of our methodology to increase the density of rare tests by several orders of magnitude.

1.4 Related Work

Relevant related work has been discussed at the beginning of each chapter, wherever necessary.

1.5 Thesis Organization

We first analyze the constraint random test generation problem. In Chapter 2, we take a look at two of AMD's in-house x86 tests generators - Random Test Generator (RPG) and AMD Exerciser (Amex). We identify key differences in their capabilities based on their design principle.

In Chapter 3, we make a comparative analysis of the two exercisers to confirm our findings. We propose a flow to compare two exercisers based on functional coverage. We use coverage data from core-level simulations to verify Secure Virtual Machine (SVM) nested-paging features of the x86 ISA. We identify the limitations of Amex and propose to utilize external test filters to overcome them.

In Chapter 4, we develop an ISA simulation trace based filtering approach for novel test selection, which is independent of the test generator. We test the effectiveness of this methodology to identify novel tests from past regressions and generate new novel tests for x86 microprocessor verification. We gather experimental data from instruction cache, branch prediction and TLB operation to understand the practical limitations of this approach.

In Chapter 5.1, we develop a solution based on microarchitectural models to bridge the gap between RTL behavior and machine learning based test filtering. We demonstrate how microarchitectural models can give a more accurate description of test behavior. Experimental results based on the latest AMD multi-core processor design are presented to demonstrate the feasibility of our proposed approach.

Chapter 6 concludes the dissertation and discusses future line of work.

Chapter 2

Random test program generation

2.1 Introduction

The equivalence of a design and its functional specification can be proved by exhaustive simulation or a formal proof of correctness. An exhaustive simulation, in which all valid states of the design are exercised, can be done only for very small designs. This holds true for formal verification as well. As it is more scalable, simulation based verification is the mainstay of functional verification. In industrial designs, a relatively small subset of tests is simulated. The challenge, therefore, lies in developing a test suite that provides high confidence in the correctness of the design. Since the test-space is prohibitively big, this process has to be performed semi-automatically via software, which is aware of the design architecture and the verification plan. These are referred to as Constrained Random Test Generators, Random Test Program Generators (RTPG) or exercisers.

Constraint-based test generation is a well-studied behavioral level functional test generation paradigm, where a given design is converted into a set of constraints and constraint solvers are employed to generate tests.

2.1.1 Related work

Random test program generators were introduced to functional verification in the early 80s by IBM[40]. It owes its origin to software testing[10]. The basic idea, as shown in Fig. 2.1, remains the same to date. Random self-checking tests that are architecturally valid are generated to comprehensively verify the functional correctness of a DUT. The nature of the test cases ensure that they execute to completion and the results of the execution are predicted at the time of generation. So wherever possible, the tests are self-checking.

Genesys[22], Genesys-Pro[1] are two widely known random functional test generators developed by IBM. They follow the model-based test generation approach suggested by Aharon, Goodman, Levinger et.al.[43] and Malka, Lichtenstein and Aharon[6]. They use an architectural model and a constraint solver (in the case of Genesys-Pro) to generate random tests based on user defined template. Piparazzi[2] includes a microarchitectural model for higher precision and control over test behavior.

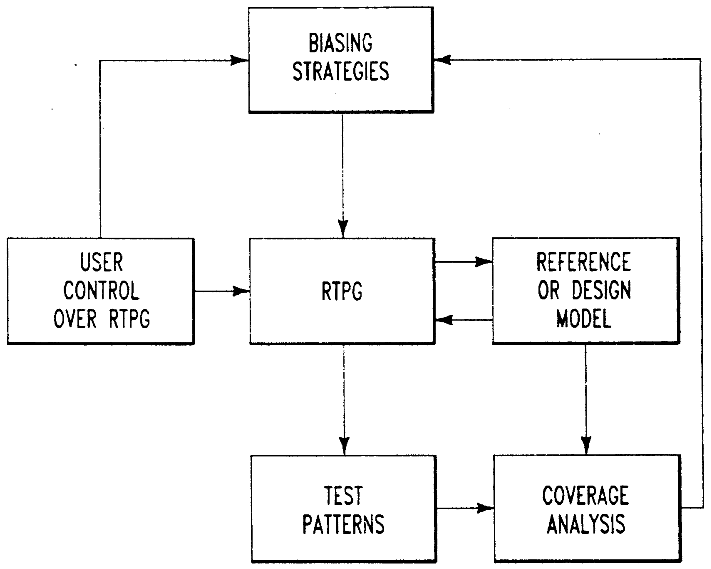


Figure 2.1: *Dynamic process for the generation of biased pseudo-random test patterns for the functional verification of hardware designs* proposed by Aharon, A., Bar-David, A., Gewirtzman, R. et.al. from IBM in 1990[5]

In this chapter, we examine the test generation problem and its practical challenges. We discuss the methodologies used by two AMD exercisers and identify differences in their working.

2.2 Constraint based verification

Constraint based verification automates the verification process. Instead of using manually generated tests to verify the design space, constrained random simulation relies on a constraint-based testbench. The constraints define the boundaries of legal design space as a function of design inputs and states. Constraints are executable through constraint solving. Randomizations during constraint solving enable the simulation to explore a much wider section of the verification space.

2.2.1 Reactive testbench

It would appear that having a reactive testbench that actively generates tests based on the processor state is an obvious solution to the test generation problem. As shown in Fig. 2.2, the testbench at any time is aware of the Design Under Test(DUT)'s state. It can synchronously generate constraints based on the machine state. These constraints can then be solved to generate relevant stimulus. This would allow it to precisely target any design behavior by executing the most appropriate instruction from a known RTL state, based on coverage goal requirements. In reality, however, this is not a viable solution for large designs. For example, in the case of a microprocessor, this is possible only if the memory, caches and instruction fetch mechanisms are bypassed and the instructions are directly injected to the pipeline for decode and execution. When instructions are inserted into the memory, it is impossible to predict the exact processor state at the time of execution, given the deep pipelining, super scalar execution and non-core components interacting with the processor core. These elements insert variable delays and indeterminism that make calculating the micro-architectural

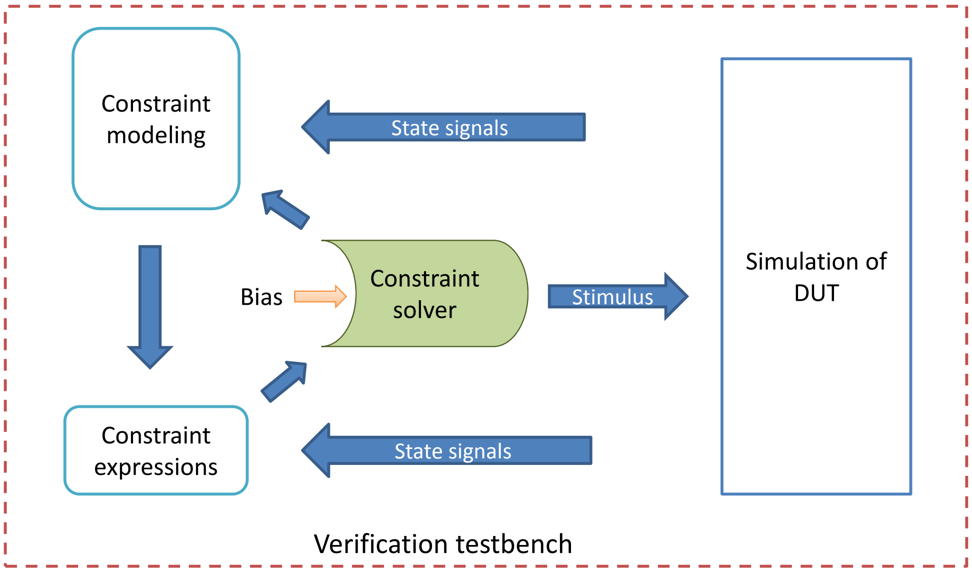


Figure 2.2: Testbench based stimulus generation

state at execution time impossible. Bypassing instruction fetch logic leaves a significant portion of logic untested.

Out-of-order execution is yet another roadblock to this approach. Let us suppose that we want to perform two back to back additions with a register dependency. We generate two `add` instructions where the second instruction uses the result of the first. The first instruction serves to *set up* the event, while the second *triggers* it. Consider the following piece of code:

```
mov ebx, 0xAEDC0987
mov rax, [ebx]
sub rdx, rax
add rcx, 0x8898
add rbx, rcx
...
```

There are three randomly generated instructions preceding the two `add` instructions. Since the second `mov` and the subsequent `sub` depend on a read from memory, there will be a pipeline stall if the instructions are executed in-order. To avoid this, the processor will fill the void by executing the first `add` instruction (and any subsequent instructions) that do not have a dependency on this load. This implies that the temporal relation of the two additions cannot be guaranteed. They will retire after the `mov` and `sub`, but their dispatch depends on the pipeline. Thus, the insertion of random instructions has disrupted the register-dependency event that was set up. This can also happen because of random code inserted after the `add` instructions. Test generation, therefore, is not as simple as knowing the architecture state based on the last retired instruction. It requires us to generate instructions beforehand to fill the prefetch pipeline. This has to be done in a way that does not disrupt an event being set up.

2.2.2 Constraints and biases

Constraints are formal specifications of design architecture and behaviors. In the context of constrained random simulation, there are basically two types of constraints: architecture constraints and constraints used as test directives. Architecture constraints define the design behavior. They play the role of describing the legal test space. Test directives determine the randomization and distribution of inputs within the test space. Biases guide tests towards coverage goals. Test directives are typically presented in the form of user generated templates.

2.2.3 Constraint representation

There are several ways to represent constraints depending on the domains being continuous or discrete, and finite or infinite, and the constraints being linear or non-linear. Boolean constraints, for example, can be represented as Binary Decision Diagrams (BDDs). Constraints required to generate a test can be represented as a single BDD, which is the combination of all behavioral constraints. The legal input space defined by this constraint is captured by the set of paths in the corresponding BDD that lead to the leaf node which evaluate to TRUE. Each of these paths can be viewed as an assignment to the variables on that path. The assignment to state variables represents a set of states, whereas the input assignment represents a set of input vectors that are legal under each of these states.

Depending on the constraint, the legal input space can be empty under certain states. These states are referred to as *illegal states* since the simulation cannot proceed upon entering those states. Consider the constraint $(x_1+x_2+s_1+s_2 \leq 1)$, where x_i represents an input and s_i represents a state variable. For the state $(s_1 = 0, s_2 = 0)$, all values of x_1 and x_2 satisfying the constraint $(x_1 + x_2 \leq 1)$ are valid inputs. So all inputs other than $(x_1 = 1, x_2 = 1)$ are legal input vectors. The state $(s_1 = 1, s_2 = 1)$ is an illegal state, since no input assignments can satisfy the constraint.

2.2.4 Input biasing and vector distribution

Solutions to test constraints are distributed using input biases. An input probability can be given either as a constant, or as a function of state. Input probabilities can only bias the solution, since the constraints assume higher priority. In extreme cases, the constraints can prohibit an input variable from taking a specific value at all times, even if the input variable is assigned a high probability for doing so.

Suppose, the input probability of $x = 1$ is a function of the state with a range in (0,1), denoted by $p^x(Y)$; the input probability of $x = 0$ is the function $1-p^x(Y)$, denoted by $p^{\bar{x}}(Y)$. X stands for input variables and Y stands for state variables. Let $\alpha = \alpha_1\alpha_2\alpha_3 \ldots \alpha_n$ be a vector of input variables. The weight of α is defined as

$$\pi(\alpha, Y) = \prod_{i=1}^{n} [\alpha_i.p^{x_i}(Y) + (1-\alpha_i).p^{\bar{x}_i}(Y)] \tag{2.1}$$

Constraints and input probabilities can be conceptually unified by the constrained probability of input vectors. The constrained probability of an input vector is the weight of the vector divided by the sum of the weights of all vectors that satisfy the constraint[45]. The constrained probability is defined to be zero if the given state is illegal. For a constraint f, if we denote the legal input space for a state s by f_s, the constrained probability of an input vector α for a state s is given by

$$p_s(\alpha) = \frac{\pi(\alpha, s)}{\sum_{\beta \in f_s} \pi(\beta, s)} \quad if\ \alpha \in f_s \tag{2.2}$$

$$0 \quad if\ \alpha \notin f_s \tag{2.3}$$

Consider a design with three inputs in1, in2 and in3 with input probabilities 1/3, 1/6 and 1/2 respectively. If we begin with no constraints, all eight vectors are valid inputs. The distribution of inputs is shown in Table. 2.1.

Table 2.1: Input vector distribution for a 3-input system with no constraints

in1	in2	in3	Vector probability
0	0	0	2/3 * 5/6 * 1/2 = 5/18
0	0	1	2/3 * 5/6 * 1/2 = 5/18
0	1	0	2/3 * 1/6 * 1/2 = 1/18
0	1	1	2/3 * 1/6 * 1/2 = 1/18
1	0	0	1/3 * 5/6 * 1/2 = 5/36
1	0	1	1/3 * 5/6 * 1/2 = 5/36
1	1	0	1/3 * 1/6 * 1/2 = 1/36
1	1	1	1/3 * 1/6 * 1/2 = 1/36
			Total = 1

The design space is limited by applying constraints on what constitutes a legal input vector. For example, if we apply a constraint that (in1 + in2 + in3 $\leq$ 1), inputs (0,1,1), (1,0,1), (1,1,0) and (1,1,1) are no longer legal inputs. The remaining vector with their constrained probabilities are shown in table 2.2

Table 2.2: Input vector distribution for a 3-input system with the constraint (in1 + in2 + in3 $\leq$ 1)

in1	in2	in3	Original probability	Normalized Probability
0	0	0	5/18	5/18 * 4/3 = 10/27
0	0	1	5/18	5/18 * 4/3 = 10/27
0	1	0	1/18	1/18 * 4/3 = 2/27
1	0	0	5/36	5/36 * 4/3 = 5/27
			Total = 3/4	Total = 1

As shown in the last column, the probabilities of the input vectors are normalized to account for the reduction in the size of the test-set. Similar biasing can be used to get a desired distribution of random instruction sequences.

2.3 Challenges of test generation

There are two main challenges to constraint based test generation:

1. Converting the design to a set of constraints

2. Solving the constraints

Both of these are problems that current techniques and processing power are incapable of solving. Firstly, translating design specification into constraints is not practical, both manually and automatically. While smaller units can be represented declaratively as a set of constraints, it is impractical for something as big as a processor core. Even if we did manage to develop a representative set of constraints, we cannot verify its completeness or correctness. Since the design space is exceptionally large, solving a set of constraints this big is computationally intractable. As a result, test generation methods are, at best, semi-automated where a reduced set of constraints and biases guide the test generation process. The biases are provided by the user based on coverage goals.

When there is a large number of constraints, forming the conjunction BDD is expensive as the computation blows up because of large intermediate BDDs. They are harder to solve and the conjunction BDD slows down vector generation. This is true irrespective of how the constraints are represented. Exercisers therefore resort to a form of sequential constraint solving. Instead of generating one massive combined constraint, they solve a series of smaller low-complexity constraints[15]. This makes the process more manageable. However, without good heuristics and backtracking, constraints can appear unsolvable. That is to say, if there exists a solution for a constraint set, since they are solved sequentially and independently, finding the solution is not guaranteed.

2.3.1 Practical difficulties

In addition to the issues discussed so far, there are a number of practical difficulties that engineers face when designing an exerciser. Here is a list of some of the most prominent challenges:

1. **Constrained randomization** The foremost challenge of random test generation is striking the right balance between constraining generated tests and randomizing them. While on one hand, we want an even distribution of tests across the verification space, on the other hand, we want high coverage on all the less likely to reach ”corners”. In other words, besides giving the correct inputs, giving âĂIJgoodâĂİ inputs that are more likely to exercise interesting scenarios is a top priority for a constrained random generator. Constraint solving and randomization have to be done simultaneously to achieve the desired distribution or weighting.

2. **ISA complexity** The sheer complexity of ISAs make them hard to verify; in this case the x86 ISA. x86 is a dated architecture. Even though the x86 ISA has evolved significantly several times, the hardware is required to maintain backwards compatibility. Several memory-addressing models have to be in place to allow older code to interoperate on the same processor without modification. Instruction codes, addressing modes, memory addressing etc are

much more complicated than the ones for most RISC or MIPS CPUs. The complexity of the ISA makes it difficult to randomly generate meaningful tests.

3. **Size of verification space** Because the architecture is complex, the test space to be verified is ginormous. Even for a sequence of 4 instructions, there are thousands of possible input combinations. Some of the variables for an instruction sequence are:

    - Opcodes
    - Addressing modes
    - Privilege levels
    - Paging modes
    - Memory addressing and memory type
    - Self modifying code (SMC) / Cross Modifying code (CMC)
    - Faults and exceptions
    - External interrupts
    - Machine check errors
    - Virtualization

This makes generating long sequences of tests that are functionally useful challenging. To appreciate the vastness of the verification space, consider the following fact. The total number of particles in the universe that have ever existed is estimated to be around 10^{80}[39]. Twenty 64-bit general purpose registers in an x86 processor can be initialized to roughly 10^{385} different states. And this is just an insignificant fraction of a processor's verification space.

4. **Memory management** Similar to generating interesting test sequences, managing data and memory is a formidable task. Memory management includes defining data and code segments, descriptor tables, page tables, stack and data structures such as Virtual Machine Control Blocks (VMCB). While all this data has to be randomized, it should not lead to a failed processor state on execution. The constraints on randomization are applied not only by the architecture, but by the user and the test. There are more input constraints on data when trying to target specific behavior. In addition to solving all these constraints, tests have to be relevant and interesting.

   For example, since the smallest supported page size on x86 is 4Kbyte, the lower 12 bits of a linear address (LA[11 : 0]) server as the offset for accessing data. So these bits are the same for a linear address and its corresponding

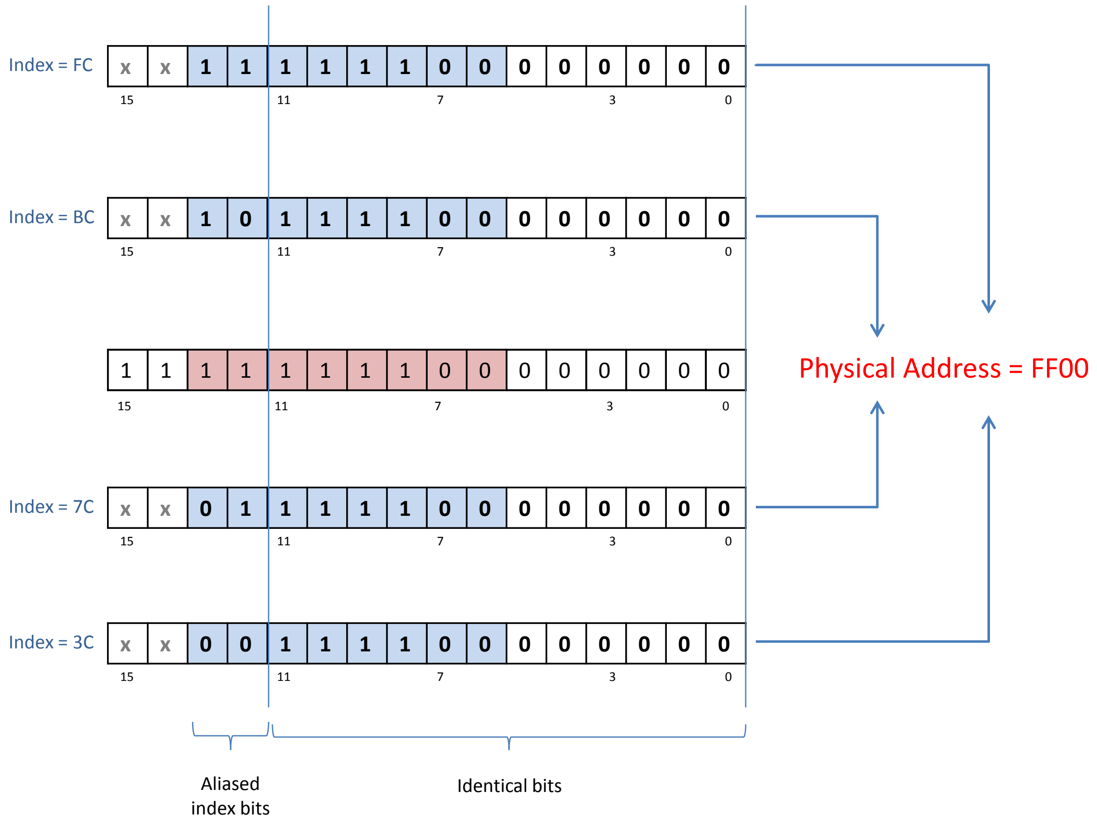


Figure 2.3: Index aliasing for a 2-bit overlap. Linear addresses that map to a single physical address can have one of fours indices.

physical address. Consider an instruction cache with 256 64-byte entries. Its index is given by LA$[13:6]$. For a 32-bit physical address space, PA$[31:12]$ serve as the tag bits. Each physical address can map to 4 different indices because of the aliasing of bits 13 and 12. For example, suppose the physical address is 0xFF00, as shown in Fig. 2.3. Its linear address could have an index of 0xFC, 0xBC, 0x7C or 0x3C. For all fetch requests from the L1 instruction cache, the L2 cache back-probes the IC to ensure that there is no aliasing. That is to say, it has to check all the four possible indices to ensure that the requested physical is not already present in the IC. To verify the back-probing mechanism, the exerciser has to generate multiple random linear addresses which map to the same physical address and generate appropriately timed fetch requests.

Another example of generating interesting scenarios through intelligent memory allocation is having the code and data within the same cacheline. The Load-Store Unit (LSU) ensures that it has exclusive access to a location before modifying it. This is required to maintain data coherency. It does this by sending an invalidating probe to other units within the core, and other processor cores in the case of a multi-core processor. One likely bug is the IC not invalidating a cacheline because it was in the process of fetching it from the L2 cache. Because the data is not present in IC when the invalidating probe from LSU arrives, it could escape the probe. The data can arrive shortly after IC does a look-up and remain as a valid entry. Unless the fetch request from IC and the probe are perfectly timed, this bug cannot be exposed.

Creating and managing page tables also requires thought. An operating system switches between several page tables every second. This is hard to model in a test because of the demands on system memory required to generate such a test. In a test, the memory is divided into pages anywhere from 4Kbytes to 1Gbyte in size. The address translation is maintained using one or more page tables. Every single page table entry has its own set of attributes (like Read/Write, User/Supervisor, Present, Dirty, Accessed etc). Hierarchical page tables increase the size and complexity of tracking page tables. The virtual address space that has to be tracked is also huge; a 52-bit physical address space is addressed by 64 bits of virtual memory space. To add to the complexity, some operating systems use sophisticated schemes for memory management. For example, they let page tables loop back on themselves instead of having different tables for each level of the hierarchy, to save page table space.

5. **Performance features** Since memory operations are expensive, processors device ingenious methods to avoid keeping the core idle between memory accesses. For example, consider the stack engine. The purpose of the stack engine is to reduce the number of stack operations dispatched and executed and to eliminate serial stack pointer (RSP) register dependencies. Pushes, pops, near-call and near-return instructions are optimized to avoid too many updates to the stack pointer by keeping a copy of an offset and updating this. When this offset is about to overflow, the processor inserts a âĂIJfixâĂİ operation to adjust the actual stack pointer once and reset the offset. This

offset is not architecturally visible to the program âĂŞ it only perceives the stack pointer itself.

What makes the stack engine hard to verify is that it is a performance feature. For example, x86 does not provide an instruction to read the current instruction pointer (RIP). Commonly, software gets around this issue by doing a call and then reading the value of the calling instruction from the stack pointer. In reality, transparent to software, when the stack engine recognizes this, instead of doing real writes and reads from memory (stack push and pop), the hardware just forwards register values from the current instruction pointer (RIP) without doing a push or a pop. Bugs in performance features are hard to catch. To verify the stack engine, the stack pointer has to be used extensively, exercising all possible scenarios, however unusual they may be. In addition to this, for simulation to catch a bug, it must be triggered such that it manifests as a change in an architectural value.

6. **Valid tests** All generated tests have to be architecturally valid. They should execute to completion when simulated and not run into infinite loops. Since ISA simulations do not model the micro-architecture, the peripherals or the data fabric, it is impossible to guarantee test validity before simulation. For example, tests should be able to handle randomly injected machine check

errors and external interrupts. Much like DUTs themselves, unless a test is robust enough to handle all kinds of RTL behavior, it can end prematurely.

In addition to being valid, tests must simulate meaningful behavior. In case of the stack engine, for example, its not sufficient to have increased stack pointer related operations. Exercisers must verify the values read from the stack. Discarding data read off the stack allows issues in the stack implementation to escape. Instead, they must be validated by reusing them to modify the processor configuration.

7. **Exerciser verification** Exercisers are not verified for accuracy; primarily because there is no efficient way to do it. The legality of generated tests is verified by the architectural simulator. The quality of tests, though, is hard to assess. This is largely determined by the quality of constraints, the constraint solver and biases used. Both over-constraining and under-constraining reduce test quality. These issues can only be identified after prolonged use of an exerciser. If a high quality verification plan is used, exerciser issues show up as coverage holes. Otherwise, coverage loss due to poor exerciser design or poor user directives go unnoticed.

2.4 Exerciser I: Random Program Generator (RPG)

2.4.1 Overview

RPG is a simple, light-weight, legacy test generator. It takes a minimalistic approach to test generation by having only the most essential test constraints. These constraints merely ensure the legality of tests. RPG does not follow a strict model based approach, of keeping the test generator separate from the processor model. Constraint solving is done on a ad-hoc basis, merely to ensure that each instruction inserted into the test sequence is architecturally valid. It functions largely like a randomizer, than a constraint solver.

This approach provides RPG with a lot of flexibility. For example, for a memory access that causes a page fault, it will not try to fetch the page from disk to memory. It does not use a fixed exception handler code either. Instead, it resumes execution from a random location. This not just allows it to keep the exerciser design simple, but also makes its behavior more unpredictable. RPG tests are of fixed length. Their behavior can be controlled through:

a) Feature disabling/enabling

b) Input biases

c) Custom procedures (macros)

Constraints are applied by enabling or disabling features. For example, we can choose to disable instruction prefixes. Custom procedures, called *macros*, allow inserting specific instruction sequences. To test more complex design behavior that require tight sequencing of instruction, it allows the use of custom procedures called *macros*.

2.4.2 Test generation

RPG first breaks up memory into code and data segments. The number and size of code and data segments can be specified by the user as an integer or a range of numbers. Following this, page tables are created based on user specified count, paging modes and paging attributes. `GlobalPagePct,NotPresentPagePct, SupervisorPagePct,CodeCross4kBndPct` and `DataCross4kBndPct` are some examples of paging attributes.

The code space is populated with randomly generated instructions selected based on user defined weights. The description of every instruction and macro (instruction block) available for use by RPG is described in the form of a 3-level hierarchical structure, as shown in Fig. 2.3.

1. **Table:** Each entry is a general type of instruction. For example, all arithmetic and logic instructions are organized into one table.

2. **Group:** Each entry is (usually) a different mnemonic. For example, all versions of the ADD instruction are in the same group.

3. **Instruction:** Each entry is an instruction or macro. For example, the register-memory version of ADD

Table 2.3: Illustration of the 3-level hierarchical table describing the instruction-set, which is used by RPG to generate random instructions

```
#Table ArithLogic
//==================================================================================
#Group Aaa
"AAA",     OP_NA, OP_NA, OP_NA, NA,  0x37,  C_GEN | INV64,     Create<cInstGen>
#Group Aad
"AAD",     IMM8,  OP_NA, OP_NA, NA,  0xd5,  C_GEN | INV64,     Create<cInstGen>
#Group Aam
"AAM",     IMM8,  OP_NA, OP_NA, NA,  0xd4,  C_GEN | INV64,     Create<cInstGen>
#Group Aas
"AAS",     OP_NA, OP_NA, OP_NA, NA,  0x3f,  C_GEN | INV64,     Create<cInstGen>
#Group Add
"ADD",     R_AL,  IMMS8, OP_NA, NA,  0x04,  C_GEN,             Create<cInstGen>
"ADD",     R_XAX, IMMX,  OP_NA, NA,  0x05,  C_GEN,             Create<cInstGen>
"ADD",     REG8,  IMM8,  OP_NA, NA,  0x80,  C_GEN,             Create<cInstGen>
..
..
..
#Table LoadStore
//==================================================================================
#Group Mov
"MOV",     R_AL,  DISP8, OP_NA, NA, 0xa0, C_GEN,     Create<cInstGen>
"MOV",     R_XAX, DISPX, OP_NA, NA, 0xa1, C_GEN,     Create<cInstGen>
"MOV",     DISP8, R_AL,  OP_NA, NA, 0xa2, C_GEN,     Create<cInstGen>
..
..
```

At the beginning of code generation, the weights corresponding to each of these are read from the user input. In cases where they are not specified, random weights are assigned. Each instruction is defined using several attributes in a tabular format. All the information required to legally use an instruction is contained in this definition. The following is a description of some of the fields used to define an instruction (this list is not exhaustive):

1. **Mnemonic String** Mnemonic description of the instruction.
2. **Operand 0** This specifies what type of first operand the instruction takes (or OP_NA if none).
3. **Operand 1** This specifies what type of second operand the instruction takes (or OP_NA if none).
4. **Operand 2** This specifies what type of third operand the instruction takes (or OP_NA if none).
5. **Prefix** This is only needed for instructions that depend on a certain prefix for their functionality.
6. **Opcode Byte** Opcode of the instruction
7. **Info** Specifies various instruction information. For example:
    - Instruction Class: C_GEN (general purpose),C_FPU (floating point), C_SVM (secure virtual machine) etc
    - Instruction Attributes: XFER (branch), SP (can modify stack pointer), SW_* (can switch modes) etc

8. Pointer to instruction class generation function: The default class for instructions that do not require setup code is `Create<cInstGen>`. Each different type of instruction needing special setup code is a different class derived from the parent class `cInstGen`. This field includes a pointer to a function that returns the correct type of object required to generate an instruction or sequence.

RPG does not track conditional branches. It uses valid addresses as operands when inserting branch instructions. For indirect branches, the register used as operand is pre-loaded with a valid memory pointer. RPG does not force the condition flags prior to conditional branch to regulate code flow. In fact, RPG makes no attempt to guide the flow of control in a test.

Instruction generation is concluded once 5000 instructions are generated. Any pending sequences are completed before the test is concluded. On rare occurrences, tests run into infinite loops. They are identified by running an architecture simulation. Simulations that do not complete before a preset time are discarded.

RPG has a dedicated working mode for producing higher address collisions in TLB and cache. In this mode, linear addresses generation is restricted to a few indices to cause higher than normal address collisions in TLB and cache.

2.4.3 Macros

Similar to the table containing instruction groups, RPG has a table for *macros*. Macros are a way to generate specific instruction sequences. Macros describe the sequences using operators provided by RPG. For example, consider the following macro created to cause TLB reloads. This macro was targeted at a specific bug that caused the TLB to stall during a reload (inserting new entry) under some (unknown) configurations. Since sufficient debug information was not available, verification engineers tried to trigger the bug using tests that caused increased TLB activity.

```
%macro TlbRldStallSqnce
 %info CM32 CM16 PM32 PM16 RM32 RM16 SMM
          %assign @addr1 @RandDataAddr
          %assign @addr2 @RandDataAddr
          %assign @reg1 @RandGpReg
          %assign @instr @RandMemRegInstr
          @instr [@addr1], @reg1
          %rep (0,2)
             *
          %endrep
          prefetch byte [@addr2]
          %rep (0,2)
             *
          %endrep
          clflush byte [@addr2]
 %endmacro
```

.

The first two lines indicate the name of the macro and its supported operated modes. The next four lines randomly pick addresses (`addr1` and `addr2`), a register (`reg1`) and a register-based memory instruction (`instr`). It then uses them to create a sequence that is likely to trigger the bug. First, the randomly selected instruction is used to generate a memory write using the chosen register as the source and one of the random addresses as a memory pointer. This is followed by 2 random instructions (that are unconstrained). This is followed by a PREFETCH, two more random instructions and a CFLUSH. PREFETCH loads the entire 64-byte aligned memory sequence containing the specified memory address into the L1 data cache. CFLUSH flushes the cache line specified by the linear-address. The instruction checks all levels of the cache hierarchy and invalidates the cache line in every cache in which it is found. Because this sequence causes back-to-back writes and reads from memory for random addresses, it will generates high TLB traffic.

Macros are processed sequentially. Once a macro is completed, the next instruction or macro is processed. RPG does not support interleaving of multiple macros.

2.5 Exerciser II: AMD Exerciser(Amex)

2.5.1 Overview

AMD Exerciser(Amex) is a newer exerciser compared to RPG. Amex is more structured and constrained and that gives it better control over most test behavior.

What sets Amex really apart from other exercisers is that it does not generate tests using pre-determined random data. Model-based exercisers like Genesys Pro use a single value for each processor resource[1]. They use a boolean value to track whether a given value is actually known. If the value is known, successive instruction sequences are tailored to that value. Amex was developed with a strong emphasis on generating tests with multiple arbitrary paths to test aggressive branch prediction and cache coherency of multi-core processor systems[15]. Amex tests can therefore be rerun from the beginning and a different path through the program can be exercised within the same simulation run. This introduces indeterminism and reduces resource reloading, thereby improving test quality[3]. This also allows random resets from the test bench to be handled properly. Amex was developed as an improvement over RPG.

2.5.2 Amex test structure

Fig. 2.4 is structure of a typical Amex test. The control flow through a program at the highest level is fixed before test generation commences. Amex tests have multiple paths through a test and the path taken depends on the initializing data. Constructing a code flow graph ensures code convergence for all tests.

Tests are generated based on a control flow graph made up of basic blocks containing randomly generated code. Blocks are regions of physical memory that are reserved for a particular purpose and they can contain one or more pages of instruction or data. The random code is encapsulated within a framework containing initialization, termination and support code to handle interrupts. Since tests are self checking, the termination code compares the results of the execution on successful completion of the test case and reports a pass or fail. The control flow graph is a directed acyclic graph with a single node having no parents. In Fig. 2.4, the parent (root) node is $n1$.

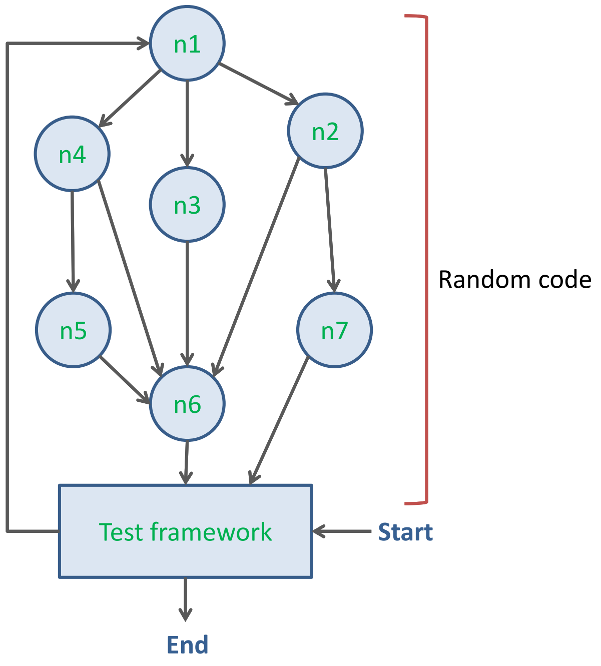
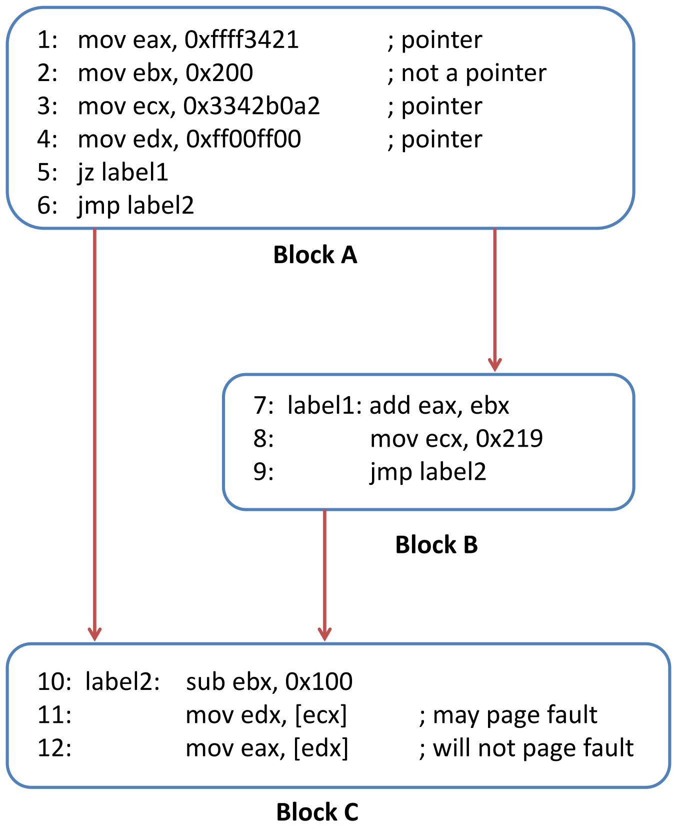


Figure 2.4: Structure of an Amex generated test. Each node, n1 through n7, is a basic block.

Each basic block has an exit point that connects it to another basic block or test framework. Conditional jumps create multiple exit points in a block. There are 5 possible paths through this graph; for example, one path is $n1 \rightarrow n3 \rightarrow n6$. Because of the possibility of having multiple execution paths in a test case, resource values depend on the path taken. Amex therefore represents values as the union of the values of all the processor states. In cases where there is indeterminism, such as a conditional branch, it uses this union to predict what is possible over subsequent instructions.

Fig. 2.5 shows a snippet of code with a re-converging control flow. After diverging from `Block A`, the control flow reconverges at `label2` in Block C. The diverging behavior is caused by the conditional jump at line 5. If the jump is taken, then `Block B` overwrites the address stored in `ecx`. Register `ecx` can therefore take two possible values: 0x3342b0a2 and 0x219. While the first value is a known pointer value, the second one is just a random data value. If the conditional branch at line 5 is taken, the load at line 11 could page fault. The load at line 12, however, will not page fault as the value of `edx` was not modified by `Block B`. Amex gets around this ambiguity by representing the value of processor states using classes that can represent a set of values. These classes support basic arithmetic and set operations.



```
mov eax, 0xffff3421        ; pointer
mov ebx, 0x200             ; not a pointer
mov ecx, 0x3342b0a2        ; pointer
mov edx, 0xff00ff00        ; pointer
jz label1
jmp label2
```

```
label1: add eax, ebx
        mov ecx, 0x219
        jmp label2
```

```
label2:   sub ebx, 0x100
          mov edx, [ecx]         ; may page fault
          mov eax, [edx]         ; will not page fault
```

Figure 2.5: Snippet of Amex code

2.5.3 Test generation algorithm

The test generation mechanism of Amex can be described at a high level using the following algorithm:

1. Memory is allocated to code and data blocks and page tables are set up to map effective addresses to linear addresses.

2. Basic blocks are grouped into directed acyclic graphs with a root node having no parents. Separate graphs are created for each processor thread.

3. One graph is selected for processing.
4. For the graph being processed, a block is selected that has no unprocessed parents (or no parents in case of the root block).
5. The starting state for the block is calculated. The initial state of the root block depends on the processor state configured by the testbench. This randomization is known to Amex. For all other blocks, the state is based on the union of the states of the block's parents' final state. The size of the union is limited to a compile-time limit. When this limit is breached, the state is marked GENERAL.
6. If there are blocks left unprocessed, go to step 4; else, go to step 7.
7. Select the next graph. If all graphs have been processed, test case is complete.

2.5.4 Test generation

Blocks are regions reserved in memory for specific purposes. The four main types of block are *shared*, *non_shared*, *guest* and *guest_end*. These block types contain randomly generated instructions. There are other types of blocks, such

For performance reasons, a state is marked as GENERAL if the number of possible values is higher than a preset limit. In effect, it means that the possible outcomes for the state are so high, that it can be treated as unknown. This conversion happens rarely in reality.

as blocks to hold data and blocks that are used to hold static code that makes up the test case framework. These other blocks are not dependent on the flow of code and are therefore not included in the acyclic graph used to generate tests. All blocks of the same type are the same size. The size and count of each type of block is controlled by user defined parameters. The static code fragments used in creating the test framework(refer to Fig. 2.4) are generated based on user defined constraints.

Blocks contain links that connect them to other blocks in the graph. The code blocks are filled with random instructions based on the constraint and biases supplied by the user. These inputs affect both, the choice of *generators* that produce random instructions and the type and format of instruction picked up a selected generator. The choice of instructions is biased by a weight tree constructed from user defined biases. The implementation is very similar to that of RPG (described in Sec. 2.4). Higher weights increase the likelihood of an instruction being picked. Undefined weights are assigned default values. After a code block is filled, the next block is chosen by selecting randomly from a list of blocks in which all the parents have been processed. Amex does not track the actual values of data read from or written to memory. Information about data values read into registers is picked from the block type it comes from. There are many different block types in Amex that can supply data for instructions. Some block types provide control data specific to x86. These value types are tracked symbolically and can be moved from register to register. They can be written to a control register or machine-specific register(MSR) as long as no arithmetic operations have been performed and the symbolic type is compatible with the target register.

Since blocks are chosen for processing only if all of its parents are processed, the blocks get processed in a 'breadth first' manner. The initial state of each block is derived from the union of its parents' final state. After the first pass, Amex does a `SaveState` at the end of an architecture simulation to dump processor state to memory. In then includes a comparison with the saved state at the end of the test to make the test self-checking.

2.6 Differences between Amex and RPG

1. The most important difference is the flow of control. While RPG does not control the program flow in any way, Amex has a structured approach to the control of program flow. This allows us to customize aspects such as program size, number of guests and the distribution and placement of branches.

2. Amex offers more randomized setup code. Instructions have requirements such as operating modes, operand values and priority levels, to be legally executed. In RPG, the setup code is inserted just before each instruction

and it does not intersperse setup code with random instructions. The data used by the setup code is randomized, but other than that, there is not much of a variation. In Amex, this is implemented through generators. There is no fixed constraint between the setup and the trigger instruction because Amex can lock resources. So the trigger can be spatially separated from the setup code. This gives it flexibility in placing the trigger event.

3. RPG has a data-restrict mode where the choice of addresses is constrained. For example, it can generate more address collisions in the TLB and causes higher cache evictions. In this mode, it picks up linear addresses that map to s restricted set of TLB or cache indices. RPG has therefore been traditionally more successful at using all the ways of set-associatively mapped TLBs and caches.

4. Amex is better suited for generating focused tests since it offers more test constraints. Amex includes thousands of pre-built constraints that the user merely has to enable. For example, there are multiple weights controlling the choice of a memory pointer:

    - Address size
    - Block to pick an address from (eg: shared, guest, stack)
    - Address representation (base only/base + displacement/RIP-relative)

- Minimum distance to a previously chosen address
- Reuse a previously used address

RPG does not offer this kind of a flexibility. These behaviors have to be induced implicitly by defining custom macros such as the `TlbRldStallSqnce` sequence discussed in Sec. 2.4. Since macros operate at the instruction and operand level, the constraints are less precise.

The flexibility of Amex, however, comes at a price. A lot of these constraints are also overlapping. For example, we have seen how code is generated using blocks in Amex. Generators are picked at random to insert instruction sequences into code blocks. Amex requires generator to complete code insertion within a block. If it does not complete generating a sequence within that block, the generator fails. So even if a heavily weighted generator is picked repeatedly, the required sequence will not be inserted if there is an opposing constraint or resource paucity. The price of having higher control is the difficulty in understanding how the different constraints interact and resolving any conflicts.

5. RPG creates page tables by itself whereas Amex delegates this to an external program. This gives RPG full control over the paging structure. For example, it can assign as many linear addresses to a single physical addresses as

desired. So for operations such as a string copy, that require a large number of memory pointers, RPG can easily generate tests without a large memory overhead. Amex attempts to generate the setup code by doing a large number of moves to registers. Sometimes, this fails due to resources (pointers or free registers) not being available. In other instances, the setup code (string of move operations) disturbs the existing pipeline state. Another example is Amex's limitation in generating page-faults. Amex can request page faults only at leaf-level entries of the hierarchical page table. Since PML4 (the highest possible level) is never at the leaf-level, Amex cannot trigger page faults at this level.

2.7 Summary

This chapter gives an overview of the constrained test program generation paradigm. It describes the challenges of constrained pseudo-random test generation. It also discusses the practical difficulties of designing an exerciser for x86 microprocessor verification.

The main focus of this chapter are two of AMD's exercisers: RPG and Amex. RPG takes a direct and simplistic approach to the test generation problem by keeping test constraints to a minimum. RPG generates fixed-length tests without any control over the program flow or structure. It merely attempts to generate valid tests by picking instructions randomly. This keeps its tests truly random and unpredictable. Amex, being modern, takes a more structured approach to test generation. The control flow at the highest level is pre-calculated for each test. Tests are constructed with multiple arbitrary paths. To account for this ambiguity, values of resources such as registers are stored as a union of possible values. Amex provides pre-defined constraints in the form of generators to induce specific behaviors. Amex also contains abundant constraints and biases to modify test behavior, giving it a contained, yet robust test generation capability.

Chapter 3

Evaluating effectiveness of exercisers

The use of legacy test generator can be justified if we can demonstrate that its scope of test generation is not a subset of a newer test generator. There are no existing flows to compare exercisers and evaluate their relative benefits and drawbacks. In this chapter, we propose a flow to compare Amex and RPG based on their functional coverage of SVM nested-paging. We identify tests that are unique to RPG.

3.1 Exerciser performance metrics

As is the case with design coverage, there is no single metric to fully qualify exerciser performance[31]. An exerciser can be assessed on several different parameters. In this section, we will focus our attention on comparing the two exerciser that have been previously discussed, RPG and Amex. The driving factor behind this comparative study is the need to reduce test redundancy arising from the use of multiple exercisers. To begin with, let us take a look at the most important exerciser qualities.

1. **Completeness of verification:** Bug discovery rate is among the most important sign-off criteria for design verification. As seen in Fig 1.3, it provides a general sense of verification completeness. Bug discovery rate begins to drop after extensive verification. This makes bug data a great resource for understanding exerciser differences. But the descriptive nature of bug reports is a limiting factor. It lacks a well defined structure. Since the design and verification environment change continuously, recreating bugs is tedious. Besides, bug reports in themselves are subjective and the contents vary depending on the stage of the design cycle at which it is filed at and the person filing it.

   On the other hand, coverage data is a more rigorous metric to define verification effectiveness. It gives a sense of how much of the design space has been covered. Coverage information is collected by creating and implementing test plans, generating stimulus, running simulations and measuring response. Unlike bug data, we have the ability to generate as much data as necessary. Coverage information can be functional or code coverage (includes micro-code coverage and toggle coverage).

2. **Test distribution:** In addition to the total coverage provided, the rate at which a design is covered is indicative of an exerciser's efficiency. Greater diversity of test behavior accelerates design space exploration and reduces wastage of simulation cycles. Coverage goals are met faster with better test randomization.

3. **Usability:** An exerciser's usability is a measure of how well it can be constrained to achieve desired coverage. Ideally, an exerciser has to be completely controllable to target any design behavior, and be completely random over all unconstrained parameters. But in the interest of generating valid tests, some user imposed constraints may be overridden during test generation. When constraints do not produce desired results, sufficient exerciser debug information should be available to explain the deviation from constrained behavior. This reduces the turn-around time required to eliminate dependencies and contentions between constraints.

3.1.1 Exposing differences in exerciser performances

To expose the differences between any two exercisers, we have to choose the right design feature to observe differences. For example, comparing the coverage of fundamental design properties such as memory writes, integer ALU operations or architectural properties cannot reveal any differences. By making a comparison based on using all general-purpose registers for multiplication, we cannot draw any useful conclusion. All exercisers are guaranteed to deliver 100% functional coverage.

We can choose features with a reasonable level of complexity by analyzing data from past regressions. The domain knowledge of design and verification engineers is helpful in this regard. For our analysis, we will be testing the nested paging functionality of Secure Virtual Machines (SVM) on an x86 processor. This specific feature has been chosen for comparison because load-store unit (LSU) verification is challenging. Since SVM adds an additional layer of complexity to LSU verification, testing all nested paging functionalities requires considerable manual effort. Besides, functional coverage data from past regressions showed a significant difference in the coverage from the two exercisers.

We use the existing verification plan as a reference for comparing exerciser capabilities. We will not attempt to assess or improve the quality of the test plan. Our objective is to ensure that all the functionality to be verified is done optimally.

The focus of this work is on reducing the coverage closure time by providing test engineers with tools that help identify novel tests.

3.2 Secure Virtual Machine (SVM)

The Secure Virtual Machine (SVM) feature is also known as virtualization technology. This is because SVM provides hardware which allows a single processor to run multiple operating systems in a secure and efficient manner. It consists of a Virtual Machine Monitor (VMM, also called the hypervisor), which is software that controls the execution of multiple guest operating systems (OSs) on a single processor core[4]. It gives each guest OS a false appearance of full control over a complete computer system (memory, CPU, and peripheral devices). The use of the term ”host” hence refers to the execution context of the hypervisor. A context switch refers to the operation of switching between the host and guest operating systems.

3.2.1 Running a virtual machine

On a SVM enabled core, a virtual machine (guest) can be run using the VMRUN instruction. VMRUN takes the physical address of a 4KB-aligned page as a single argument. This is a pointer to the virtual machine control block (VMCB), which describes a virtual machine (guest) to be executed[4]. The VMRUN instruction saves some host processor state information in the host state-save area in main memory; it then loads corresponding guest state from the VMCB state-save area. As the name suggests, the VMCB is a block of memory that describes the virtual machine to be executed. Specifically, a VMCB contains:

- Guest processor state (such as control registers)
- Various control bits that specify the execution environment of the guest or that indicate special actions to be taken before running guest code
- A list of instructions or events in the guest (e.g., write to CR3) to intercept

The TLB_CONTROL field in the VMCB, for example, tells the host if the TLB should be flushed when the virtual machine is started. The encoding of this field is as follows:

- 00h - Do nothing
- 01h - Flush entire TLB (all entries, all ASIDs)
- 03h - Flush this guest's TLB entries
- 07h - Flush this guest's non-global entries

The CR3 control register points to the base address of the highest-level page-translation table. A change in the CR3 value induces a TLB flush, as the translations are no longer valid. Only those entries marked as global are retained across context switches.

3.2.2 Page translation

The x86 page-translation mechanism enables system software to create separate address spaces for each process or application. These address spaces are known as virtual address spaces. System software uses the paging mechanism to selectively map individual pages of physical memory into the virtual-address space using a set of hierarchical address-translation tables called page tables. The paging mechanism and page tables are used to provide each process with its own private region of physical memory for storing its code and data.

Processes can be protected from each other by isolating them within the virtual-address space. A process cannot access physical memory that is not mapped into its virtual-address space by system software. System software can also use the paging mechanism to selectively map physical-memory pages into multiple virtual-address spaces. Mapping physical pages in this manner allows them to be shared by multiple processes and applications. The physical pages can be configured by the page tables to allow read-only access. This prevents applications from altering the pages and ensures their integrity for use by all applications.

The system-software portion of the address space necessarily includes system-only data areas that must be protected from accesses by applications. System

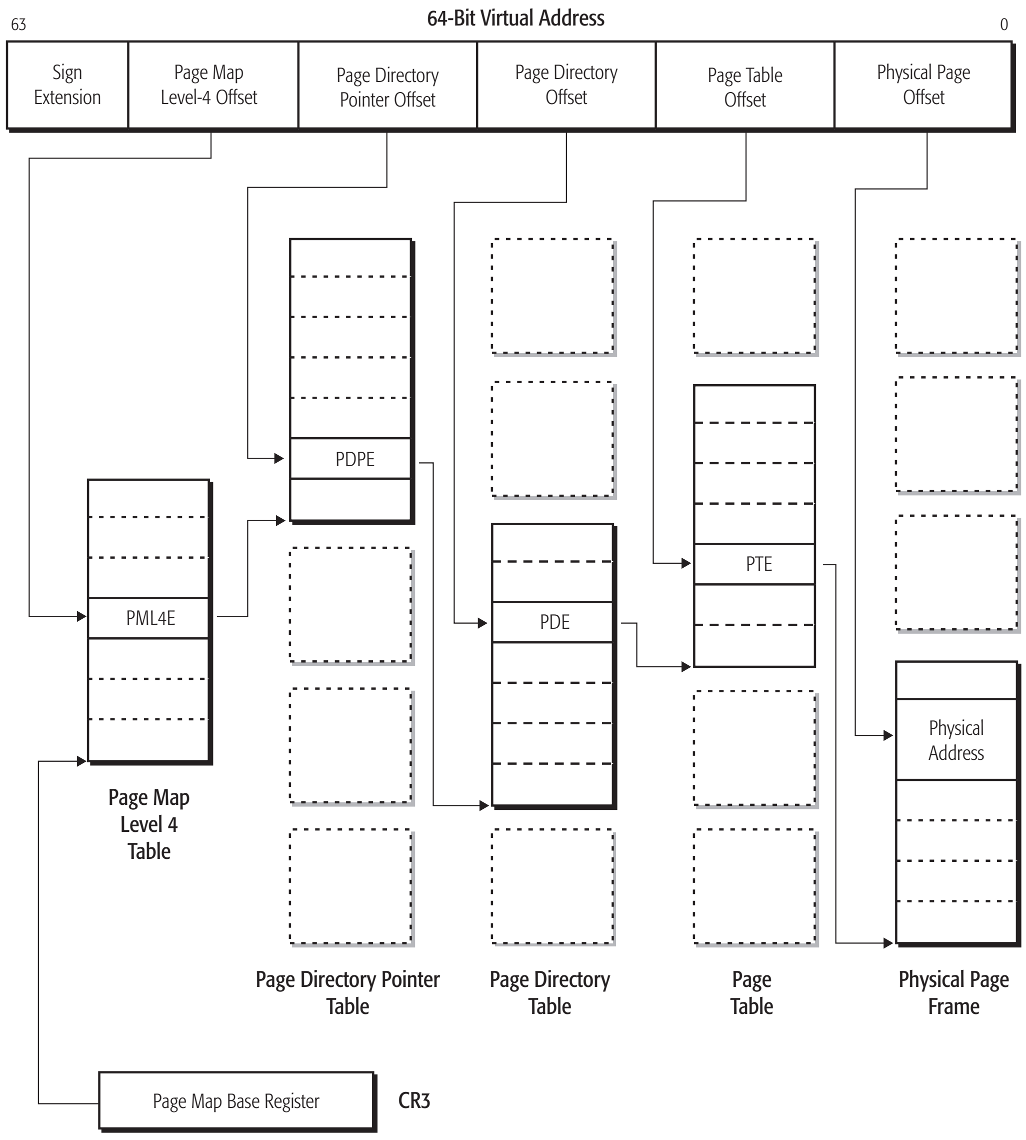


Figure 3.1: 64-bit virtual address translation mechanism using page tables in x86

software uses the page tables to protect this memory by designating the pages as supervisor pages. Such pages are only accessible by system software.

x86 supports 4 KB, 2 MB, 4 MB and 1 GB pages[4]. As show in Fig. 3.1, page translation can use upto four hierarchical levels for virtual address translation. The illustration shows how Long Mode page translation from 64-bit virtual addresses to 52-bit physical addresses is performed. Address translation is performed by dividing the virtual address into six fields. Four of these fields are used as offsets or indices into the level page tables. The CR3 control register always points to the highest level page table (PML4 in the case of Long Mode). The entries read from the page table (using the virtual address offset) point to the base of the next lower level page table. For example, the Page Map Level-4 (PML4) entry points to the base of the page-directory pointer (PDP) table. In addition to the physical address of the next page table, each entry also contains the following page table control and management fields:

1. Present (P) bit
2. Read/Write (R/W) bit
3. User/Supervisor (U/S) bit
4. Page-Level Writethrough (PWT) bit
5. Page-Level Cache Disable (PCD) bit

6. Accessed (A) bit
7. Dirty (D) bit
8. Page Size (PS) bit
9. Global Page (G) bit
10. Page-Attribute Table (PAT) bit
11. No Execute (NX) bit

The page size (PS) bit is present in page-directory entries (PDE) and Long Mode page-directory pointer entries (PDPE). When the PS bit is cleared to 0 in all levels, the lowest level of the page-translation hierarchy is the page-table entry (PTE), and the physical-page size is 4 Kbytes. When the PS bit is set in the PDPE or PDE, that entry is the lowest level (leaf level) of the page-translation hierarchy. If the PS bit is set in the PDE, then the page size is 2M and if it is set in the PDPE, the page size is 1G. The remaining portion of the virtual address then becomes the physical page offset. The PS bit is absent in PML4 entries.

3.2.3 Nested Paging

Fig. 3.2 shows how a page in the linear address space is mapped to a page in the physical address space in traditional (single-level) address translation. Control

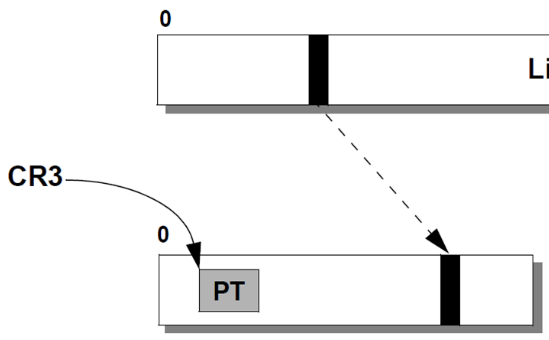


Figure 3.2: Traditional address translation from linear address to physical address by paging

register CR3 contains the physical address of the base of the page tables (PT, represented by the shaded box in the figure).

The SVM nested paging feature provides for two levels of address translation as shown in Fig. 3.3. All physical addresses generated in the guest OS are guest-only physical addresses. They have to be converted by the host OS to host physical addresses before memory read/write operations. The main differences that distinguish it from conventional linear address translation are summarized below.

- Both guest and host levels have their own copy of CR3, referred to as gCR3 and nCR3, respectively.

- Guest page tables (gPT) map guest linear addresses to guest physical addresses. The guest page tables are in guest physical memory, and are pointed to by gCR3.
- Nested page tables (nPT) map guest physical addresses to system physical addresses. The nested page tables are in system physical memory, and are pointed to by nCR3.
- The most-recently used translations from guest linear to system physical address are cached in the TLB and used on subsequent guest accesses.

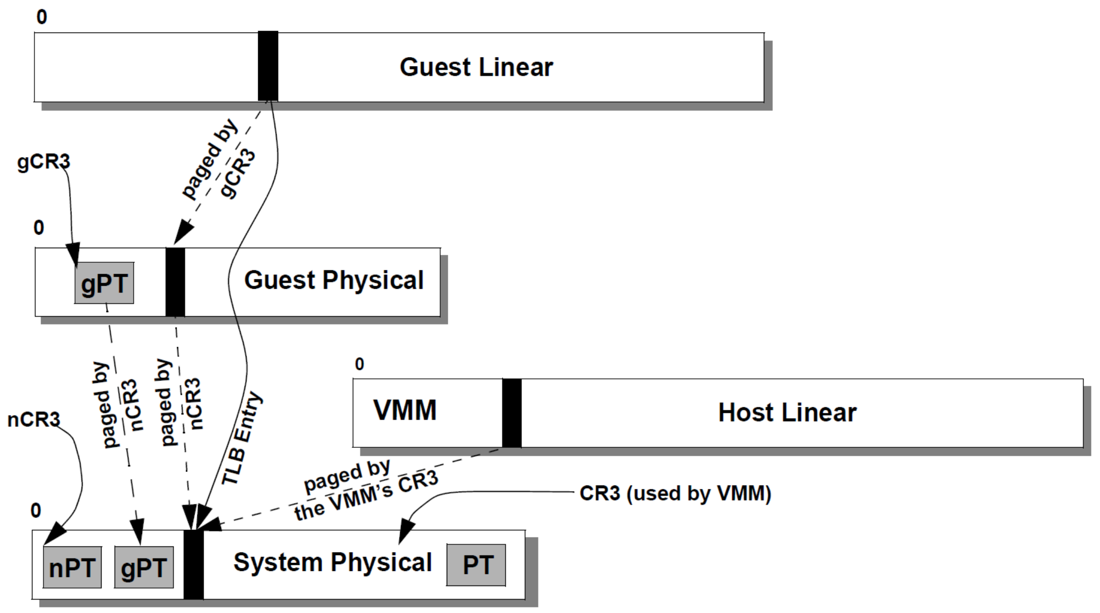


Figure 3.3: Linear address translation by nested paging

All virtual addresses of an OS (guest or host) share a common address space ID(ASID) which helps distinguish them from the address space of other guests and the host. ASID 0 is reserved for the host OS. The guests take values between 1 and 7. When nested paging is enabled, all (guest) references to the state of the paging registers by x86 code (MOV to/from CRn, etc.) read and write the guest copy of the registers; the VMM's versions of the registers are untouched and continue to control the second level translations from guest physical to system physical addresses.

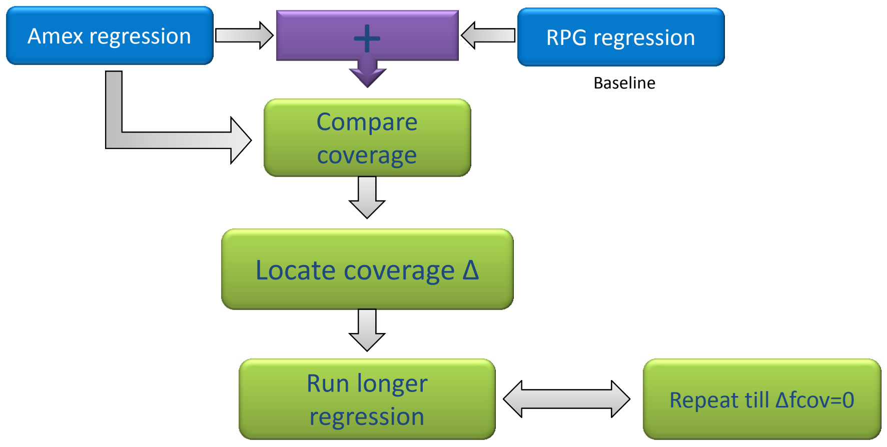


Figure 3.4: Flow used to improve the verification efficiency of Amex using RPG as a baseline.

3.3 Identifying limitations of Amex

Fig. 3.4 describes at a high-level the flow used to compare the two in-house exercisers, Amex and RPG. As discussed in Chapter 2, RPG is an older exerciser. To identify if it provides any unique coverage to SVM nested paging verification, we use it is as a baseline. The first step is to determine all functionalities that RPG is able to verify **exclusively**. In other words, we find coverpoints that are hit solely by RPG. A *coverpoint* or *cover variable* is a design property or micro-architectural event defined in the verification testplan as a necessary micro-architectural event to be tested. In a simulation, the testbench records a coverpoint hit whenever the property is satisfied. Two or more coverpoints can be combined to form a *cross-cover* variable. All coverpoints and cross-coverpoints contain *bins*, which are all the valid values for the coverpoint. A *covergroup* is a set of related coverpoints.

Constrained random tests are generated with analogous inputs using both exercisers and the functional coverage is measured by simulating these tests. In addition to individual coverage values, we calculate the cumulative coverage from running Amex over and above the baseline, RPG, for each regression, as shown in Fig. 3.4. We compare Amex with this cumulative value instead of comparing it with RPG directly. Such a comparison reveals if there is any coverage unique to RPG. The design is simulated until the coverage number of RPG stabilizes. At this point RPG has provided maximum possible coverage for the chosen design sub-space. Any additional coverage is a matter of chance and cannot be guaranteed. All covergroups which RPG uniquely covers are inspected. The coverpoint(s) responsible for the difference is added to a list $\mathbf{H}$. These are the coverage deficiencies of Amex as compared to RPG for the chosen feature $\mathbf{T_i}$. If there are no such coverpoints (**H** is null), we can conclude that tests generated by RPG is a subset of Amex for verifying the properties included in the current verification plan.

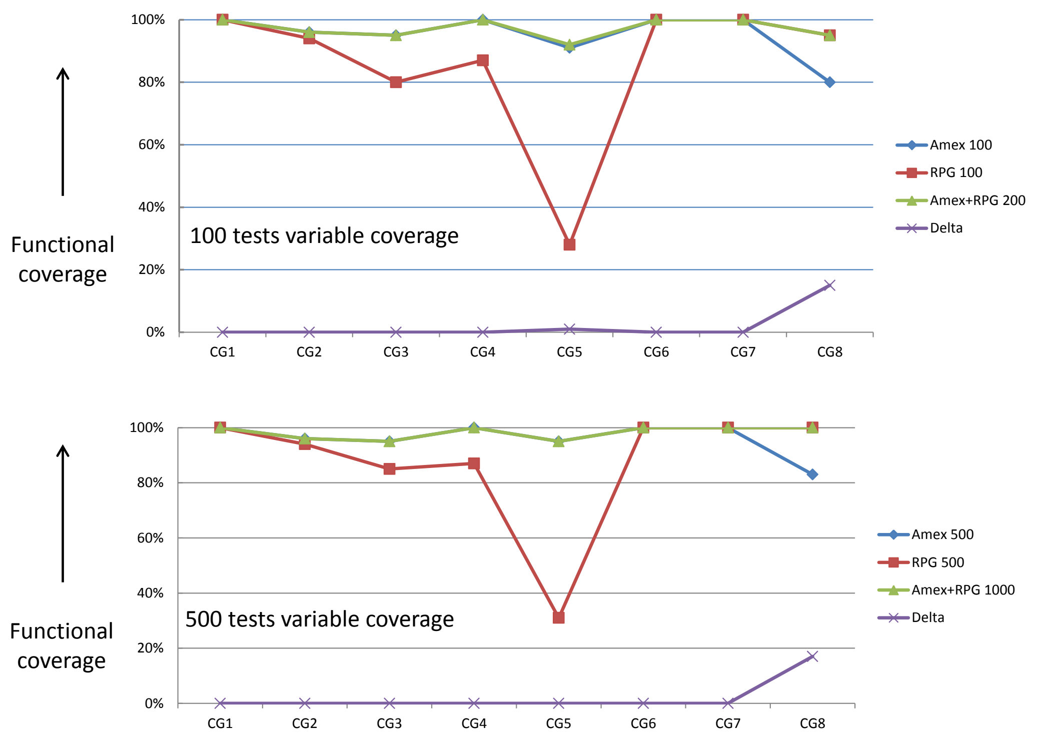


Figure 3.5: Comparison of the functional coverage of SVM nested paging coverage variables using 100 (top) and 500 (bottom) randomly generated tests

3.3.1 Experimental results

Fig. 3.5 compares the functional coverage of Amex and RPG over 8 groups containing only cover variables (no crosses). Each covergroup is a collection of related coverpoints. For example, one of the covergroups covers page faults occurring during a nested page walk. This covergroup is made up of coverpoints that consider all possible page faulting scenarios, such faults in the guest, faults in the host, different kinds of page-faults (page not present, write permission, user/supervisor, etc), page faults on the i-side, etc.

- In both the graphs, Amex performs at least as good as RPG in covergroups CG1-CG7. This is indicated by the red (square) line being the same as or below the blue (diamond) line.

- The only exception is CG8. CG8 has higher RPG coverage for both 100 tests and 500 tests.

- The cumulative coverage of Amex and RPG is indicated by the green line (triangle). The difference between this and Amex is shown in purple (x). There is a delta of 15% for CG8 and very marginal value for CG5 in the first regression. In the second regression, however, the delta for CG5 disappears. But the delta for CG8 increases.

The results of these two regressions is shown for the cross-coverage variables in covergroups CG1 through CG5 in Fig. 3.6. The other three covergroups do not contain any crosses. A cross is a combination of two or more coverpoints. For example, by combining a page-fault coverpoint containing 5 bins with guest/host pages, we get 10 bins that give all possible combinations of page-faults over both guest and host pages.

- There is a delta between Amex and the cumulative coverage values in all but CG3.
- Between the first and second regression, the delta increases for CG1 and CG4, but decreases for CG2 and CG5.

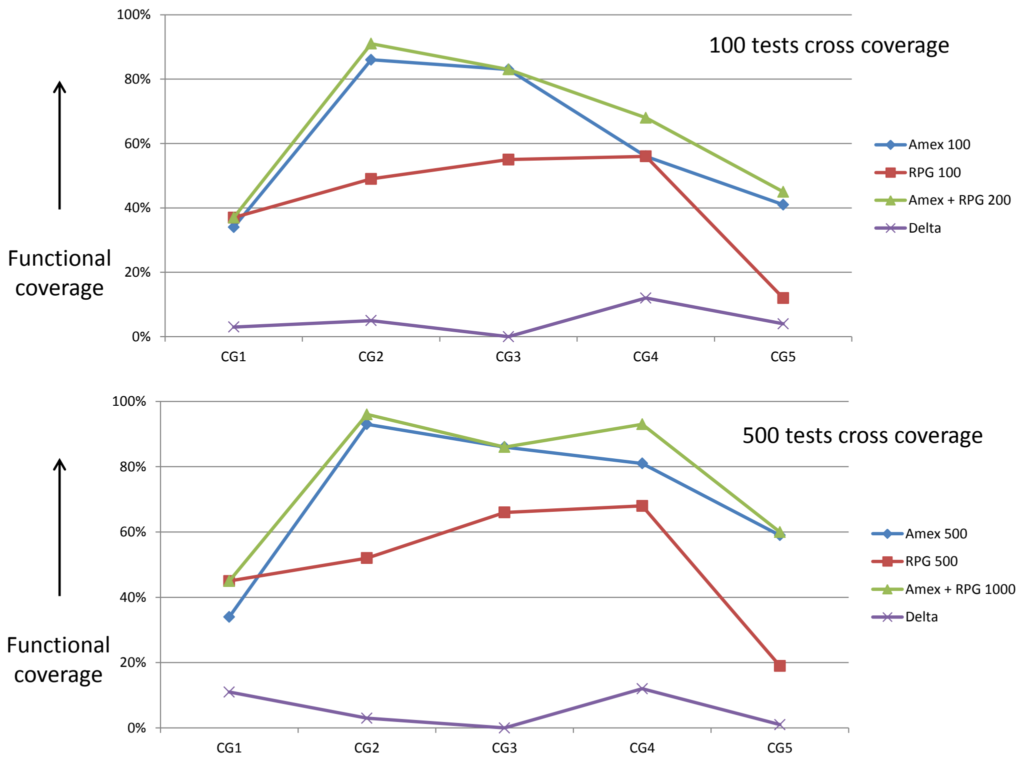


Figure 3.6: Comparison of the functional coverage of SVM nested paging cross-coverage variables using 100 (top) and 500 (bottom) randomly generated tests

To observe the steady state condition, we run longer regressions until there is no new coverage. The last useful test in a regression indicates how frequently previously uncovered bins are hit. Table. 3.1 shows the last useful test from four Amex (top) and four RPG (bottom) regressions across the eight SVM nested-paging covergroups. The tests were ordered based on their time of completion. The four rows in each table corresponds to regressions of 100, 500, 2000 and 5000 tests. The value 97 corresponding to CG1 and Amex 100, for example, indicates that the last 3 (of 100) tests did not hit any new bins. The 97th test was the last useful test with respect to CG1. For each regression, the last useful test overall is indicated by a shaded red cell. For example, the 4726th test was the last useful test in the 5000 test RPG regression. In other words, the last 274 tests did not contribute to the verification of any of the features covered by the eight SVM covergroups. The table shows us how the last useful test occurs farther away from the end of a regression as its length increases. For example, in case of Amex, it occurs 700 from the end (4300 of 5000) as opposed to 96 (1904 of 2000) as we increase the regression length from 2000 to 5000. Similarly, it occurs 274 tests prior to the regression end (4726 of 5000) as opposed to 12 tests (1988 of 2000) in a 5000 tests regression as compared to a 2000 test regression. The fact that close to 300 tests did not yield any new coverage gives us the confidence that 5000

Table 3.1: Last useful test from Amex (above) and RPG (below) SVM regressions

	CG1	CG2	CG3	CG4	CG5	CG6	CG7	CG8
Amex 100	97	97	79	58	82	31	12	76
Amex 500	261	495	271	325	478	63	46	211
Amex 2000	783	1904	776	1695	1797	128	66	1649
Amex 5000	1148	4300	1611	1635	3333	64	62	4228

	CG1	CG2	CG3	CG4	CG5	CG6	CG7	CG8
RPG 100	81	96	99	82	99	18	8	61
RPG 500	215	468	469	293	486	12	7	57
RPG 2000	1988	1721	180	41	397	7	24	76
RPG 5000	4445	4726	1854	97	692	9	6	176

tests is a reasonably long regression to assume steady state behavior of both the exercisers.

An interesting point to note is that there can be incidental coverage of *non-targeted* features, but they are disregarded. For example, SVM behavior can be triggered by a Streaming SIMD Extensions (SSE) regression (unless SVM is explicitly prohibited). Such unintentional coverage is inconsistent and a matter of chance. From a design verification standpoint any kind of coverage is useful coverage. The intent of the regression triggering a behavior is immaterial as long as the test is valid for the processor configuration. But from an exerciser analysis standpoint, we will ignore all such coverage as it is fortuitous and unreliable.

3.3.2 Analysis of results

Our experiment on SVM nested-paging shows that for three covergroups, the RPG-specific coverage is more than Amex. In all other cases, RPG-specific coverage is equal to less than that of Amex. The following three covergroups have to be analyzed for low Amex coverage:

1. CG8 variable coverage
2. CG1 cross coverage
3. CG4 cross coverage

CG8 variable coverage

RPG specific coverage of CG8 drops to 5% at steady state, as seen in Fig. 3.7. CG8 consists of twelve coverpoints pertaining to the instruction cache (IC) and i-side translation look-aside buffer (TLB). Of the twelve coverpoints, two coverpoints are covered 100% by RPG but only get 37% and 25% coverage from Amex tests. Both coverpoints have eight bins each. All other coverpoints in this covergroup are covered identically by both exercisers. Each bin corresponds to the different ASID values of the linear address translated using the I-side TLB. Since there are eight possible ASID values, there are eight bins in each coverpoint. Refer back to Sec. 3.2.3 for details regarding ASID values. The i-side TLB has 4 ways. These two coverpoints refer to the highest two ways - way2 and way3. The lack of Amex coverage is caused by its inability to produce linear address collisions for the same TLB index; leaving the higher ways to the i-side TLB unused.

An interesting observation in Fig. 3.7 is that CG3, which previously did not show any delta, has a 5% delta. This delta is caused by a single bin: page faults at the PML4 level. As discussed in Sec. 2.6, Amex delegates page table creation and other memory allocation tasks to an external program. This limits its flexibility in creating page tables. For example, it can only request for page faults in a leaf-level entry of the hierarchical page table. Since PML4 is never the leaf-level entry (PDP is the leaf-level entry for 1G pages), Amex cannot directly set up the page tables to fault at the PML4 level. RPG is not limited by any such restriction.

CG1 cross coverage

The cross-variable coverage results of regressing 5000 tests of Amex and RPG are shown in Fig. 3.8. We can see that the difference in CG1 cross-variable coverage has increased drastically from 10% to 40%.

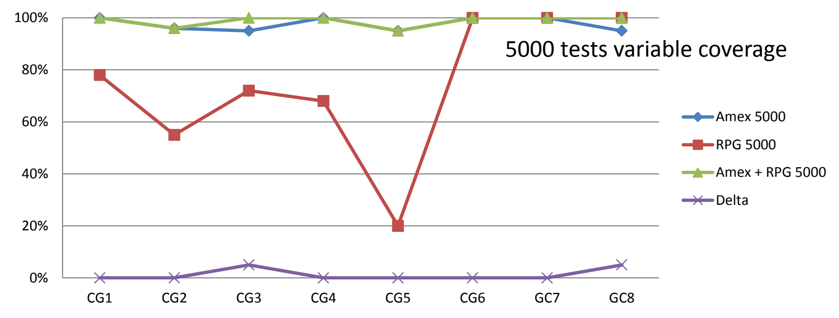


Figure 3.7: Functional coverage of SVM nested paging coverage variables using 5000 randomly generated tests

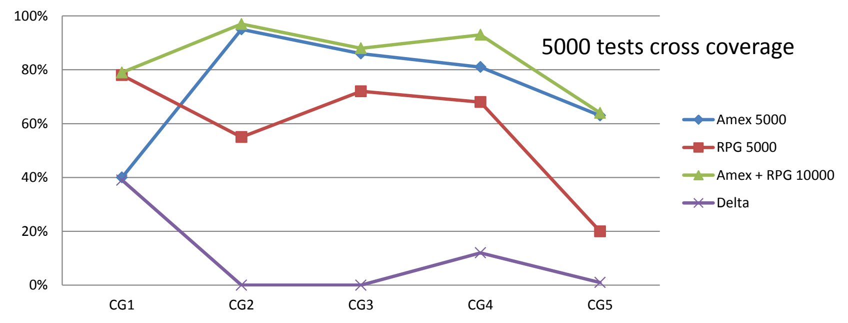


Figure 3.8: Functional coverage of SVM nested paging cross-coverage variables using 5000 randomly generated tests

The higher coverage of CG1 is because of a single cross coverage term `cp_LS_Curr_ASID_X_utlb_lkup_asid`. This term is a cross of two variables:

1. `LS_curr_asid`: Current Address Space ID(ASID)
2. `utlb_lkup_asid`: ASID of the virtual address being looked up in the i-side TLB

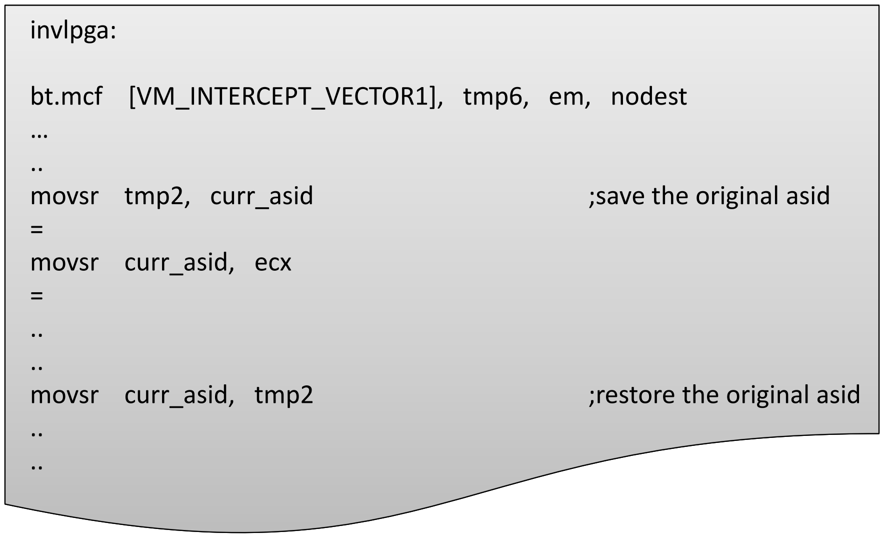
Typically, these two values are identical as the linear address being translated to a physical address belongs to the address space of the OS that is running. In instances where the processor switches between the host and one or more guest OSs, other combinations are also possible. The ASID of the host is 0 and that of a guest is a value between 1 and 7. The different combinations of these two variables is summarized in Table 3.2

Table 3.2: Possible combinations of current ASID and lookup ASID

Current ASID	**Lookup ASID**	**Context**
Zero	Zero	Host translating own LA
Zero	Non-zero	Host translating guest LA
Non-zero	Zero	Nested-paging guest linear translation
Non-zero	Non-zero	Page fault

When nested paging is enabled, every guest physical address has to be translated to a real (host) physical address to perform a memory read/write (refer to Sec. 3.2.3 for details). Amex hits were confined to the first three combinations listed in Table. 3.2. Since accessing another guest's address space causes a page-fault, Amex does not attempt such translations.

Consider the first RPG regression of 100 tests (Fig. 3.6). The 3% additional coverage provided by RPG, comes from 2 unique hits of the fourth type: $asid[1]\_asid[5]]$ and $asid[2]\_asid[5]$. These are valid bins and such accesses should trigger page faults because of an access violation. Of the 100 tests generated by RPG that cover 37% of the cross-coverage variables in CG1, only one test was responsible for both the atypical hits. Contrary to expectations, the test only ran guest5 and did not run guest1 or guest2. The test also did not page-fault due to an access violation, suggesting a case of false coverage recording.



```
invlpga:

bt.mcf   [VM_INTERCEPT_VECTOR1],   tmp6,   em,   nodest
...
..
movsr   tmp2,   curr_asid                              ;save the original asid
=
movsr   curr_asid,   ecx
=
..
..
movsr   curr_asid,   tmp2                              ;restore the original asid
..
..
```

Figure 3.9: Snippet of microcode of the INVLPGA instruction.

The cause of this false coverage was traced to temporary changes of register `LS_Curr_ASID` to values other than the ASID value of the current OS. This switch was caused by the execution of a specific instruction called INVLPGA. The INVLPGA instruction invalidates TLB mappings for a given virtual page and ASID [4]. As can be seen in Fig. 3.9, the microcode temporarily changes the register value from the current ASID to the ASID of the TLB entries to be flushed. Because the coverage points was not well qualified, it recorded this change of value as legitimate coverage. This hypothesis was confirmed by generating a second test with an INVLPGA instruction that used an ASID value that did not occur in the test.

CG4 cross coverage

There is one cross-coverage term `cross_np_pg_guest_host_sizes` defined in CG4. x86 supports 4 page sizes: 4K, 2M, 4M and 1G. This cross-coverage term covers all 16 possible combinations of guest and host page sizes.

From Table. 3.3, you can see that the 5000 Amex generated tests cover only 13 of the 16 valid bins. The RTL simulation resulted in a total of 170,061 hits on this coverpoint. 3 bins were not hit by any of the 5000 tests viz. `sz1g_sz4k, sz4k_sz1g and sz4m_sz1g`. One way to remedy this is to rerun Amex with the following modifications to its constraints:

Run1 Increase likelihood of 1G host, 4K guest and 4M guest pages. Reduce likelihood of all other page sizes

Run2 Increase likelihood of 1G guest and 4k host pages. Reduce likelihood of all other page sizes

Table 3.3: Coverage summary of 5000 Amex generated tests for the guest-host page size cross-coverage variable

Guest PgSize	Host PgSize	Hits (% of total)	Tests (% of total)
sz1g	sz1g	1471 (1%)	190 (4%)
sz1g	sz2m	80 (0%)	17 (0%)
sz1g	sz4m	25 (0%)	7 (0%)
sz2m	sz1g	23 (0%)	6 (0%)
sz2m	sz2m	6719 (4%)	694 (14%)
sz2m	sz4k	1141 (1%)	299 (6%)
sz2m	sz4m	827 (0%)	156 (3%)
sz4k	sz2m	2663 (2%)	370 (7%)
sz4k	sz4k	153150 (90%)	2653 (53%)
sz4k	sz4m	1649 (1%)	232 (5%)
sz4m	sz2m	35 (0%)	3 (0%)
sz4m	sz4k	241 (0%)	49 (1%)
sz4m	sz4m	2037 (1%)	156(3%)

But in Amex, the guest and host page sizes cannot be controlled separately. This makes getting directed coverage for the remaining three bins difficult. The same parameter has to be used to control both guest and host pages sizes. The smaller page sizes (4K and 4M) are more likely to be picked in favor of 1G pages since they are more easily accommodated within the existing blocks. So `sz4k_sz4k` is more likely to occur than `sz1g_sz4k`.

3.3.3 Re-constraining exercisers

We now fine-tune the exerciser constraints to generate directed tests targeted specifically at the coverage holes. In instances where the targeted events are complex and the relation of the test to the test constraints is non-obvious. It can take several iterations to get the right combination. One way to speed-up this process is to extract knowledge from the simulation data to guide the test constraining process[18, 11, 29, 14]. The extracted knowledge can be then be used for test template refinement to improve design coverage. As seen in the case of CG4, it may not be possible to generate good constraints because of exerciser limitations. To remedy this, we have to resort to test filters external to the exerciser. Generating a large amount of tests and filtering out novel tests that trigger a target behavior, prior to RTL simulation, has the same effect as constraining the exerciser.

3.3.4 Fairness of comparison

This experiment was repeated twice to confirm that the observed numbers and trends were consistent. It is important to note that these comparisons were made by test numbers (and not simulation cycles or run-time) even though the tests generated by the two exercisers are not necessarily of the same length. Because we are doing a comparison based only on verification completeness in this paper, test lengths have been ignored. Coverage numbers can be normalized by number of simulation cycles if there are performance concerns, to give a better sense of run-time utilization. Our experiments did not show any difference in the results when comparisons were made based on simulation cycle count. Results have hence not been provided.

3.4 Summary

In this chapter, we compare the verification effectiveness of Amex and RPG using functional coverage as a metric. The comparison was based on their ability to verify the SVM nested-paging functionality at the core level, for a single core simulation setup. We use RPG as a baseline to evaluate its contribution to verifying this feature.

Regressions of 5000 tests each show that Amex provides higher coverage than RPG across 8 covergroups containing 148 coverpoints, defined in the existing verification plan. This confirms our understanding from Chapter 2 that Amex is more efficient at test generation. There are three coverpoints hit only by RPG.

One of this was falsely recorded coverage, caused by a poorly defined coverpoint. The other two are caused by Amex's design limitations. We propose to overcome these limitations by using external test filters.

Chapter 4

ISA simulation based test filtering

4.1 Introduction

Coverage closure requires multiple iterations of test generation. On each iteration, the test templates are modified to cover any remaining design space. This is possible as long as there are parameters and/or constraints to control the targeted behavior either directly or indirectly. For example, consider the cross-coverage term of guest and host page sizes discussed in Sec. 3.3.2. There are 16 possible combinations of guest and host page sizes. From the results in Table 3.3, we can see that this regression covers only 13 of the 16 valid bins. 3 bins were not hit by any of the 5000 tests viz. `sz1g_sz4k, sz4k_sz1g and sz4m_sz1g`.

Since Amex guest and host page sizes cannot be separately controlled, it makes getting directed coverage for the remaining three bins difficult. The same parameters have to be used to control the likelihood of both guest and host pages sizes. The smaller page sizes (4K and 4M) are more likely to be picked in favor of 1G pages since they are more easily accommodated. So `sz4k_sz4k` is more likely to occur than `sz1g_sz4k`.

When an exerciser design doesn't provide a method to accurately target a desired behavior, the density of targeted tests remains low($<$3% of generated tests). Simulating thousands of tests to identify these novel tests is a waste of simulation cycles. In practice, if one or more bins remain uncovered after a few attempts, verification engineer resort to writing directed tests manually. This however requires significant effort and tests have to separately handwritten, directed at each bin. Handwritten tests cannot be created in bulk and lack enough test randomization.

When desired constraints are not available, a test generator itself can be modified. We cannot, however, guarantee that the scope of the modified test generator is a superset of the existing version. For a test generator, like Amex or RPG, that lacks a constraint solver, the effect of added design constraints/modifications cannot be evaluated. It is possible for a modification made to enhance its ability to generate a particular type of test to negatively impact its ability to generate some other kind of test. Instead of making modifications to an exerciser when the desired constraints are not available, we build a test filter in the form a wrapper. This guarantees that there is no unfavorable impact on test generatorŠs performance.

4.1.1 Related work

Closing the gap between test generation and coverage closure has remained the foremost challenge of simulation based verification. Test generation programs work independent of design simulation, and they do not receive direct feedback on the quality of generated tests. Converting coverage requirements into directives for test generation requires considerable manual effort. Early attempts to solve this issue involved describing the processor implementation control as a Finite State Machine(FSM) to derive transition coverage [41, 28]. This approach does not scale well with the complexity of modern microprocessors. Attempts have since been made to automate constraint generation[11, 29, 23, 44]. [25] proposes an approach for automatic extraction of word-level model constraints from the behavioral verilog HDL description. The scenarios to be tested are also expressed as constraints. The model and the scenario constraints are solved together using an integer solver to arrive at the necessary functional test.

The extraction of testing knowledge from simulation data to guide the test constraining process using machine learning algorithms such ILP[18], C4.5 decision trees [32] and CN2-SD[14] has received a lot of attention lately. Fine, Freund, Jaeger et.al.[20] used machine learning to research the impact of design initial state on test generation and coverage. A fully automated push-button solution for automated test generation is a remote possibility. Other approaches include a hybrid of formal and simulation based verification[9, 27, 16].

In our approach, we use testing knowledge to filter tests using a wrapper. The novelty of our approach lies in imparting the testing knowledge to an external filter; instead of using it directly to modify test generation through exerciser directives. Tests generated with the existing setup are subjected to filtering based on extracted rules that describe RTL behavior in terms of architectural values. This gives us several advantages. First, we do not have to convert the extracted rules into directives for test generation. This process is non-trivial since each exerciser has its own format for specifying test inputs. Secondly, it gives us better control over the selection of tests. Test inputs often do not deliver results as expected because of constraint conflicts within an exerciser. Using the rules for an external filter gives us a better understanding of how the constraints regulate test generation since it is independent of the test generation process. This also means that this approach can be used to filter tests from any exerciser without any modification.

4.2 Simulation time reduction

In such instances where user inputs do not adequately constrain tests, having the ability to filter out unlikely candidates prior to RTL simulation lets us generate directed tests with minimal use of simulation resources. Even an approximate method that can filter out the most likely candidates can reduce the need to run lengthy regressions or manually write directed tests.

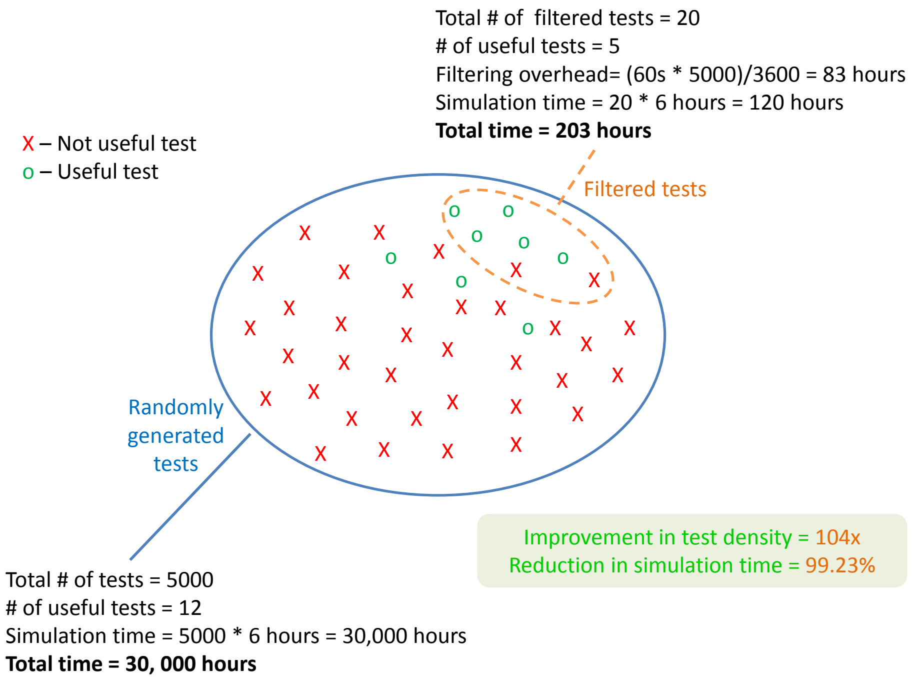

Total # of filtered tests = 20
# of useful tests = 5
Filtering overhead= (60s * 5000)/3600 = 83 hours
Simulation time = 20 * 6 hours = 120 hours
Total time = 203 hours
X – Not useful test
o – Useful test
Filtered tests
Randomly generated tests
Improvement in test density = 104x
Reduction in simulation time = 99.23%
Total # of tests = 5000
# of useful tests = 12
Simulation time = 5000 * 6 hours = 30,000 hours
Total time = 30, 000 hours

Figure 4.1: An illustration of the benefits of pre-simulation test filtering

Suppose we rerun the exerciser with modified constraints and generated 5000 more tests, of which 12 tests hit the remaining 3 bins, as illustrated in Fig. 4.1. Even if we identified only 5 of these 12 tests along with 15 other false positives, we would be increasing the test density by a factor of $(\frac{5}{20}/\frac{12}{5000}) = 104$x by simulating only the 20 tests (as opposed to all 5000 tests). A typical processor core-level simulation takes roughly 6 hours to complete. If we can filter tests based on ISA simulation traces, with less than a 30s overhead, it will take us at the most 60 seconds to analyze a test. This reduces the total simulation time (including filtering overhead) down to $\frac{203}{30000} = 0.67\%$ of the original. This is a substantial reduction in test redundancy and simulation time.

We shall now discuss one such method to approximate test behavior prior to RTL simulation. All experiments have been performed on AMD's latest x86 ISA based microprocessor which supports upto 4 cores per cluster. The processor is capable of super-scalar and out of order execution. Each core has a 32KB instruction cache and a 32KB data cache. A 2MB L2 cache is shared between the cores. Experiments have been performed using a single core for the sake of simplicity.

4.3 Instruction-set architecture (ISA) simulation based test filtering

One way of estimating test behavior prior to RTL simulation is through architectural simulation. Fig. 4.2 illustrates this idea. The conventional approach

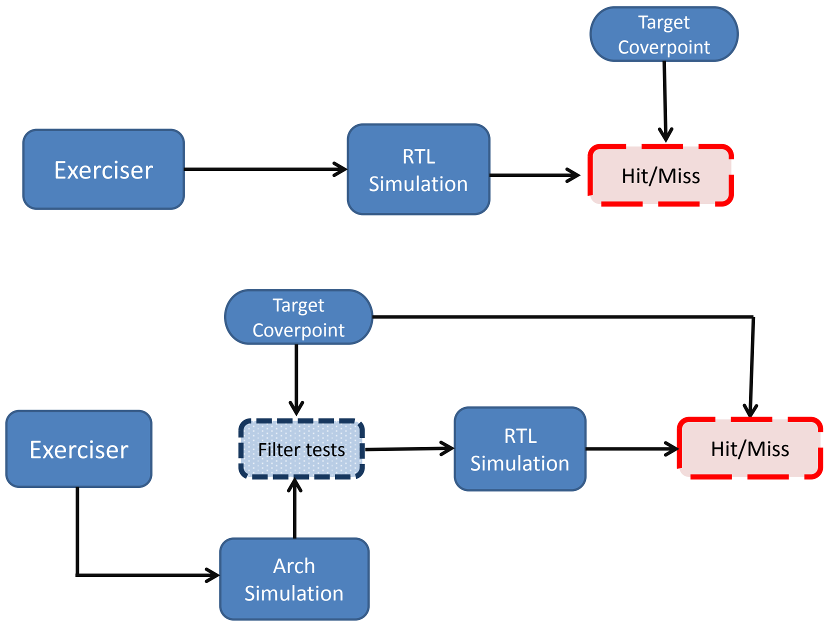


Figure 4.2: a) (Above) Conventional coverage closure flow b) (Below) Architecure simulation based test filtering

(Fig. 4.2a) is to assess functional coverage *after* simulating tests. Alternatively, we can learn microarchitectural behavioral rules from past regressions. This testing knowledge is in the form of rules that relate microarchitectural events to known values, such as architectural values as shown in Fig 4.2b. Only those tests that contains patterns known to trigger the target behavior are simulated on RTL.

4.3.1 KOS

AMD's x86 ISA simulator is called KOS. KOS serves as the golden reference model for standalone simulations (architecture model only) and co-simulations (architecture model vs RTL simulations). KOS also supports the use of extensions, that interact with its simulation flow. Test selection based on KOS simulations can therefore be done in one of two ways.

1. KOS wrapper: An executable that includes standalone KOS, that gets control at the end of a KOS simulation.

2. KOS plug-in: A library that an existing KOS executable can load, that gets control at the end of each KOS simulation-step.

Exercisers use KOS to validate tests after generation. Plug-ins intercept KOS at every simulation step and compare the architecture state with expected values. A wrapper uses KOS simulation trace after its completion. Either ways, KOS based filtering allows us to seamlessly integrate test filtering into the test generation process. Let us now examine the use of KOS based test filtering for two coverpoints, both of which are triggered by less than 1% of the SVM tests.

4.3.2 Example 1: Guest with cache disabled

In this example, we test guest memory-types where nested-paging is enabled. (Refer to Sec. 3.2 for details of SVM and nested-paging). A *memory-type* is an attribute that can be associated with a specific region of virtual or physical memory. A memory-type designates caching and ordering behaviors for loads and stores to addresses in that region. Most memory types are explicitly assigned, although some are inferred by the hardware from current processor state and instruction context. For example, for a memory designated as *uncacheable* (UC), reads and writes are not cacheable. Reads from UC memory cannot be speculative. Write-combining to UC memory is not allowed. Reads from or writes to UC memory cause the write buffers to be written to memory and be invalidated prior to the access to UC memory.

Suppose we want to specifically generate tests that run a guest with its cache disabled. To do this, a test should satisfy the following criteria:

1. SVM has to be enabled
2. Paging has to be enabled
3. Nested paging has to be enabled
4. Cache Disable (CD) bit in the guest CR0 control register must be set
5. Test should switch context to a guest at least once

All the above 5 properties are architecturally defined. Tests that comply with these conditions can be filtered unambiguously by analyzing an architecture simulation trace. Guest settings are configured in the Virtual Machine Control Block(VMCB). VMCB describes the virtual machine (guest) to be executed. VMCB contains a list of instructions or events in the guest (e.g., write to CR3) to intercept, various control bits that specify the execution environment of the guest or that indicate special actions to be taken before running guest code, and guest processor state (such as control registers, etc.). The first 4 conditions can be verified by examining the VMCB used to run a guest. The VMRUN instruction marks the execution of a guest instruction stream, which is the last condition.

Identifying existing novel tests

The filter was first applied on simulated tests. Since the result is already known from the coverage report of the simulated tests, the effectiveness of this filtering mechanism can be evaluated. The results are shown in Table. 4.1. There are no tests present in the first Amex regression. All tests present in the other three regression are identified.

Table 4.1: Results of filtering tests (from a past regression) that run guests with cache disabled

Regression	**Amex 100**	**Amex 500**	**Amex 2000**	**Amex 5000**
Tests present	0	1	7	12
Tests identified	0	1	7	12
Escapes	0	0	0	0

Generating new novel tests

Additional tests are generated by applying the filter as a wrapper around the test generator. The results of generating new tests are summarized in Table. 4.2.

4 tests were identified from a pool of 1000 tests. All 4 tests were simulated on RTL to confirm their behavior.

Table 4.2: Result of generating new tests that run guests with cache disabled

Generated tests	**Filtered tests**	**Novel tests**
1000	4	4

Because the target property can be defined as a combination of existing architectural properties, there are no escapes or false positives.

4.3.3 Example 2: Page-not-present fault in PDPE

The previous example illustrated how architectural properties can be accurately identified using KOS. Let us consider yet another architectural property - page-not-present page faults at the PDP level, as described in Fig. 4.3. We use the same address translation mechanism, as defined in Sec. 3.2.2. This fault happens when the page containing a virtual address is not present in memory and has to be fetched from disk. The fault is indicated by the page table entry P-bit being set to 0. This ends the table-walk. The page-directory and page-table entries(shown by dashed lines) are not accessed.

This fault can be identified by scrutinizing the table-walks simulated by KOS. In an instance of a page-fault, KOS indicates the source of the fault.

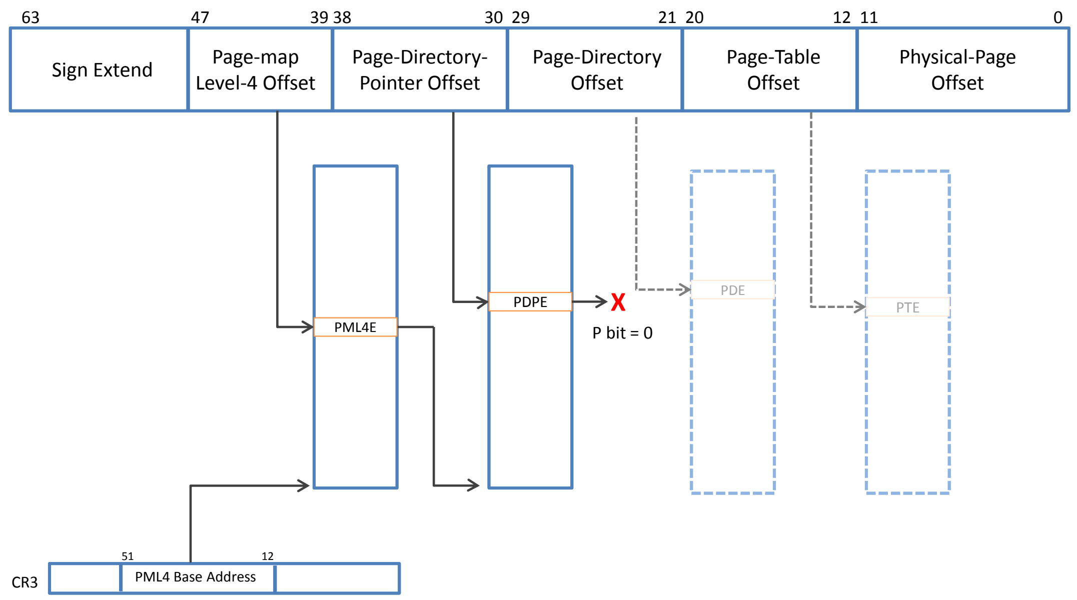


Figure 4.3: Page-not-present fault occuring at the PDP level during a long mode linear address translation

Identifying existing novel tests

Unlike the previous example, when this filtering mechanism is applied to known tests, there are escapes. The results are shown in Table. 4.3. There are no tests present in the first regression, while the other three regression contain 1, 17 and 39 novel tests, respectively. When the simulation traces of these tests are parsed, only 2 tests from *Amex 5000* were recognized. Most of the tests escape detection.

Table 4.3: Result of KOS-based filtering of tests (from a past regression) that trigger a page-not-present fault at the PDP level

Regression	**Amex 100**	**Amex 500**	**Amex 2000**	**Amex 5000**
Tests present	0	1	17	39
Tests identified	0	0	0	2
Escapes	0	1	17	37

The large percentage of escapes is caused by speculative execution of code. By design, pipelined processors begin executing instructions even before the preceding instructions retire. They are executed out of order if they do not have any resource dependencies. When it encounters a branching instruction, the branch predictor determines the path to execute. Once the branch has been resolved, the processor state is resolved. If a branch is predicted correctly, the speculatively executed instructions are retained. In cases where a branch is mispredicted or an exception is encountered, the (speculatively) executed instructions are discarded. Even though instructions are executed out-of-order or speculatively, they are re-ordered at the time of retirement. Therefore, at all times, the processor maintains the correct architectural state irrespective of the performance optimizations used. These optimizations are done transparent to the software. Since KOS only models the architectural states, it is oblivious of speculative code execution. Therefore, any properties of speculatively executed code cannot be tested using KOS. For example, of the 39 tests in *Amex 5000*, two tests page-fault non-speculatively at the PDP level (at least once). This allows us to detect these tests via KOS-based filtering. The other 37 tests page-fault purely speculatively; thereby evading detection by KOS.

4.3.4 Conclusion

Since architectural values are visible to software, they can be easily controlled. But the micro-architecture is transparent to software. This makes micro-architectural properties harder to control and verify. The challenge of microprocessor verification lies predominantly in testing micro-architectural features and their interactions. Since ISA simulators are implementation agnostic, their traces cannot be used to predict a wide majority of test behaviors without knowing the underlying relation. For instance, since ISA simulation does not take microarchitecture into account, they are unaware of cache hits, misses and cacheline fetches. Similarly, ISA simulators are unaware of TLB insertions, pipeline states and speculative execution.

To filter microarchitectural events, we need to develop a better understanding of how they can be controlled by known values i.e. architectural values. This understanding of how software can be used to control RTL behavior is often referred to as testing knowledge[1, 32]. Testing knowledge is highly design-specific. It can even change during the same project due to RTL changes. Verification engineers apply testing knowledge through test generator templates to guide the test generation process. In the following sections, we extract testing knowledge from ISA simulation traces. We will assess the limitations of imparting this testing knowledge into KOS based filtering.

4.4 Extracting testing knowledge

Testing knowledge can be gathered in multiple ways. One way to do this is to use the information from a formal model of the processor. This is impractical since formal models are a very low level description of the design. Another approach is to parse implementation reference manuals. Since technical documentations lack a defined structure, they are not machine readable. The most prevalent approach to gathering testing knowledge has been through the use of machine learning. Machine learning techniques are applied to simulation data represented in a vector format to glean meaningful rules that explain design behavior.

Microarchitectural events can be treated as the result of a properly timed trigger applied from a specific processor state. For example, a stack overflow is caused by a push to a stack which is already full. An event can therefore be described in terms of architectural states and the applied stimulus. This description can be used to produce the right constraints for directed test generation. We organize all the relevant information from a simulation trace into a two-dimensional dataset. This data is then subjected to machine learning to identify rules to describe design properties. In this section, we will discuss this process in three stages:

1. Extracting features from a simulation trace

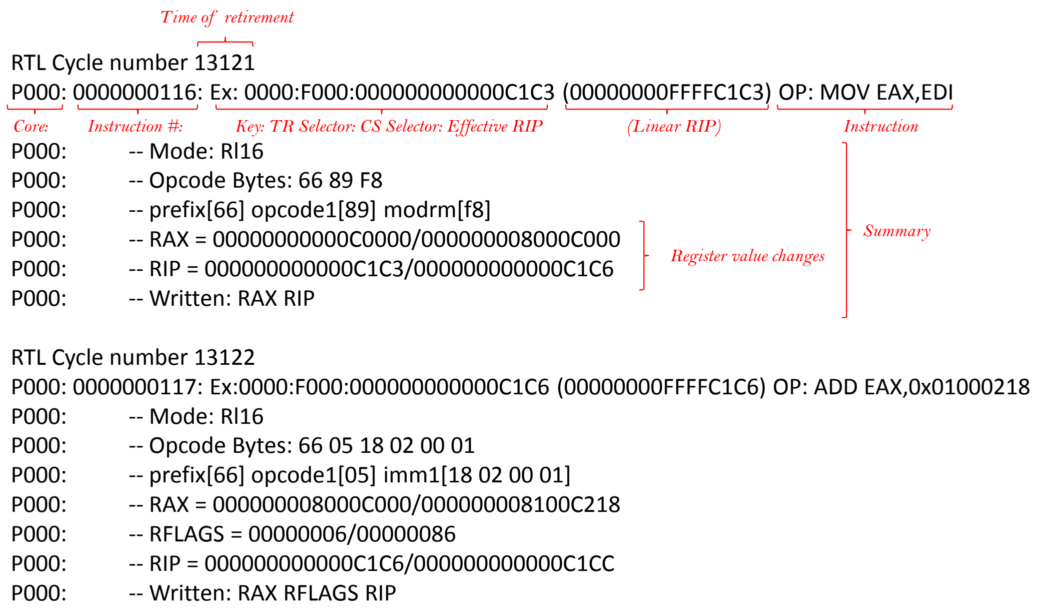
2. Generating a dataset

3. Rule extraction

4.4.1 Extracting features from a simulation trace

Suppose we want to understand what conditions cause the branch predictor to mispredict. The first step is to extract all the information relevant to branch prediction from a simulation trace. This trace is generated by simulating an assembly program on an RTL, which in our case is a processor core. We use randomly generated assembly programs containing roughly 10,000 instructions each. RTL verification is done by co-simulating an RTL with its architectural model. The output of an RTL simulation, therefore, includes a cycle accurate log of all the architectural state changes. All state values measurements are done at the time of instruction retirement.

Fig. 4.4 shows us how a KOS simulation trace looks like. All the information regarding an instruction and the state changes caused by it are summarized in a block. The first line of the block indicates the cycle in which the instruction is retired. This is followed by several lines describing the various *operations* performed as a result of executing the instruction. To identify the nature of the operation being described, each line is assigned a *key*. For example, information about the instruction itself is indicated by the key **Ex**. As shown in the figure, this line indicates the core on which the instruction was executed, the instruction count in the


Time of retirement
Core:
Instruction #:
Key: TR Selector: CS Selector: Effective RIP
(Linear RIP)
Instruction
Register value changes
Summary

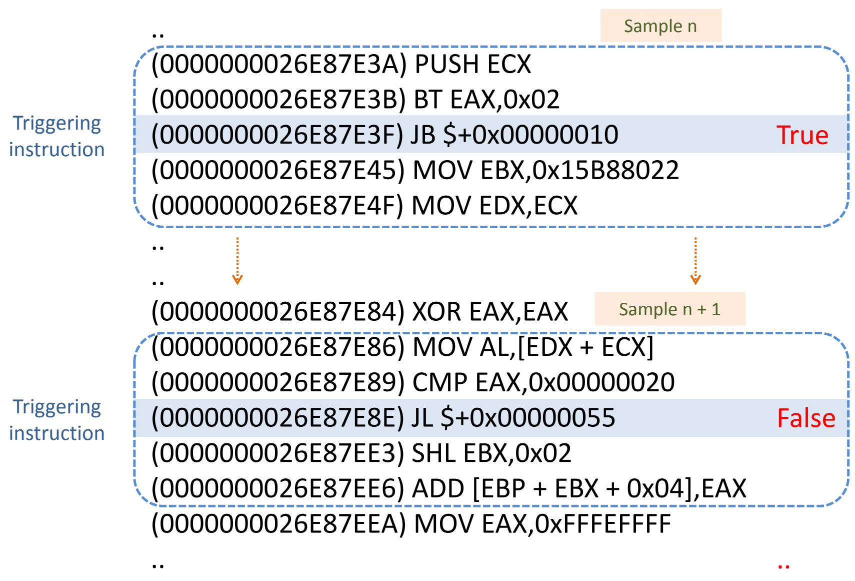
```
RTL Cycle number 13121
P000: 0000000116: Ex: 0000:F000:000000000000C1C3 (00000000FFFFC1C3) OP: MOV EAX,EDI
P000:        -- Mode: Rl16
P000:        -- Opcode Bytes: 66 89 F8
P000:        -- prefix[66] opcode1[89] modrm[f8]
P000:        -- RAX = 00000000000C0000/000000008000C000
P000:        -- RIP = 000000000000C1C3/000000000000C1C6
P000:        -- Written: RAX RIP

RTL Cycle number 13122
P000: 0000000117: Ex:0000:F000:000000000000C1C6 (00000000FFFFC1C6) OP: ADD EAX,0x01000218
P000:        -- Mode: Rl16
P000:        -- Opcode Bytes: 66 05 18 02 00 01
P000:        -- prefix[66] opcode1[05] imm1[18 02 00 01]
P000:        -- RAX = 000000008000C000/000000008100C218
P000:        -- RFLAGS = 00000006/00000086
P000:        -- RIP = 000000000000C1C6/000000000000C1CC
P000:        -- Written: RAX RFLAGS RIP
```

Figure 4.4: RTL simulation trace showing two instruction retirements

test, Task Register (TR) selector, Code Segment (CS) selector, effective address, linear address and the instruction itself. In this case, the 116th instruction(OP: MOV EAX, EDI) is executed on Core 0. It was retired on the 13121th cycle. Table walks, code reads and memory reads are some other examples of processor operations. In Fig. 4.4, there is only one keyed line each in the two blocks. This is followed by a summary of the instruction. The first three lines indicate the user mode, opcode bytes and instruction bytes description. The rest of summary has all the register value changes and a final line that lists out all the registers which were written to.

Our goal here is to find the relationship between an event and the specific instruction/instruction sequence responsible for triggering it. Because we treat an event as a function of the state and the instruction triggering it, each data sample should include information about the preceding instructions in addition to itself. It allows us to uncover the correct sequence of instructions that can act as a trigger. Since instruction execution is out of order, microarchitectural events can also be influenced by instructions following a trigger. Even though they appear after a triggering instruction, they can be executed before the trigger. Fig. 4.5 illustrates how a dataset is created from an instruction sequence.



```
..
(0000000026E87E3A) PUSH ECX
(0000000026E87E3B) BT EAX,0x02
(0000000026E87E3F) JB $+0x00000010
(0000000026E87E45) MOV EBX,0x15B88022
(0000000026E87E4F) MOV EDX,ECX
..
..
(0000000026E87E84) XOR EAX,EAX
(0000000026E87E86) MOV AL,[EDX + ECX]
(0000000026E87E89) CMP EAX,0x00000020
(0000000026E87E8E) JL $+0x00000055
(0000000026E87EE3) SHL EBX,0x02
(0000000026E87EE6) ADD [EBP + EBX + 0x04],EAX
(0000000026E87EEA) MOV EAX,0xFFFEFFFF
..
```

Figure 4.5: Use of a moving window of size 5, to generate a dataset for modeling branch mispredictions from a simulation trace. Annotation values are indicated in red.

To generate a data sample, we use a moving window that slides over the instruction sequence beginning with the first instruction. The size of the moving window is empirically set to an odd integer value $w = 2p + 1$ depending on the event being modeled. In the illustration, a window size of $w = 5$ has been used. A wider window takes the effect of more instructions into picture when describing an event. A narrow window localizes the effect. The first and the last $2p$ instructions in the test are ignored as they do not constitute a full window. If necessary, they can be included by using NOP instructions as padding.

In the example described by Fig. 4.5, features are extracted from a window only if the p+1th (triggering) instruction is predicted by the branch prediction unit (BPU). This includes all conditional and unconditional branches, and jumps and calls within the current code segment (near JMP and near CALL). Control transfer instructions such as far JMP, far CALL and RET instructions are ignored as they are not predicted. All other instructions that do not modify program control flow are also ignored. Thus, features are extracted from instructions specific to the target behavior. In cases where the target behavior is not instruction specific, feature extraction is performed for all instances of the moving window.

Features collected from each instance of the moving window is used to create a single data sample. For each instruction within the window, features that are relevant to branch prediction are extracted. We use domain knowledge to extract the right features. For example, the type of a branch instruction is important (conditional or unconditional) but the user mode at the time of execution is irrelevant. Other features relevant to branch prediction are:

- Branch location
- Branch target
- Branch outcome - Taken or not taken
- Opcode of the branching instruction

Each moving window instance is also annotated with a binary value indicating whether or not the modeled event was successfully excited by the triggering instruction, which is the $(p+1)$th instruction. In our example, a label '1' (boolean True) indicates that the instruction was mispredicted. And a label '0' (boolean False) implies that an instruction was correctly predicted by the branch predictor. These annotation values are generated by tapping into the appropriate RTL signals. Whenever the target event is triggered, the next retired instruction is annotated with a value of '1'. For example, if two branch instructions retire on the 200th and 205th cycles respectively, and a branch is mispredicted on the 203rd cycle, the second instruction is annotated with a value of '1'.

4.4.2 Generating a dataset

Data gathered from a trace is in the form of attributes. Table. 4.4 lists out the attributes used for creating the branch misprediction dataset. *Opcode*, *Op1type* and *Op2type* represent the opcode and the operand type of the first and second instruction operand, respectively. Operands can be of type immediate, register or displacement. $IsBrnTaken$ is a binary value that is set to 1 if a branch is taken and 0 otherwise. Since there can be non-branching instructions within a window, a large integer value, denoted by the constant NA, is used. This distinguishes a branch not being taken from the value not being applicable. For example, if a window has four *mov* instructions, $IsBranchTaken$ is assigned the value NA in all four instances. For the trigger instruction, the appropriate boolean value is used.

Branch location is used to track the history of each branch. Branch history includes each branch's visit count, the branch outcome (taken or not taken) on each of the last two visits (if any) and the branch target on the last visit. This is used to generate attributes $IsBrnTakenPrev1$, $IsBrnTakenPrev2$ and *BranchType*.

$IsBrnTakenPrev1$ ($IsBrnTakenPrev2$) is set to 1 if a branch was taken on its (second to) last visit. *IsExcp* indicates if a branch was taken due to an exception. *BranchType* can take the following nominal values:

Table 4.4: Illustration of attributes collected from a simulation trace

Attribute	**Value Type**
Opcode	Nominal
Op1type	Nominal
Op2type	Nominal
IsBrnTaken	Ternary
IsBrnTakenPrev1	Ternary
IsBrnTakenPrev2	Ternary
BranchType	Nominal
IsExcp	Binary

- *never_taken:* Never taken
- *taken_once:* Taken exactly once
- *static:* Always taken and fixed branch target
- *dynamic:* Variable branch target
- *not_AT:* Not always taken

Once generated, the attributes are arranged in the same order as the instructions that they are derived from, to form a data sample as shown in Fig. 4.6.

In this case, since the window size is 5, each data sample consists of processor state information relevant to branch prediction corresponding to the five instructions within the moving window. If k attributes are generated for each instruction within the moving window, for $w = 5$ each data sample has $5k$ attributes. Each

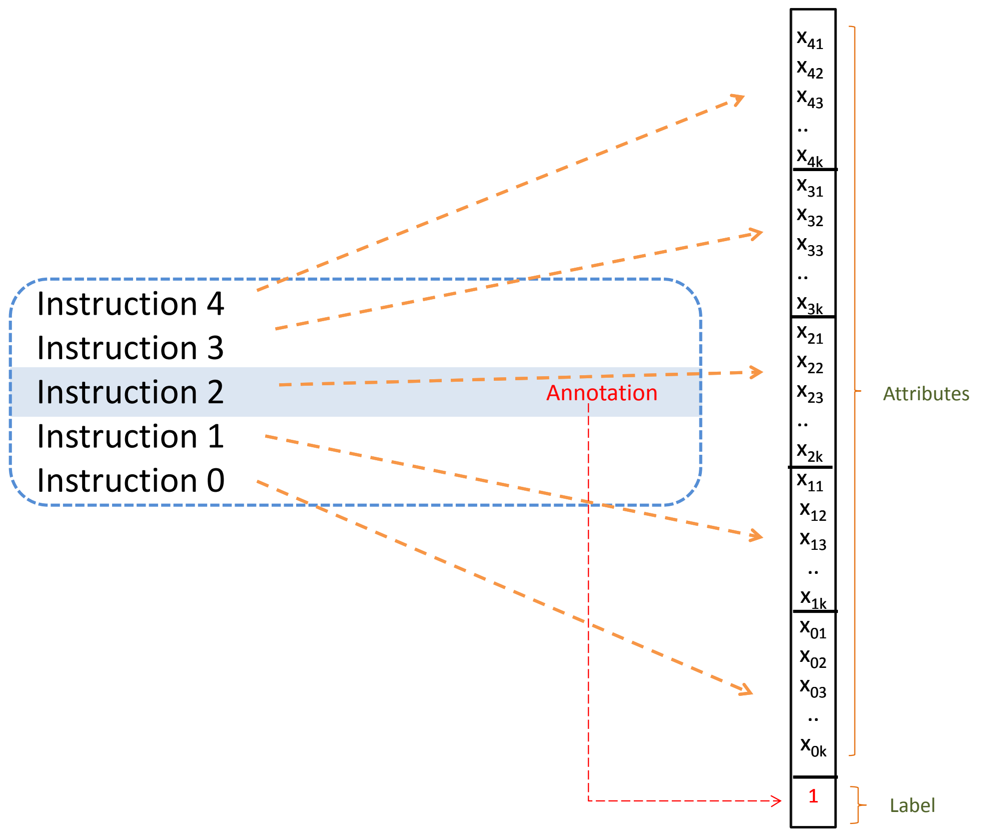

Instruction 4
Instruction 3
Instruction 2
Instruction 1
Instruction 0
Annotation
x_{41} x_{42} x_{43} .. x_{4k}
x_{31} x_{32} x_{33} .. x_{3k}
x_{21} x_{22} x_{23} .. x_{2k}
x_{11} x_{12} x_{13} .. x_{1k}
x_{01} x_{02} x_{03} .. x_{0k}
1
Attributes
Label

Figure 4.6: Generating a data sample to model branch mispredictions using a moving window of size 5.

attribute is also given a suffix between 0 (newest instruction) to $w - 1$ (oldest instruction), to indicate the instruction that it was derived from. RTL annotation value represents the label or class of the data sample. In this example, each dataset has as many samples as there are predicted branch instructions in a test. Samples can be combined from multiple tests, as long as no changes are made to the configuration of the branch predictor.

4.4.3 Rule extraction

Depending on the dataset, a number of machine learning algorithms can be used to extract knowledge. We will be using decision-tree based rule learning as an example.

Decision-tree based rule learning

The construction of decision tree classifiers does not require any domain knowledge or sophisticated parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. Their representation of acquired knowledge in tree form is intuitive and easy to comprehend. The learning and classification steps of decision tree induction are simple and fast. Decision trees can easily be converted to classification rules. A decision-tree is a flowchart-like tree structure, where each internal node (non-leaf node) denotes a test on an attribute, each branch represents an outcome of the test and each leaf node (or terminal node) holds a class label. The topmost node in a tree is referred to as the *root node*.

In 1984, L. Breiman, J. Friedman, R. Olshen, and C. Ston) published the book Classification and Regression Trees (CART), which described the generation of binary decision trees[12]. The ID3 (Iterative Dichotomiser 3) decision tree algorithm was proposed in 1986 by J. Ross Quinlan, a researcher in machine learning[37]. He then went on to present C4.5 (a successor of ID3), which became a benchmark to which newer supervised learning algorithms are often compared[38]. ID3 and CART follow a similar approach for learning decision trees from training tuples. They adopt a greedy (i.e. non-backtracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow such a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built[24].

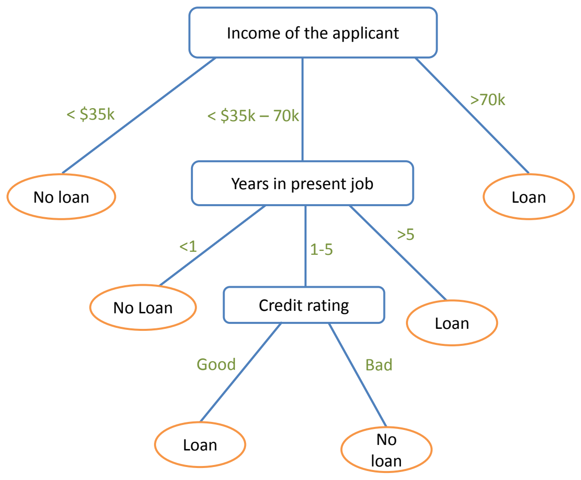


Figure 4.7: A simple decision-tree to represent a loan approval process.

A simplified decision-tree is shown in Fig. 4.7. It indicates the loan approval process of a fictitious institution. The leaf-nodes are represented by ellipses and they represent the decision of the loan approval process. The internal nodes are rectangles that indicate factors that influence the decision. To use the decision tree, each applicant is first represented as a vector. The attributes of the vector describe the loan applicant. The decision tree is traversed starting from the root node. By tracing the decision-tree from the root to a leaf node based on the applicant's attributes, a decision is arrived at.

In Fig. 4.7, the income of the applicant is used as the first splitting attribute. Applicants with an income of under \$35,000 are considered ineligible for a loan, while those with an income of over \$70,000 are eligible for a loan. Applicants who fall between these two categories are subjected to further tests to evaluate their eligibility. Those who have been on their current job for over 5 years can be approved while those under a year are not. The rest of the applicants are eligible only if they have a good credit history.

Constructing a decision-tree

Consider a dataset D. Vectors used for creating a decision-tree are referred to as training vectors or training tuples. Each vector d_i consists of features (attributes) $x_1, x_2, x_3, \ldots, x_t$ and an associated label y_i that indicates the class of the tuple. The training dataset is recursively partitioned using the most suitable attribute. An internal node N is created denoting the attribute used to partition the dataset in j subsets. Each of the outgoing branches represents the j different outcomes of the splitting attribute. If a resulting partition has samples belonging to only one class, it is marked as a leaf node and marked with the class label. A partition is also marked as a leaf node if it reaches the limit set for the minimum number of samples. The class of the majority of the population is assigned to a leaf node as its node label. This process is repeated until either all samples have been classified into leaf nodes, or until all attributes have been used. In case of the latter, the class of the majority population is assigned to all remaining nodes.

If an attribute is discrete-valued, the outcome of a test at an internal node N corresponds directly to its known values. So a branch is created for each of its known j values. For continuous valued attributes splitting points must be determined to split the attributes into discrete 'buckets'. The entropy minimization heuristic developed by Fayyad, U. and Irani, K.[19] can be employed for the partitioning such that a small range of feature values with rare occurrence is considered important and identified as a separate bin. A large range of feature values commonly-appearing in many samples are considered less important and grouped into the same bin.

Heuristic methods that best separate a given data partition into individual classes are used for selecting the best splitting attribute. If we were to split dataset D into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (i.e., all of the tuples that fall into a given partition would belong to the same class). Conceptually, the best splitting criterion is the one that most closely results in such a scenario. Attribute selection methods are also known as *splitting rules* because they determine how the dataset at a given node is to be split. The rule provides a ranking for each attribute describing the given training tuples. The highest ranked value is used as the splitting attribute. Some of the most commonly used metrics are information gain, gain ratio and gini index.

Consider the following decision-tree. This tree was created using 231 tuples to identify branches that are mispredicted. Samples labeled *true* indicate misprediction and samples marked $false$ are not mispredicted by the branch predictor.

```
BranchType = dynamic
|    |
|    IsBrnTakenPrev1 = true
|    |    |
|    |    IsBrnTaken = true: false {false=2, true=1}   => PREDICTED    ... 1
|    |    IsBrnTaken = false: true {false=0, true=2}   => MISPREDICTED .. 2
|    IsBrnTakenPrev1 = false: false {false=2, true=0} => PREDICTED    ... 3
BranchType = never_taken: false {false=120, true=0}  => PREDICTED    ... 4
BranchType = static: false {false=178, true=4}       => PREDICTED    ... 5
BranchType = taken_once: true {false=0, true=38}     => MISPREDICTED .. 6
```

Six rules can be extracted from the decision-tree, corresponding to each of the six leaf-level nodes. The node labels have been annotated on the right-hand side for convenience. An interpretation of these rules, in the order in which they appear, is as follows:

Rule 1: Dynamic branch taken on the previous and current visit is NOT mispredicted

Rule 2: Dynamic branch taken on the previous visit, but not taken on the current visit is mispredicted

Rule 3: Dynamic branch not taken on the previous visit is NOT mispredicted

Rule 4: If never taken, branches are NOT mispredicted

Rule 5: Static branches are NOT mispredicted

Rule 6: Newly discovered branches (taken only once) are mispredicted

5 samples have been wrongly classified by this decision tree. This includes one true sample under Rule 1 and four true samples under Rule 5. This is the training error in the tree generation process. Similar errors can be made while using the tree for classifying tuples of unknown class. This is the testing error.

4.5 Experimental Results

In this section, using four examples, we demonstrate how machine learning can be used to realize the dependence of design behavior on architectural state

We use the term *NOT mispredicted* in place of *predicted* to make it clear that all braches are predicted. But some are predicted incorrectly.

and design stimulus. We generated two datasets from simulation traces of two different tests using a moving window of size 5. Each sample's label represents the class assignment based on the RTL value for the trigger instruction, which is the third instruction in the five-instruction window. Features of this instruction have a _2 suffix. Instructions older to it have lower suffixes (_0 and _1) and newer instructions have higher suffixes (_3 and _4). All results are based on decision-tree based learning using gain ratio as the splitting criterion for its better performance over gini_index and information gain.

The two tests, T1 and T2, contain 4620 and 6595 instructions respectively. We evaluate the accuracy of our learning algorithm using the *holdout method*[24]. As shown in Fig. 4.8 the data is partitioned into two non-overlapping sets, a training set and a validation set, by random sampling. The training set is larger than the validation set to reduce training error. We have used a 60-40 split for training and validation respectively. The training set is used to generate a decision-tree. The accuracy of the model is evaluated using the validation set. The estimate is pessimistic because only a portion of the initial data is used to derive the model.

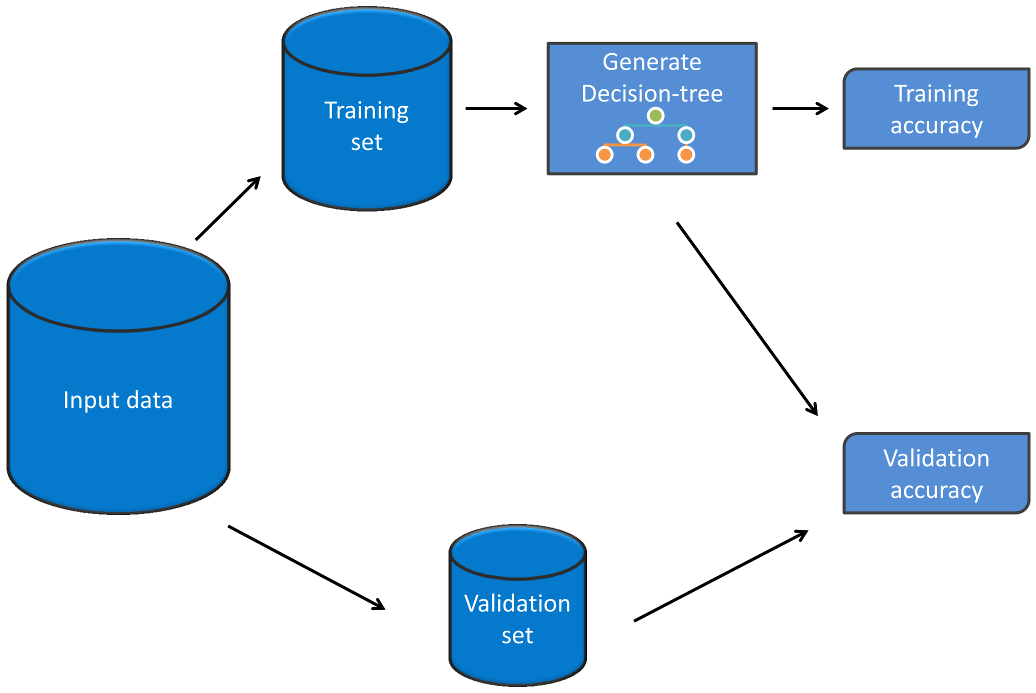


Figure 4.8: Estimating the accuracy of decision-tree based learning using the holdout method

4.5.1 Example 1: Write to CR0

Objective

CR0 is a control register. It provides operating-mode controls and some processor-feature controls. The objective of this experiment is to identify tests that write to the CR0 register. The privileged instruction `MOV CRn` is used to write the contents of a 32-bit or 64-bit general-purpose register to a control register.

Results

The decision tree, as shown in Fig. 4.9, is based on a single feature, `IsCR0written_2`. The training results are given in Table. 4.5. The training set of the first test, T1, contains 2770 samples, of which 6 contain the `MOV CR0` instruction. These samples are labeled as 'True' and all other samples are marked as 'False'. All 6 positive samples are classified accurately by the decision-tree. Similarly, all 8 positive samples in T2 are also classified correctly.

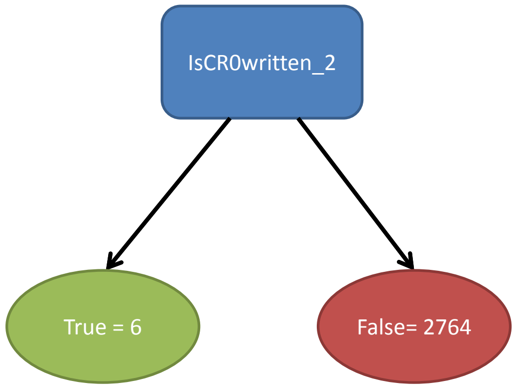


Figure 4.9: Decision tree: Write to CR0

When the generated models were applied to the corresponding validation sets, we get identically accurate results. Table. 4.6 shows the validation results. This indicates that the behavior is completely defined by the trace data. In other words, this coverpoint is pure architectural.

Table 4.5: Write to CR0: Training results

T1	**False**	**True**	**Class Precision**
Predicted False	2764	0	100%
Predicted True	0	6	100%
Class Recall	100%	100%	

T2	**False**	**True**	**Class Precision**
Predicted False	4002	0	100%
Predicted True	0	8	100%
Class Recall	100%	100%	

Table 4.6: Write to CR0: Validation results

T1	**False**	**True**	**Class Precision**
Predicted False	1840	0	100%
Predicted True	0	7	100%
Class Recall	100%	100%	

T2	**False**	**True**	**Class Precision**
Predicted False	2669	0	100%
Predicted True	0	5	100%
Class Recall	100%	100%	

4.5.2 Example 2: TLB Flush

Objective

TLBs are on-die caches that hold the most-recently used virtual-to-physical address translations. Each memory reference (instruction and data) is looked-up in the TLB. If the translation is present in the TLB, it is immediately provided to the processor, thus avoiding external memory references for accessing page tables. Depending on the implementation, separate TLBs may be implemented for data and code.

TLBs can be flushed by hardware or by software. The CR3 control register points to the base address of the highest-level page-translation table. The processor invalidates the TLB whenever CR3 is loaded either explicitly or implicitly. Frequently used or critical pages are therefore stored as global pages. Entries marked as global are retained across CR3 context switches. This is an implicit TLB invalidation. TLB entries marked as global may or may not be cleared, depending on the type of TLB invalidation.

Software initiated TLB invalidations are referred to as explicit invalidations. For example, the INVLPG instruction invalidates the TLB entry that would be used for the 1-byte memory operand. This instruction invalidates the page, regardless of whether it is marked as global or not. For more details, refer to the AMD64 Architecture ProgrammerâĂŹs Manual, Vol. 2[4]. All TLB invalidation conditions are detected by the ISA simulator.

Results

The decision tree, shown in Fig. 4.10, features a decision-tree based on only a single variable `IsTlbInvl_2`. This variable indicates a TLB flush in the ISA simulation. Results indicate that, for test T1, all 11 instances of TLB flush in the RTL were correctly identified. However, there was one instance of a flush in the ISA simulation that did not occur in the RTL. A similar observation can be made in the training model of the second test. 8 RTL flushes were correctly identified and one flush did not occur as expected.

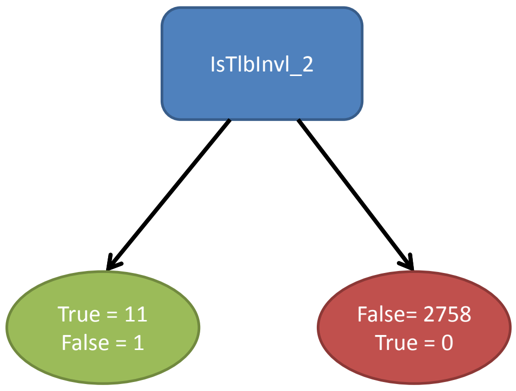


Figure 4.10: Decision tree: TLB flush

This discrepancy arises due to an RTL optimization. Writes to control register CR4 requires a TLB flush by hardware as this modifies properties of the page table. However, in both instances, the hardware does not perform a TLB flush because paging is not enabled. Though required by definition, this flush is unnecessary since the TLB is only used when paging is used.

All 4 RTL flushes in the validation set of T1 are correctly predicted. An RTL optimization, where the RTL avoids doing an RTL flush for a change made to the IC configuration, shows up as a false positive. This example is a good

Table 4.7: TLB flush: Training results

T1	**False**	**True**	**Class Precision**
Predicted False	2758	0	100%
Predicted True	1	11	91.67%
Class Recall	99.96%	100%	

T2	**False**	**True**	**Class Precision**
Predicted False	4001	0	100%
Predicted True	1	8	88.89%
Class Recall	99.98%	100%	

demonstration of how the RTL can do away with some architectural stipulations for the sake of performance, as long as it does not alter software behavior. This kind of behavioral modeling can be used to identify such differences.

Table 4.8: TLB flush: Validation results

T1	**False**	**True**	**Class Precision**
Predicted False	1842	0	100%
Predicted True	1	4	80%
Class Recall	99.95%	100%	

T2	**False**	**True**	**Class Precision**
Predicted False	2666	0	100%
Predicted True	0	8	100%
Class Recall	100%	100%	

4.5.3 Example 3: Branch misprediction

Objective

The different branch types are shown in Fig. 4.11. Undiscovered branches are not tracked by the branch predictor. All branches are categorized as *static* when discovered. As long as they are taken on each subsequent iteration and their target remains unchanged, they are classified as static. Static branches are always predicted as taken. The outcome of some branches, such as conditional branches, can change. Branches which use register values to denote their target location can have differing target addresses. Heuristic techniques are used to predict the behavior of non-static branches. The branch predictor uses a neural network based algorithm to predict the behavior of such branches. A branch whose target changes is classified as *dynamic*. Branches that are not always taken are classified as *not_AT* (not always taken). The BPU has to be trained before it can start predicting the outcome and target of non-static branches accurately. As a result, when a branch converts from static to dynamic/not_AT it is not predicted correctly for the first few iterations. Details of the interaction between the BPU and the execution unit are given in Sec. 5.2.1.

In this experiment, we extract all branches that are predicted by the BPU. Our objective is to identify mispredicted branches. Misprediction can be because

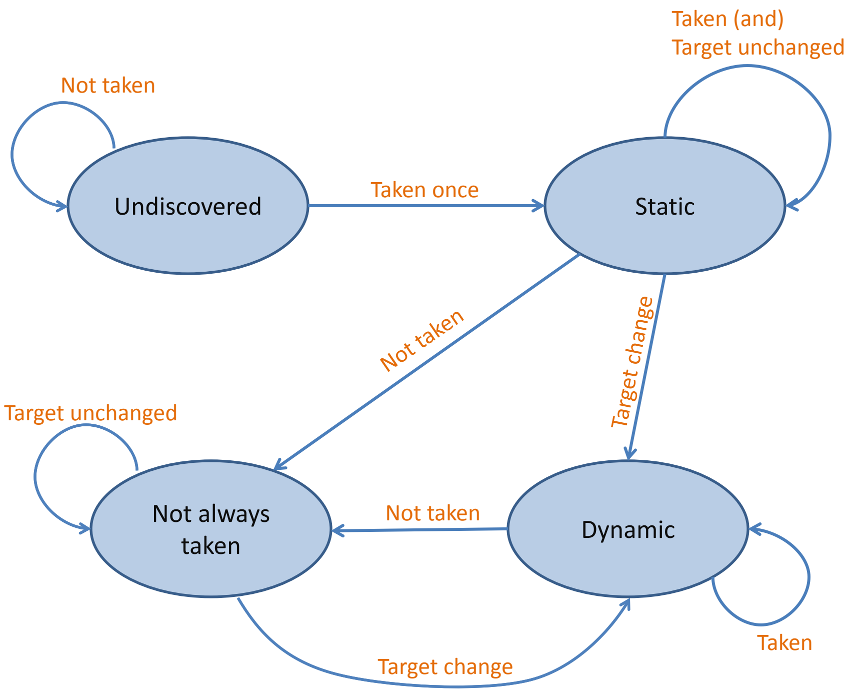


Figure 4.11: State machine representing branch types

of an incorrectly predicted branch outcome (taken or not taken) or an incorrectly predicted branch target. Unpredicted branches such as FAR jumps and FAR calls are excluded from the dataset. Samples annotated as 'true' are mispredicted and samples annotated as 'false' are predicted accurately by the BPU.

Results

The decision tree shown in Fig. 4.12 closely resembles our understanding of the BPU. Ignoring the classification errors and exceptions, the rules generated by the decision-tree can be summarized as follows::

1. Branches that are taken for the first time (discovered) are mispredicted
2. Undiscovered branches (never taken) are NOT mispredicted
3. Static branches are NOT mispredicted
4. Branches which are not always taken are:
    (a) NOT Mispredicted if they are currently taken
    (b) Mispredicted if they are currently not taken

We cannot say anything conclusively about dynamic branches since there are not enough samples in the decision-tree. Dynamic branches are predicted accurately once the BPU is trained. This is evident from Table. 4.9. It shows the 5 instances of dynamic branches in test T1. Of the three instances that are included in the training set, two are correctly predicted. Both these branches converted from static to dynamic branches on their 18th visit. By the 34th visit, the BPU was trained. Between these visits (19 through 33), both branches were of type not_AT. The two instances corresponding to the 18th visit are included in the validation set. The decision tree categorizes dynamic branches as predicted branches, by majority vote (2 false out of 3). So the two samples in the validation set will be wrongly classified as predicted.

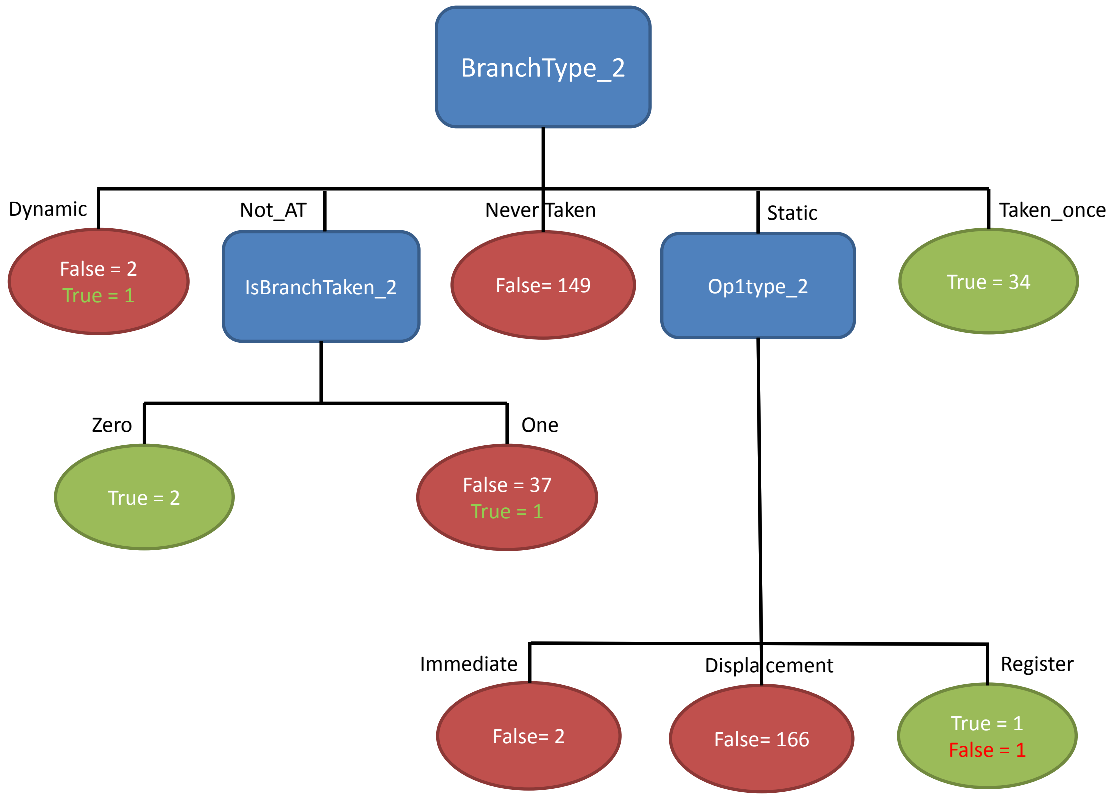


Figure 4.12: Decision tree: Branch misprediction (Based on test T1)

Branches of type not_AT behave similar to dynamic branches and require BPU training. There is one misclassified sample of this type. The decision tree, however, does not capture the real algorithm used to predict non-static branches. One of the reasons for this is that the amount of data provided for training is substantially low. One static branch has been misclassified. Contrary to expectation, this branch was mispredicted. This was caused by the branch information not being recorded on its first visit because of an internal queue overflow.

Table 4.9: Dynamic branches in T1

Branch location	Set	Visit count	Outcome
0xffffff7d142909e3	Training	2	Mispredicted
0x7368b520b107	Training	34	Predicted
0x7368b520b087	Training	34	Predicted
0x7368b520b107	Validation	18	Mispredicted
0x7368b520b087	Validation	18	Mispredicted

The five false negatives in the T2 training set include 4 static branches and one non_AT branch. The former was caused due to two reasons - loss of branch marker information in the L2 and queue overflows when recording the branch information at the time of discovery. The latter is an untrained branch prediction. These errors perpetuated as 5 validation set classification errors.

Table 4.10: Branch misprediction: Training results

T1	**False**	**True**	**Class Precision**
Predicted False	356	2	99.44%
Predicted True	1	37	97.37%
Class Recall	99.72%	94.87%	

T2	**False**	**True**	**Class Precision**
Predicted False	302	5	98.37%
Predicted True	0	37	100%
Class Recall	100%	88.89%	

Table 4.11: Branch misprediction: Validation results

T1	**False**	**True**	**Class Precision**
Predicted False	229	4	98.28%
Predicted True	0	31	100%
Class Recall	100%	88.57%	

T2	**False**	**True**	**Class Precision**
Predicted False	183	5	97.34%
Predicted True	0	43	100%
Class Recall	100%	89.58%	

4.5.4 Example 4: IC Fetch

Objective

The instruction cache (IC) is maintained such that instructions can be filled into the pipeline with minimum latency. Since the latency of the L2 cache is roughly an order of magnitude higher than than of the L1 cache, instructions are fetched preemptively. The branch predictor is tightly coupled with the IC and it speculatively fetches cachelines from the L2 whenever required. On each request, one cacheline worth of data is filled into the IC. In our current processor, the cacheline is 64 bytes wide. All fetches are 64-byte aligned, i.e. for any request, data is fetched beginning from the nearest 64-byte address boundary that contains the instruction. The ISA simulator indicates whenever a 32-byte boundary is crossed, with a *code-read*. This can be used as an indicator for IC fetches in the design.

In this example, we will extract the IC-fetch behavior based on an ISA simulation trace. An IC fetch request is really a cache miss function. A fetch request is issued to the L2 cache whenever an instruction required by the execution pipeline is not present in the IC. For cacheable memories, this happens only the first time. As long as the entry is still valid, it is not re-fetched on subsequent iterations. Uncacheable memories are requested on each iteration. If an instruction fetch request is recorded in RTL between the retirement times of instructions n and $n + 1$, the latter is assigned a 'true' class label. This indicates that instruction $n + 1$ is most likely to have triggered the fetch.

Results

Table 4.12: IC fetch: Training results for test T1 (first attempt)

T1	**False**	**True**	**Class Precision**
Predicted False	2669	99	96.42%
Predicted True	0	2	100%
Class Recall	100%	1.98%	

The results shown in Table. 4.12 illustrate that no meaningful rules can be extracted from the current dataset. Only 2 of the 101 fetches are identified. The rest 99 are false negatives. Clearly, the 'code-read' indicating 32-byte boundaries cannot be used to accurately define a cache-miss event. Lowering the threshold used for by decision-tree algorithm for variable selection produces better results. But this results in overfitting and the extracted rules are not meaningful. Since the data samples are restricted to a window of just 5 instructions, the rules cannot account for a previous fetch. The cache miss function is a function of an instruction's fetch history. A revisited instruction does not trigger a fetch in RTL if it is cacheable. However, since there is no feature in the dataset that reflects if an instruction has been fetched before.

This can be resolved by tracking instruction fetch history and including a binary value indicating whether or not an address has been fetched in the past, whenever a 'code-read' is detected. Since instructions are fetched one cacheline at a time, the cacheline width has to be taken into account to generate this feature. One method is to take a micro-architecture agnostic approach and use the trace to deduce what the cacheline size is. We can assume different cacheline sizes and see what best describes the observed behavior.

Fig. 4.13 shows effect of the newly generated feature `WasFetched` on the learning process for four different cacheline sizes assumptions: 16 bytes, 32 bytes, 64 bytes and 128 bytes. For an assumed cacheline width of 16 bytes, if an instruction within a 16 byte-aligned boundary of the current instruction has been visited before, the value of attribute `WasFetched`is set to 1. If not, it is set to 0 and so

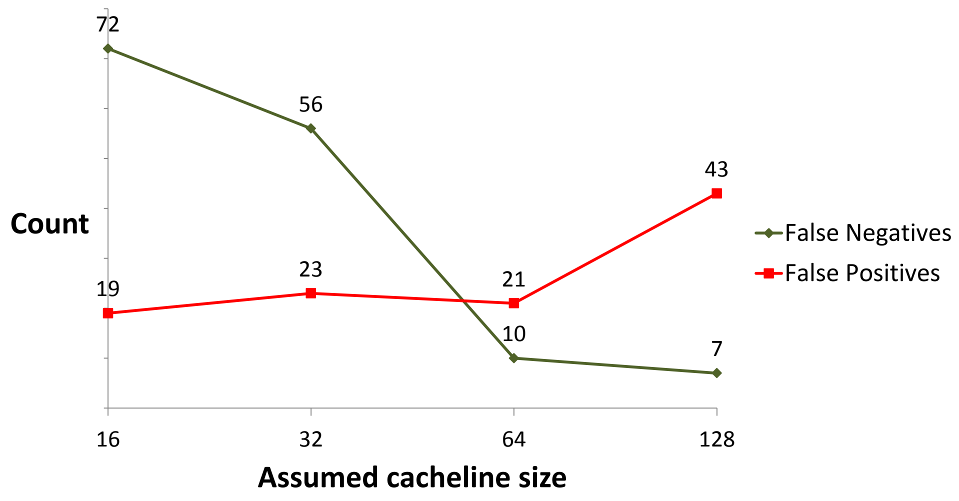


Figure 4.13: Training error for test T1 using four different cacheline sizes to track code fetch hisotry

on. Starting with 99 false negatives, when this attribute is not included in the dataset, we can see that the false negatives drop monotonically as the cacheline size is increased. The change is more dramatic for the first two step, from 16 to 32 (reduction of 16) and then from 32 to 64 (reduction of 46). But there is a difference of only 3 false negatives from 64 to 128. However, there is a sharp increase in the number of false positives, from 21 to 43, for this step. Since the cacheline size of 64 minimizes the overall training error, we can conclude that the design utilizes a cacheline of width 64 bytes (For aliasing related problem, see conclusion).

The decision tree generated from this modified dataset is shown in Fig. 4.14. We can see that the newly added variable `WasFetched_2` best partitions the data.

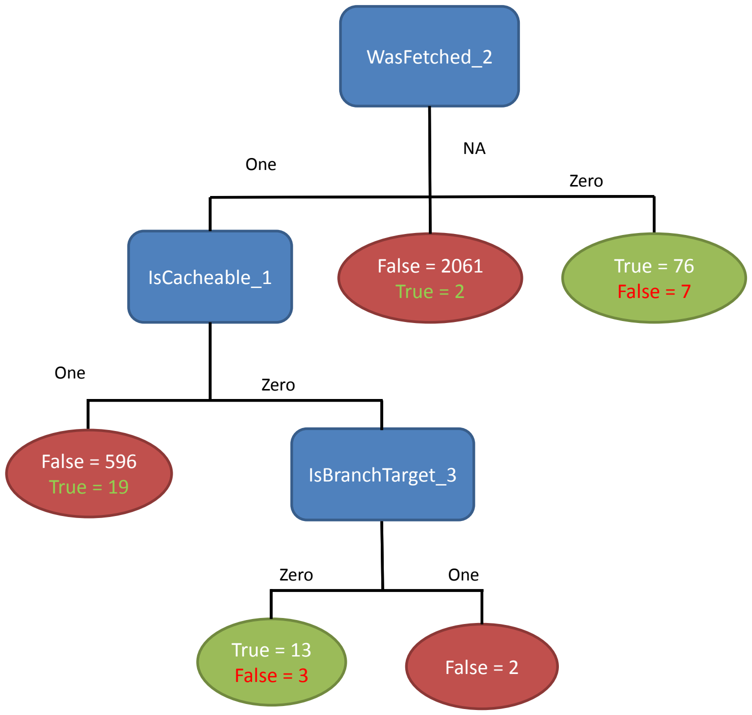


Figure 4.14: Decision tree: IC fetch

Addresses not already fetched are *most likely* to be fetched. While those addresses that are already fetched are *most likely* to be not fetched. If there is no 'code-fetch', an IC fetch is *unlikely*. As we can see from the decision tree, these rules have their exceptions.

From the results shown in Table. 4.13, we can see that the percentage of correctly identified IC fetches has gone up to 79%. There are 21 instances of IC fetches that are not identified by the model. For example, 19 cacheable addresses were re-fetched. This is caused by:

- Re-accessing a memory of memtype uncacheable

- Change of memtype from an uncacheable memtype to a cacheable memtype
- Cache eviction
- Cache invalidation

Uncacheable memory is not retained in the instruction cache. So fetch requests are issued every time they are accessed.

There are two instances that are not identified by a code-fetch. This is because of the presence of serializing instructions. Serializing instructions force the processor to retire the serializing instruction and all previous instructions before the next instruction is fetched. Instructions that change the processor configuration such as MOV CRn and WRMSR are serializing[4]. Certain configuration changes, such as change in memtypes forces the processor to re-fetch code or data to maintain coherency.

Table 4.13: IC fetch: Training results for test T1 (second attempt)

T1	**False**	**True**	**Class Precision**
Predicted False	2659	21	99.22%
Predicted True	10	80	88.89%
Class Recall	99.63%	79.21%	

The decision-tree also shows 7 instructions not triggering a fetch even though they have not been previously fetched. While generating the dataset, to make the associations between fetch requests in the RTL and triggers in the ISA simulation trace, a queue of 10 requests was used. So if a request came much earlier than 10 requests, it is very likely that the instruction itself did not trigger the fetch. The fetch request was issued due to some other speculative activity. So these 7 instructions did not directly trigger a fetch because the code was fetched preemptively.

The sub-tree that uses `IsBranchTarget_3` as the splitting variable, shows a good degree of correlation with the data with only 3 false positive classifications. This is non-causal. In other words, though these features correlate with the data, they are not related to the behavior in any meaningful way. We can prune the decision tree and replace the sub-tree with a single leaf node labeled as 'True' to correctly represent IC fetch behavior. The will result in a single leaf with 13 'True' samples and 5 false positives. Again, these five false positives are caused by fetch requests being issued much before the instruction execution.

These misclassifications can be rectified by adding additional variables to the dataset. For example, we can track each linear address memtype and indicate if its memtype has changed between two visits. Each newly added attributes that describes the cache and memory architecture improves the training results.

The results for test T2, shown in Table. 4.15, are comparable to that of test T1.

Table 4.14: IC fetch: Validation results for test T1

T1	**False**	**True**	**Class Precision**
Predicted False	1770	14	99.22%
Predicted True	2	61	96.83%
Class Recall	99.89%	81.33%	

Table 4.15: IC fetch: Training (top) and Validation (below) results for test T2

T2	**False**	**True**	**Class Precision**
Predicted False	3776	27	99.29%
Predicted True	19	188	90.82%
Class Recall	99.50%	87.44%	

T2	**False**	**True**	**Class Precision**
Predicted False	2510	21	99.17%
Predicted True	15	128	89.51%
Class Recall	99.41%	85.91%	

4.5.5 Conclusion

Our ability to extract testing knowledge depends on the type and complexity of the function we are learning. As seen in the first example, architecturally defined behavior are easy to learn. As the behavior becomes more implementation dependent, the complexity and size of data required to extract the mapping function increases.

Consider the IC fetch example. Here, we are trying to learn the cache miss function. Every time a cache-miss happens, a fetch request is sent to the L2 cache.

Learning this function from the simulation trace is in essence tracing the cache behavior; more specifically the following:

- Cache architecture: Width and number of entries
- Cache policies: Associativity and replacement policy

As was demonstrated, these parameters can be determined from simulation data to accurately predict when a cache miss will occur. Suppose there are w and e different values that the width and number of IC entries can take, respectively. Also, let us suppose that there are a different types of cache associativities and r different replacement policies. This gives us a total of w x e x a x r hypotheses. Table. 4.16 lists out some of the most common cache configurations. This gives us a total of 2 x 4 x 9 x 7 - 7 * 4 * 2 + 4 * 2 = 456 combinations (The adjustment is because direct mapped caches do not need a replacement policy). We have to generate sufficient data to rule out all but one hypothesis to learn the cache-miss function of a given design.

While generating data we also have to account for aliasing. Consider accessing instructions from 4 different cachelines sequentially in the following order: $CL1 \longrightarrow CL2 \longrightarrow CL3 \longrightarrow CL1$. Suppose all these three cachelines have the same index and the cache was cleared prior to running this sequence. This would cause an address collision at the index corresponding to these cachelines. If a

Table 4.16: Table enumerating some of the possible IC configurations

Parameter	Possible values	Total
Width	32 64	2
Entires	32 64 128 256	4
Associativity	Direct mapped 2-way set associative 3-way set associative 4-way set associative 5-way set associative 6-way set associative 7-way set associative 8-way set associative Fully associative	9
Replacement policy	Least Recently Used Most Recently Used Pseudo-LRU Random Replacement Round Robin Segmented LRU Least-Frequently Used	7

cache is 2-way set associative with LRU replacement policy, then CL3 would replace CL1 as CL2 was used last. This sequence would generate a cache-miss for all four accesses. A direct mapped cache would also behave in the exact same way. Since all three cachelines have the same index, each one would evict the previous, resulting in four consecutive cache-misses. So the amount of data required to distinguish all the different combinations is significantly high. Tests should be able to distinguish each parameter and configuration. The combinations listed in Table. 4.16 are only some of the most commonly used values. Since many more combinations are possible in theory, a generic solution to this problem is extremely tedious. Learning a function like cache-miss requires a large volume of simulation data. The dataset must also include the entire set of features that define the cache implementation.

4.6 Summary

Random test program generation has its limitations. Expressing target machine behavior in the form of user directives is not always adequate to constrain test generators. We propose to use a filter external to test generators that is independent of the test generator design. Our filter uses architectural values extracted from ISA simulation traces. Experiments show that this is inadequate to filter all RTL events. To filter complex microarchitectural behavior, we have to understand the relation between RTL events and architectural values.

We use decision-tree based rule learning to extract rules that explain the target behavior in terms of known architectural values. In cases where the target behavior is closely dependent on the design microarchitecture, rule learning is challenging. To accurately model complex properties requires large volumes of data. The dataset also has to include all the features required to fully define the targeted behavior. The complexity of the feature generation process and necessity to have large datasets makes this process impractical for industrial use.

Chapter 5

Micro-architecture model based test filtering

5.1 Introduction

Simulation based verification works on the principle that a design bug in a processor feature can be detected provided that a sufficiently large percentage of the input sequences are applied. This is used as a guiding principle by test generation software[30]. For this reason some test program generators integrate it into the stimulus generation process[2]. As seen in Chapter 4, without the use of a micro-architectural model, test behavior cannot be constrained accurately. It is impractical to extract a micro-architectural model from simulation data using existing non-parametric machine learning algorithms. Instead, we propose to use a manually constructed model using our knowledge of the design (RTL).

Since our test generators are ISA model based, we can use an external microarchitectural model for targeted test generation. The process of generating test directives based on simulation data requires manual effort. By using an external filter, we can bypass this. Secondly, it gives us better control over the selection of tests. Test inputs do not always deliver results as expected because of constraint conflicts within an exerciser. Using the rules for an external filter gives us a better understanding of how the rules regulate test behavior since it is independent of the test generation process. This also keeps the test filtering process independent of the test generator.

This methodology is illustrated in Fig. 5.1. The flow is very similar to the one shown in Fig. 4.2. ISA trace is used for filtering tests prior to RTL simulation. The only difference between the two flows is the introduction of the micro-architectural model (MAM). Simulation traces are extrapolated using a micro-architecture model to provide a closer approximation of real test behavior. The results are then compared against the target property. Only those tests that match the target behavior are simulated on RTL. The MAM used for filtering depends on the target property.

We will now use two examples to demonstrate the use of MAMs. MAMs provide enough complexity to mimic RTL behavior, but are simple enough to

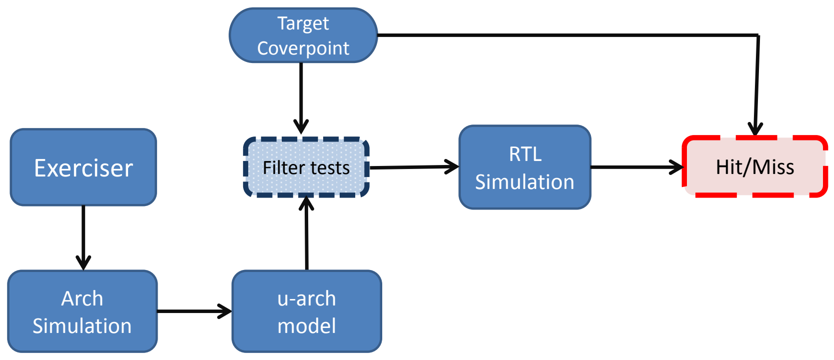


Figure 5.1: Architecure simulation based test filtering using a micro-architecure model(MAM)

create and use without any significant overhead. Given their simplicity, they are also easy to modify and reuse.

5.2 Dense branch re-fetches

5.2.1 Background

For pipelined microprocessors, the penalty of taking a branch is very high. The pipeline has to be flushed of all the 'bad path' instructions and reloaded with instructions from the branch target. With deeper pipelining, the penalty of flushing and reloading the entire pipeline has increased. The delays are higher if the target instruction does not reside in the instruction cache (IC) and has to be fetched from higher memory hierarchies. Modern processors therefore rely on speculation to boost IPC. Instead of stalling execution on encountering a branch, they execute code speculatively. To avoid stalling the pipeline due to memory access latencies, code and data are prefetched. Based on the Branch Prediction Unit's (BPU) path speculation, instructions not present in the IC are prefetched.

Branch prediction

A branch must be *discovered* for the BPU to start predicting it. The BPU discovers branches the first time that they are retired as 'taken'. In other words, to be discovered, a branch has to be taken at least once. A branch that is never taken will be ignored by the BPU. Once a branch is discovered, a *marker* is stored in a marker array to identify its location and properties. Stored branch properties include the branch type, outcome and target. For all subsequent visits, the BPU predicts the branch by the presence of its marker in the marker arrays at fetch time. Both, the direction and target of discovered branches are predicted. If a branch is predicted-taken, the Instruction Fetch (IF) unit is redirected to the predicted target of the branch. This way the pipeline does not have to wait till branch execution to know the path to execute. The BPU only predicts near branches. Far branches are not predicted.

Consider the cacheline shown in Fig. 5.2. Let us assume that this section of code has not been executed before. So no branch prediction information is

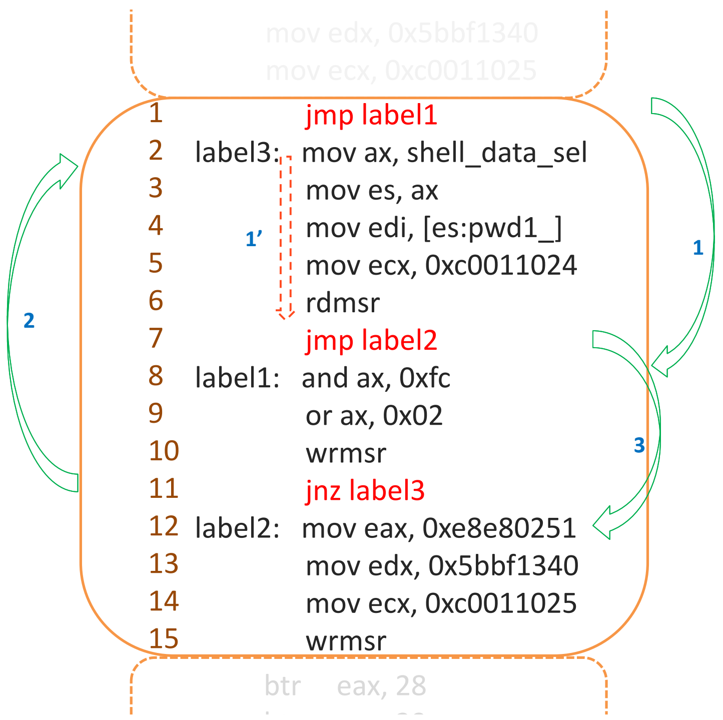
A far branch transfers control outside of the current code segment. A near branch transfers control within the current code segment.

available prior to decode. The decode unit decodes the branch instruction (`jmp label1`) and all subsequent instructions (line 2 onwards). These instructions are dispatched for execution. When the `jmp label1` instruction is executed for the very first time, the branch is discovered. The decode unit now gets redirected to the branch target `label1`, indicated in the figure by path 1. The *speculative or bad-path* is indicated by path 1'. This redirect (or pipeline flush) causes the decode unit to restart the decode process starting from `label1`. If any of the instructions have been executed out-of-order on the bad-path, their results are discarded. The BPU registers the branch location, type and target. This is called *branch discovery.*

On the following iterations, the BPU is aware of the `jmp label1` instruction. At the time of decode, it asks the decode unit to stop decoding after this instruction and resume decoding from `label1`. Thus branch prediction avoids expensive pipeline flushes once a branch is discovered.

Sparse and dense branches

The first two branches within a cacheline are referred to as *sparse* branches. These are predicted using faster and low power logic. Subsequent branches in the same cache line (after the first two sparse-marked branches) are marked in the *dense* marker arrays. These are referred to as *dense branches.* The order is



```
            mov edx, 0x5bbf1340
            mov ecx, 0xc0011025
1                   jmp label1
2   label3:   mov ax, shell_data_sel
3                   mov es, ax
4                   mov edi, [es:pwd1_]
5                   mov ecx, 0xc0011024
6                   rdmsr
7                   jmp label2
8   label1:   and ax, 0xfc
9                   or ax, 0x02
10                  wrmsr
11                  jnz label3
12  label2:   mov eax, 0xe8e80251
13                  mov edx, 0x5bbf1340
14                  mov ecx, 0xc0011025
15                  wrmsr
            btr     eax, 28
```

Figure 5.2: Illustration of a cacheline with one dense branch and two sparse branches

decided by the position of the branch and not the order of discovery. The latency of predicting dense branches is higher. The objective of this experiment is to identify tests that first evict and then re-fetch(from L2 cache to IC) at least one cacheline containing dense branches.

Fig. 5.2 shows one cacheline worth of instructions (illustration only). The first instruction is an unconditional jump to the 8th instruction. This is the first discovered branch. This branch is marked as the first branch of the cacheline in the sparse marker array. The next discovered branch in this cacheline is at line 11. Since the result of the *OR* instruction is non-zero, the conditional jump will be taken. This branch is recorded as the second sparse branch. The jump at line 7 is discovered third. This is an unconditional jump to the MOV instruction labeled as $label2$. Since this jump is located before the conditional jump at line 11, it replaces JNZ as the second sparse branch in the cache line. The conditional jump JNZ gets promoted to a dense branch. Thus, by end of executing this cacheline, the BPU would have recorded two sparse branches and one dense branch.

5.2.2 Objective

In this experiment, our intention is to generate tests that cause the re-fetch of cachelines from the L2 cache that contain dense branches. This implies that a cacheline on which dense branches are discovered has to be first evicted from the IC. Subsequently, this cacheline should be fetched at least once by the IC. This coverage property ensures that dense marker information is calculated correctly for dense branches and the markers are not destroyed upon evictions to the L2.

5.2.3 Why ISA-based filtering does not suffice

To filter tests that cause dense branch re-fetches, we should be able to distinguish the following test properties:

1. Cacheline fetch (to IC) and write-back (to L2 cache)

2. Occurrence of dense branches in a cacheline

3. Speculative code execution

The need for an MAM to select tests accurately is summarized by Table. 5.1. Cache architecture is independent of the ISA. Therefore, an ISA simulator like KOS does not model the IC structure or replacement policy. Consequently, it is unaware of cacheline fetches, evictions and BPU behavior. Given how implementation dependent this property is, the target behavior cannot be learned from the simulation trace. This was demonstrated in Sec. 4.5.4.

Table 5.1: Role of micro-architecture model in identifying dense branch re-fetches

RTL behavior	**Provided by ISA sim**	**Provided by MAM**
IC fetch	Instruction retirement Time of retirement Opcode bytes Instruction LA/PA Memory type	IC specification IC replacement policy
IC evictions	*All of the above* Cache invalidation Warm resets Unexpected redirects	IC specification IC replacement policy
Branch prediction	Branch history	Branch discovery logic Branch prediction logic

5.2.4 Using an IC and BPU model

IC fetch and evictions

Using an IC model allow us to track the state of the IC throughout the course of a test execution. For every cacheable instruction retired in the trace, the IC model is looked up. On a miss, the cacheline containing the retired instruction is added to the IC model at the appropriate index and way. On a hit, the IC model is left unchanged. For example, let us assume that the processor contains a 2-way set associative IC with 128 entries indexed by linear address bits [12:6]. Suppose each entry holds a 64-byte aligned cacheline. The IC replicates this structure. For each retired instruction, we identify the 64-byte-aligned cacheline that it belongs to. If the cacheline is already present in the cache model, we proceed to the next instruction. If not, we simulate a 'fetch' from L2 i.e the cacheline is added to the IC model. By tracking the instructions in an ISA simulation trace, we can track the IC fetch and evictions as they would happen in RTL.

```
00000000F6E4C7F        jrnp err_guest
00000000F6E4C84        btr eax, 3
00000000F6E4C88        inov cr0, eax
00000000F6E4C8B        pop eax
00000000F6E4C8C        iretd
```

In the piece of code shown above, the conditional jump located at 0xF6E4C7F belongs to a cacheline whose index (LA[12:6]) is 31h. This 64-byte cacheline extends from linear address 0xF6E4C40 to 0xF6E4C7F. At the time of retirement of the jump instruction, this cacheline will be present in the i-cache. The next cacheline, with index 32h, extends from 0xF6E4C80 to 0xF6E4CBF. This instruction will cause a prefetch of the next cacheline starting at 0xF6E4C80 (unless it is already present in the IC). This happens irrespective of the branch outcome, as the instruction itself spans into the next cacheline. If index 32h is vacant, the cacheline will be added to way0. If way0 is occupied, it will be filled in way1.

If both the ways of index 32h are already occupied, the IC model follows the way-replacement policy of the processor to ascertain which cacheline has to be evicted. For example, if either of ways is occupied by an invalidated cacheline, it is replaced. If not, if the processor follows a LRU-based replacement policy, the least recently used cacheline is replaced.

Tracking dense branches

Knowing the cache architecture allows us to calculate the cacheline boundaries. We can also track branch discoveries and calculate if a discovered branch is recorded as a sparse branch or a dense branch. Thus an IC model allows us to use ISA simulation data to calculate if a test contains dense cacheline. It also allows us to estimate if a cacheline containing one or more dense branches is evicted to L2 or refetched to IC.

Speculative IC evictions

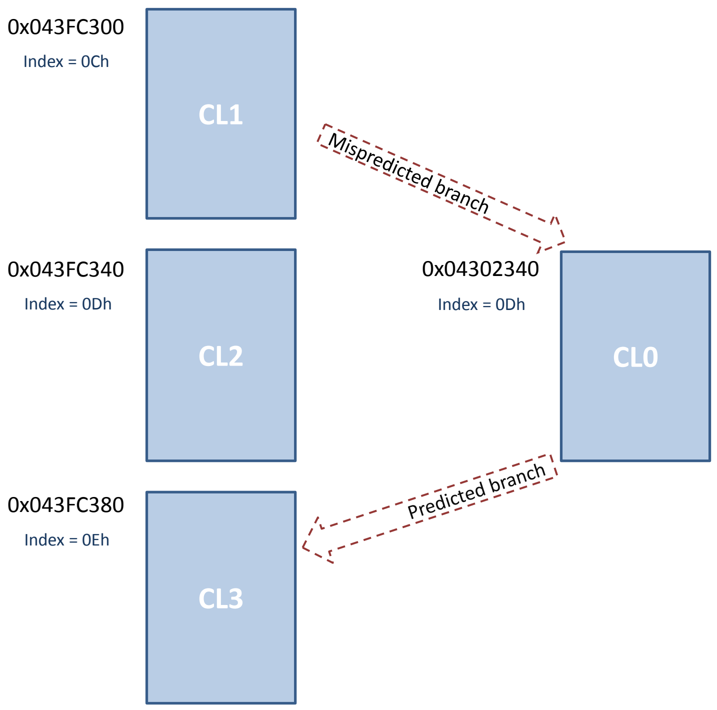


Figure 5.3: Demonstration of how cache evictions can happen due to the speculative nature of code execution

Fig. 5.3 illustrates the flow of control between three cachelines. Cacheline CL1 (starting address 0x043FC300) contains a branch to CL0 (starting address 0x04302340) and has never been taken. Because the BPU has not discovered the branch, the fetch unit does sequential prefetches of cachelines CL2 (starting at address 0x43FC340) and CL3 (starting at 0x43FC380). Let us assume that CL0 already exists in the IC (since it belongs to a lower address range). CL0 contains a branch to CL3 that is known to the BPU. Let us also assume that there is another valid cacheline CL5 (not shown) at index 0Dh of the IC, which has been used most recently recently. CL2 has an index of 0Dh and there is no vacant slot in the IC for insertion. Since both IC entries (CL0 and CL5) at index 0Dh are valid, the least recently used entry is evicted. This means that CL0 will be replaced by CL2. When the execution unit encounters the branch in CL1, it issues a redirect to the branch target. The branch target lies in CL0, but the sequential prefetch evicted CL0 from the IC. So CL0 has to be refetched from the L2 cache. The branch target for the predicted branch in CL0 is in CL3, and the cacheline CL3 exists in IC. So no additional prefetching is done. This example shows how a cacheline can be evicted due to speculative execution. Even though CL2 was never executed, it evicted CL0. If we followed the code execution without account for speculative prefetching, we would fail to see the eviction and the subsequent re-fetch of cacheline CL0. On the contrary, there would have been no such eviction if the branch in CL1 was known to the BPU.

Estimating speculative path behavior

To account for speculative evictions, in addition to updating the IC model based on code execution, we also 'prefetch code'. This is done whenever we encounter a branch instruction or redirect. Entries are added to the IC model to account for microprocessor prefetches caused by speculation. Table 5.2 lists out possible causes of program flow change. Whenever we encounter a branching in-

Table 5.2: Sources of program flow change

Expected	**Unexpected**
Static branch	Branch discovery
	Dynamic branch
	JMP_FAR/CALL_FAR
	RETs
	Interrupt
	Exception

struction or a change in program flow, we use the rules extracted in Sec. 4.5.4 to predict the next k cachelines that the RTL is likely to prefetch. A *static branch* is an example of an expected redirect. A static branch is a discovered branch whose outcome and target remain unchanged. A static branch is always taken and predicted as taken by the BPU. Since the branch behavior is known, there is no bad-path overhead. When the BPU encounters a predicted branch, instruction decode automatically restarts from the branch target. For unexpected redirects, the processor is likely to have speculated incorrectly.

To determine the speculative path, we make use of code history. We use branch history information to locate the targets of discovered branches. A recursive prefetch is done along the speculative path until the next k cachelines are determined. If there are no discovered branches on the speculative path, since the prefetch is sequential in RTL, we simply follow the code flow to track IC activity.

5.2.5 Pseudocode

The pseudocode for updating the IC model based on speculative execution is given below.

```
main ()
        ..
        ..
        CALL clr_IC

        WHILE (next instruction != NULL)
                // Speculative execution only happens if there are redirects
                IF ((CurrInstr.IsABranchInstr  or CurrInstr.IsBranchTaken) and
                                                  CurrInstr.Iscacheable ) THEN
                        CALL do_spec_load with CurrInstr
                END

                //Clear IC when required
                IF (IC invalidation is detected) THEN
                        CALL clr_IC
                END

        END
        ..
        ..
END

// Does speculative loads to the IC model
SUB do_spec_load (CurrInstr)
        SET TargetLinAd to zero
        //Look if we have a history of the current branch
        IF (CurrInstr.BrnHist != NULL) then
                //If this was a predicted branch, then use
                // its previous target if it was taken
                IF (CurrInstr.BrnTknPrev1 and CurrInstr.IsBrnPred) THEN
                        SET TargetLinAd to CurrInstr.PrevBrnTgt
                END
        ELSE
                // If it was not taken (even if it is dynamic/
                // not always taken) assume it was not taken
                // There will be no spec. execution if a
                // predicted-taken branch is not taken
                TargetLinAd = CALL calc_spec_exec_tgt with CurrLinAd
        END
         // Set spec. load bit of corresponding way if
         // spec. execution induces a fetch
         FOR  i = 1 to depth_of_speculation
                ReplWay = lookup_ic(TargetLinAd)
                // If cacheline exists in IC model, lookup_ic() returns -1
                // Otherwise, it returns the way to be replaced
                IF (ReplWay != -1) THEN
                        Mark speculative write to ReplWay at ic_index(TargetLinAd)
                END
                //Recalculate next target
                TargetLinAd = CALL calc_spec_exec_tgt with TargetLinAd
        END
END


// Checks if cacheline exists in IC or needs to be fetched from L2
// Returns -1 on cache hit. On cache miss, follows 2 step replacement
// policy to determine way to replace and returns way(0/1).
SUB lookup_ic (LinAd)
        IF (ExistsInIC(LinAD)) THEN
                Return -1
        ELSE
                 IF (! is_val(0, ic_index(LinAd)) THEN
                        Return 0
                 ELSIF (! is_val(1, ic_index(LinAd)) THEN
                        Return 1
                 ELSE
                        RETURN lru_way(ic_index(LinAd))
                END
        END
END
```

The pseudo-code of `sub calc_spec_exec_tgt` required to identify the speculative path is given in Appendix A.1. We use a parameter k because the extent of prefetching is dependent on the pipeline stage at fetch time. This value cannot be calculated without an accurate model of the pipeline. We therefore use an empirically established value. In addition to speculative and non-speculative fetches and evictions, maintain an accurate cache model requires us to detect cache invalidation conditions. For example, the INVD instruction invalidates all levels of cache. Tests can include warm resets, which also require us to invalidate the IC.

5.2.6 Results

Our experiment consists of two steps:

1. Identifying novel tests from past regressions

2. Generating new novel tests

Identifying novel tests from past regressions

To ensure that we are able to correctly identify novel tests, we first use our setup to recognize tests that have already been simulated on RTL. This lets us evaluate the effectiveness of our approach. Of the 5000 tests generated by Amex that we used to run the SVM regression, 26 tests triggered dense branch re-fetches in RTL simulations. The results are shown in Table. 5.3.

Of the 26 known tests, we were able to identify 11 tests. The identified tests are shown in rows 1 through 11 of Table. 5.3. An analysis of 2 of the 15 escapes revealed that some cachelines were being evicted by speculatively fetched code. By tracking speculative code fetches, we were able to identify a total of 13 tests. Test #3 had instances of both speculative and non-speculative IC events. Tests #12 and #14 are purely speculative in nature.

The other 13 tests escaped detection due to various reasons. One of the main reasons is testbench randomization. Every test starts the processor with its own

Table 5.3: Result of using an MAM to identify known tests with dense branch refetches

#	Preloaded	Irritator	CacheDisable Way1	1way model	2way model
1				NS	NS
2				NS	NS
3				NS	BO
4				NS	NS
5				NS	NS
6				NS	NS
7				NS	NS
8				NS	NS
9				NS	NS
10				NS	NS
11				NS	NS
12			Y	SP	SP
13			Y	NS	
14				NS	SP
15	Y				
16	Y				
17	Y				
18	Y				
19	Y				
20		Y			
21	Y			NS	
22			Y	NS	
23			Y	NS	
24			Y	NS	
25			Y	NS	
26			Y	NS	
				20	**13**

NS	=	Non-speculative
SP	=	Speculative
BO	=	Both

processor configuration. This includes configuration registers within the units (decode unit, bus unit, IC, DC etc), debug registers and performance monitors. This changes both the initial state of the processor and its behavior. For example, one of the two IC ways can be disabled. This effectively changes the IC from a 2-way set associative cache to a direct mapped cache. It also reduces the cache size to a half. Cache misses and evictions are twice as likely as a result. These changes have to be made to the IC model used to track test behavior. Otherwise, they will yield pessimistic results.

The third column of Table. 5.3 indicates if a test disables way1 of the IC in its initial setup. 7 tests were set up to run with only one way enabled. By disabling one way of our IC model, we were able to identify all 7 of these tests. This includes test #12 which has already been identified. Naturally, all of the 13 tests that we detected using the 2-way cache model test positive when we use a 1-way cache model. By accounting for speculative execution and way disabling, we were able to identify 20 of the 26 tests. Test #21 tested positive even though it did not disable way1. This is because of the added optimism that comes with using a direct-mapped IC model. By estimating test behavior using a 2-way IC model or a 1-way IC model whenever disabled by the test, $\frac{19}{26} = 73\%$ of novel tests can be identified.

6 tests escaped detection. 5 of the 6 tests were IC pre-loaded. This means that instead of starting with an empty cache, these tests pre-loaded the IC at the beginning of the test. Pre-loading improves verification coverage by reducing test length. Tests often require long instruction sequences to set up interesting behavior, before it can be triggered. Pre-loading makes it easier to trigger such events. In this case, the IC is pre-loaded with code and the marker-arrays are populated with pre-calculated marker information at the start of the test. This has the effect of having already executed some of the code. In other words, even if a branch is visited for the first time, it can be predicted correctly. This is possible

since its branch markers were a part of the pre-loaded information. Since our IC model is not pre-loaded, it fails to have the same effect. These tests can be identified by pre-loading the IC model.

Of the 6 escaped tests, one test (test #20) registered coverage because of a randomly injected error. Testbenches often introduce random errors into the memory and logic to asses the ability of the design to handle errors. This is done to simulate real-world effects of chip-level soft errors. These errors occur when the radioactive atoms in the chip's material decay and release alpha particles into the chip. Because an alpha particle contains a positive charge and kinetic energy, the particle can hit a memory cell and cause the cell to change state to a different value. The atomic reaction is so tiny that it does not damage the actual structure of the chip[35].

Generating new novel tests

As we have already seen, 26 of the 5000 simulated tests were novel. Which means $\frac{26}{5000} = 0.52\%$ of the tests were novel. We then applied our test filter to 5000 unsimulated tests. The results of this test filtering is shown in Table. 5.4. We identified 33 tests as most likely to be novel. RTL simulations showed that 10 of the 33 tests were novel. The percentage of novel tests was raised to $\frac{10}{33} = 30.30\%$ by test filtering, increasing the density of novel tests by a factor of $\frac{0.52}{30.30} = 58.28$.

Table 5.4: Results of instruction cache and branch prediction model based test filtering of 5000 tests

	Total tests	Useful test	%age useful tests
Without filtering	5000	26	0.52%
With filtering	33	10	30.30%
		Improvement	58x

The reasons for 70% of the tests being false positives:

1. ***Speculative execution***

   We use branch history to trace the speculative path. This assumes that the branch behaviors do not change. If new branches are discovered or branch types change(static branch to dynamic branch conversion) in the current iteration, the actual speculative path can be different from the estimation.

2. ***Modeling inaccuracies***

   The branch predictor behavior is estimated using simple rules, there is a 5% chance of error. This error arises from inaccuracy in predicting the BPU output of non-static branches.

3. ***Difference in implementation of way replacement***

   Some RTL optimizations cause minor deviations from ideal behavior. The way calculation can therefore differ from RTL behavior under some circumstances.

4. ***Branch marker information loss in L2 cache***

   Branch markers are destroyed if a cacheline is evicted from the L2 cache and written to main memory.

5.2.7 Conclusion

73% of tests that caused re-fetches of dense branches could be identified pre-simulation. We can also generate new tests with 30% confidence with a net increase in the test density by a factor of 58.

On one hand, behaviors such as branch discovery (generating markers) can be modeled deterministically. The branch predictor is based on sophisticated neural network based algorithm. Though this can also be modeled accurately, as seen in Sec. 5.2.6 the outcome can be estimated for 95% of branches using simple rules. The latter is therefore a pragmatic solution. On the other hand, non-determinism arises from microarchitectural behaviors that cannot be modeled with limited data. The instruction pipeline, for example, is a large state-machine that is practically impossible to model. Therefore, state dependent values such as speculation depth have to be set parametrically. These modeling errors are responsible for test escapes and false positives in the filtered tests.

5.3 Causing TLB address collisions

5.3.1 Background

Without any kind of address translation caching, every memory access for paged virtual memory logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. To avoid this overhead, processors rely on the temporal locality of code execution; just like memory accesses, the address translations for the accesses must also have locality. By keeping these address translations in a special cache, a memory access rarely requires a second access to translate the data. This special address translation cache is referred to as a translation lookaside buffer (TLB)[26].

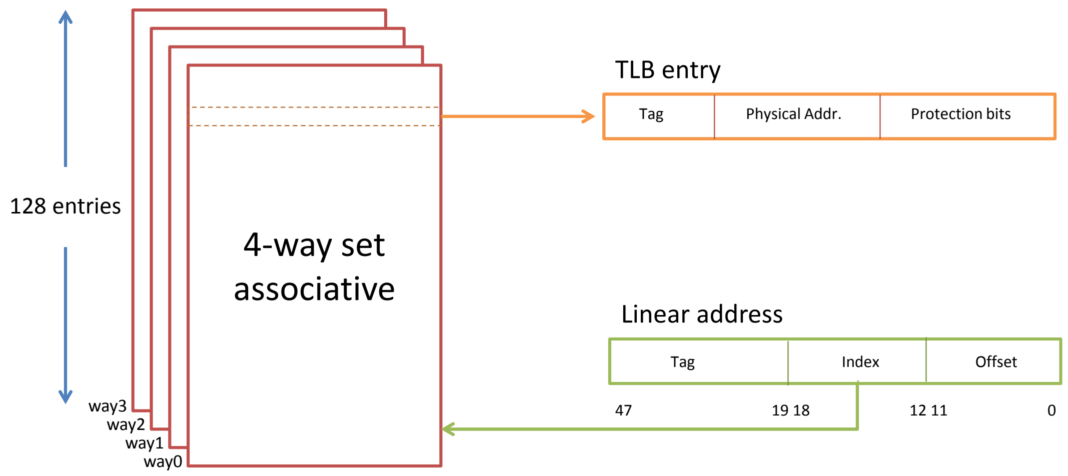


Figure 5.4: 4-way set associative TLB

N-way set associative TLBs pose an interesting challenge to random testing - generating tests such that all N indices (sets) of the cache are exercised. Fig. 5.4 describes a 4-way set associative cache. Each TLB entry is first mapped to one of the 128 indices (sets) present in the TLB structure. The index is a function of the linear address and is defined by bits LA[18:12]. The entry is then placed in one of the 4 ways corresponding to that index. This is determined by the TLB way replacement policy. A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page address and a protection field. The operating system changes these bits by changing the value in the page table and then invalidating the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits. Because the architecture is capable of supporting several guests in addition to the host operating system, the protection field includes an address-space identifier (ASID) value. This value specifies the operating system to which the linear address belongs. This allows multiple guests to securely share the same virtual address space.

5.3.2 Objective

The objective of this experiment is to insert entries into the uppermost way of the i-side TLB, i.e. way3. This has to be done in such a way that entries of all ASIDs are inserted at least once in way3. Since the ASID value is represented by a 3-bit value, there are eight possible ASIDs, 0 through 7. The host (VMM) assumes the ASID value of 0. The other 7 values are used by guests. The eight hits do not necessarily have to come from the same test. By doing this, we verify that multiple pages which share the same index (that belong to the same or different address space), can be correctly inserted and retrieved from the TLB.

An *address collision* is said to happen when two entries compete for the same index. For example, since the index is specified by LA[18:12], linear addresses 0xF3A654 and 0x2BA000 have an index value of 3Ah. To trigger address collisions, a random test generator must place code in pages that map to the same index (set). To hit way3, there has to be at least four entries that belong to the same set.

This property is one of the issues we identified in Amex in Sec. 3.3.2. While RPG was able to hit way3 of the TLB consistently, only a few Amex tests were capable to doing this.

5.3.3 Why ISA-based filtering does not suffice

To select tests pre-simulation that can cause TLB way3 insertions, we should be able to do the following:

1. Track TLB insertions and replacements

2. Recognize implicit and explicit TLB invalidations

While the latter can be done from simulation trace data, the former cannot. Similar to what was discussed in Sec. 4.5.5, this cannot be learned from the data generated by a few simulations. Unless we know the actual TLB structure and replacement policy, there are way too many hypotheses that need to be ruled out. So the sheer volume of data required to learn the TLB miss function and TLB replacement functions is enormous. However, if the TLB structure and replacement policy are already known, predicting TLB activity becomes trivial.

Table 5.5: Advantage of having a TLB model to generate TLB address collisions

RTL behavior	**Provided by ISA sim**	**Provided by MAM**
TLB lookup	Lookup address	-
TLB hit/miss	Lookup address TLB invalidation	TLB structure TLB replacement policy
Guest mode	VMRUN ASID change	-

Table. 5.5 summarizes the benefit of using a TLB model to filter tests. To know when TLB look-ups happen, we should know how the TLB is organized. Is the same TLB used for both 4K and 2M pages? If yes, is it a single level or a multi-level lookup? Any time a new page is accessed, we need to calculate its physical address. But to know whether or not a previous translation is still available, we need to know the TLB architecture. The current ASID and the ASID of the translated linear address are available in a trace.

5.3.4 Using a TLB model

The i-side TLB for 4K pages is 4-way set associative with 128 entries in each way. The fourth way is only written to when the lower three ways are already full. Our objective is to generate tests that use all four ways of the TLB, especially with guest addresses. Host translations are more frequent and easy to target.

A TLB insertion happens whenever an address translation is not already present in the TLB and it results in a page-walk. The translation is saved in the TLB to avoid expensive page-walks when accessing the same linear address again. An ISA simulator maintains a copy of the memory and it can read the page-table to perform linear to physical address translations. Because it lacks the actual TLB implementation details, it does not model TLB insertions or evictions.

As we are dealing with nested-paging enabled SVM tests, table walks are also nested. This means that every table-walk yields a guest-physical address, as opposed to a host-physical address. So there is an added task of converting the guest-physical address to a host-physical address to access memory(For details see 3.2). Luckily, nested page walks are executed in entirety by the table-walker; which is implemented in hardware. As a result, nested page-walks are performed transparent to the i-side. Every time a guest linear address is issued to the table-walker, it directly returns the host physical address (and does not return any of the intermediate guest-physical addresses). The i-side TLB only stores the guest-linear address to host-physical address translation.

Maintaining a TLB model entails four tasks:

1. ***Identifying when an address translation is required***

   An address translation is required whenever a new page is accessed. So crossing a page-boundary triggers a TLB lookup. This can happen due to sequential (un-branched) execution or from a control transfer event such as a branch, interrupt or exception. At any point in the trace, if one of these events are detected, a lookup of the TLB model is requested. In this case, since we are only interested in 4K pages, other TLB lookups are ignored.

2. ***Looking up the TLB model***

   Whenever a lookup is requested, the index of the requesting linear address is calculated. All four ways are then checked at the indexed location. If the tag of a valid entry matches with that of the lookup address, a hit is reported. Otherwise, a new entry has to be filled in the TLB.

3. ***Inserting a new TLB entry on TLB miss***

   If a translation is not present in the TLB, the translation is calculated by doing a table-walk. The result is then inserted into the TLB. The new entry is inserted at the lowest vacant way. If all four ways are valid, the least recently used entry is replaced.

4. ***Identifying implicit/explicit TLB invalidations***

   From time to time, the TLB is flushed by the hardware or by software. Software initiated TLB invalidations are referred to as explicit invalidations. For example, the INVLPG instruction invalidates the TLB entry of a single page. TLB flushes can also happen implicity. Updates to the CR3 control register cause the entire TLB to be invalidated, except for global pages. All these conditions must be detected and the TLB model has to be appropriately flushed.

5.3.5 Pseudocode

The pseudocode for maintaining the TLB model is given below:

```
main ()
        ..
        ..

        CALL clear_tlb

        WHILE (next instruction != NULL)
                //Update page start and size if a linear
                //translation is present
                IF (IsLinearTran) THEN
                        IF (LinearTran.pgsize is 4k) THEN
                                CALL do_tlb_lookup with CurrInstr.LinAd
                                                and CurrInstr.Asid
                        END
                END

                // Determine if a TLB look-up is required
                IF ((IsTblWalk || IsMemtypechange)
                        && TblWalk.pgsize is 4k) THEN
                        CALL do_tlb_lookup with CurrInstr.LinAd
                                        and CurrInstr.Asid
                END


                //Clear TLB when required
                IF (TLB invalidation is detected) THEN
                        CALL clr_tlb with Asid and IsClrGlobal
                END

        END
        ..
        ..

END

sub do_tlb_lookup (LinAd, Asid)
        set index to LinAd[18:12]
        IF (exists LinAd at index with correct Asid) THEN
                return
        ELSE
                FOR i = 0 to 3
                        IF (entry is not valid) THEN
                                insert_tlb_entry (LinAd, Asid, i)
                                return
                        END
                END
                insert_tlb_entry (LinAd, Asid, lru_way())
                return
        END
END
```

One interesting fact to note here is that the architecture simulator maintains a history of all translations in a unified (instruction and data) TLB of 'infinite size'. It also tracks the current page boundary. It retrieves the physical address from a past translation or by doing a table-walk. Both of these can be used as cues to update the TLB. A table-walk in the trace indicates a guaranteed TLB-miss. However, if a past translation is reused it is not necessarily present in the i-side

TLB. This is because of the limited size of the TLB and the fact that the i-side TLB is separate from the d-side TLB.

Since the actual physical address mapping is not relevant, we do not store any physical addresses in the TLB model. Each entry only contains a tag, ASID value, global bit and valid bit. This information is adequate to track the target behavior.

5.3.6 Results

Identifying existing novel tests

We first applied our test selection process on 5000 Amex generated tests that had been simulated, to identify known novel tests. The results are shown in Table. 5.6.

Table 5.6: Identifying known novel tests that i-side TLB way3 insertions

Test	Without TLB_CONTROL correction	With TLB_CONTROL correction
1	Y	Y
2	Y	Y
3	N	Y

3 of the 5000 simulated tests are novel. On the first iteration, only two of the three tests were identified. There were no false positives. There was one false negative because of an RTL optimization in processing the TLB_CONTROL field. RTL behavior sometimes makes minor deviations from architectural behav-

ior to improve performance, with no impact on software behavior. When this was accounted for in the TLB model, all three tests were recognized.

Generating new novel tests

The second part of the experiment is generating new tests and identifying novel tests before performing an RTL simulation. 18 tests were identified from 5000 tests. 12 of these were false positives. 8 false positives were caused by an RTL optimization similar to that of handling the TLB_CONTROL field of the VMCB. There was one false positive resulting from TLB flushes induced by testbench irritators (randomly inserted errors to simulate the effect of noise and alpha radiation). Irritators also help increase TLB traffic by inducing errors and forcing TLB flushes. Three false positives were traced to a filter bug.

Table 5.7: Results of TLB model based test filtering of 5000 tests

	Total tests	Useful test	%age useful tests
Without filtering	5000	3	0.06%
With filtering	11	6	54.54%
		Improvement	909x

The result obtained from applying the corrected model and filter to 5000 tests is shown in Fig. 5.7. Of the eleven detected tests, 6 tests are novel. So the percentage of novel tests in the simulated test-set increased from $\frac{3}{5000} = 0.06\%$ to $\frac{6}{11} = 54.54\%$. This is an increase in the novel test density by a factor of 909.

An interesting observation to be made here is that we did not have to make use of speculative execution. This is because TLB translations operate at the page level, as opposed to branch instructions which operate at the instruction level. So the odds of a entering a new page due to bad-path execution is much lower than forcing a new cacheline fetch. For 4K page, for a cacheline width of 64 bytes, we are $\frac{4 * 1024}{64} = 64$ times more likely to generate an IC fetch than to cause a TLB insertion.

5.3.7 Conclusion

From a pool of 5000 tests, 100% of the novel tests were identified by using a TLB model to track TLB transaction. Of the generated tests, 54.5% of the selected tests were novel. We achieved a 909x increase in novel test density.

Since this behavior is more deterministic compared to the first example, we are able to identify novel tests in past regressions with higher accuracy and generate new tests with higher confidence.

5.4 Summary

In this chapter, we demonstrate a non-intrusive methodology of using microarchitectural models to filter tests. We use extrapolate ISA trace data using MAMs to closely approximate RTL behavior. We use two different coverage points defined at the microprocessor core level to illustrate this idea. Both coverpoints have less than 1% likelihood of being hit using the current random test generation setup. The targeted events cannot be identified by applying any kind of ISA simulation trace based rule learning methods because of their high learning complexity.

In the first case, using an IC model, simplified BPU model and a heuristic to approximate the speculative nature of code execution, we were able to increase novel test density by a factor of 60. The second example was chosen to demonstrate that this approach can be used to address a limitation of Amex recognized in Chapter 3. Using a TLB model, the highest way of the i-side TLB was populated with guest translations. Our filtering methodology enables us to generate tests using an existing Amex template and increase the density of novel tests by a factor of over 900. Both experiments were conducted on AMD's latest x86-based microprocessor core.

Chapter 6

Conclusion and Future work

6.1 Conclusion

This dissertation evaluates the feasibility and effectiveness of developing practical data learning based solutions for functional verification, more specially, constrained random verification. Our proposed methodologies were developed based on the verification environment of a commercial x86 microprocessor and can be used as a complementary approach to existing verification flows without disruptions. We suggest practical methodologies to achieve full design closure based on existing verification plans. The suggested approaches include techniques that verification engineers can use to understand the deficiencies in their existing setup.

In Chapter 2 and Chapter 3, we explore the test generation problem. We study the design and use of two in-house random test program generators - Amex and RPG. Based on our understanding of their design, we discuss the differences in their test generation capability. We then explore possibilities of reducing test redundancy arising from the use of multiple exercisers based on microprocessor core-level verification of SVM nested-paging. We propose a novel flow of comparing two exercisers based on their functional coverage. We show experimentally, some of the differences observed in the working of the two exercisers. We use RPG as a baseline to identify relative deficiencies in Amex. Based on the functional coverage of 148 coverpoints, we conclude that Amex is superior to RPG in testing nested-paging behavior. It has better coverage than RPG across all but two coverpoints. Amex cannot be constrained to hit these coverpoints reliably. We explored the possibility of an external test filter to increase the density of these novel (targeted) tests.

In Chapter 4 we propose a test filtering methodology to select novel tests prior to RTL simulation based on ISA simulation traces. The filters are external to the test generators and independent of their design. Experimental evidence indicates that trace data is inadequate by itself to identify non-architectural behaviors such as speculative code execution. We explored the use of machine learning to extract rules (testing knowledge) from existing tests to identify novel test behavior. Experimental data shows us that this approach performs well, except when used to target microarchitecture intensive properties. For complex microarchitectural properties, it is impractical to extract accurate models from simulation data using machine learning algorithms.

In Chapter 5, we propose to use manually created microarchitectural models to impart testing knowledge into test filters. We illustrate the use of microarchitectural models to create tests filters describing complex design rules. We target design behaviors that occur with a frequency of less than 1% despite using best known test template configuration. We show experimentally that the density of novel tests can be increased by 60x to 900x by applying this framework. This includes a coverpoint that Amex could not be constrained for; proving that MAM based filtering can be used to target behaviors that test generators cannot be constrained accurately for.

6.2 Future work

6.2.1 Using performance models for test filtering

With the increasing size and complexity of SOCs, the interaction between the hardware structures built from billions of transistors is getting harder to predict through simple simulation or analytical models. Performance models have, therefore, become a critical tool in hardware design. They allow developers to study design alternatives and predict the performance of processors and systems long before actually building them. Today, design performance teams work alongside microprocessor RTL and verification to develop software models to accelerate hardware development. These models, implemented in programming languages or hardware description languages, can simulate programs to estimate design performance and correctness with considerable accuracy.

The two most commonly used models for performance estimation are

- Trace driven simulation model (TDM)
- Execution driven simulation model (EDM)

Trace-driven simulation

In trace-driven simulation, a complete execution trace is collected from running a performance benchmark. A real machine or functional simulator is used to execute a benchmark program/software in the native ISA binary. This binary is modified so that as each instruction is executed, information such as the control path values, instruction op-code, register value changes, memory operations and branch information is written out in a trace file. Traces are architecture agnostic; they can be run on any machine. They can either be used directly, to evaluate instruction set characteristics, or as input to a functional simulator to predict the performance of different architectural variants. Trace-driven simulations are most frequently used to study the behavior of memory architectures[33, 42].

Since a trace driven simulator doesn't actually execute code, it has a very low runtime. The downside of this is that it is not cycle-accurate. Traces lack timing information; they cannot model design behaviors like interrupts handling, probing, and locks. A timing simulator is used to generate timing information based on microarchitecure. Traces are also devoid of bad-path information. So the effects of speculative execution cannot be gleaned from a trace-driven simulation.

Execution driven simulation

In contrast to TDMs, in execution-driven simulation, the execution of the program and the simulation of the architecture are interleaved[17]. The simulator contains a functional simulator that emulates the target ISA. Simulating the architecture during the actual execution of a program permits us to determine the effects of the architecture on the execution of the program with a high degree of accuracy. Execution driven simulation provides timing information and bad-path information. There is no dilation in the program execution time, as the workloads are not modified for trace generation[42]. SimpleScalar[7] and GEMS[34] are some of the popular execution driven simulators.

Using EDMs for test selection

Execution driven simulators contain a great deal of implementation specific details that ISA simulators lack. This, combined with its ability to run random(exerciser generated) stimuli, makes it an excellent candidate for use in test filtering. The result of this simulation is a very rich dataset that can be used for test selection. Because the interaction of the stimulus with the microarchitecture is captured, this provides a more accurate approximation of test behavior in comparison to ISA simulation. Further, execution-driven simulation has a very low run-time. Simulation of 10,000 instruction on a single core takes approximately 120s(CPU time) on the EDM model, as opposed to approximately 20,000s for an RTL simulation.

Though such a filtering mechanism is not 100% accurate, it increases the odds of hitting the target behavior by two to three orders of magnitude. The ability to estimate test behavior at a fraction of simulation time using an existing design model can reduce coverage closure effort and time. Since performance models are developed alongside RTL every project, this is a very pragmatic approach to test selection. Preliminary experiments on this approach have shown very promising results.

6.2.2 Extending MAM based filtering

Microarchitectural model based filtering covers a wide gamut of design behaviors. This approach is, however, limited by our inability to do true speculative execution. Currently, instructions on the bad-path are not decoded or executed. Instead, we estimate bad-path behavior by using branch history. This restricts us to estimating the behavior of code that has been previously executed. Code which has not been executed before does not have a branch history, forcing us to assume that there are no branches. What is required is the ability to decode and execute instructions every time a branch is encountered. This entails manually steering the ISA simulator. For example, when we encounter a conditional jump for the first time, if it will be taken, we should be able to:

- Assume it will not be taken, save state and execute instructions down the bad path
- Stop execution wherever required and retreat to the branching instruction
- Resume execution down correct path.

This gives us the exact state changes caused by bad patch execution. However, architecture simulators do not support such customizations. Additionally, this limits the use of our methodology to the i-side. Without the ability to decode and execute instructions at will, data-cache activity which is transparent to software is hard to approximate.

6.2.3 Identifying temporal properties

Temporal properties refers to coverpoints that consist of two or more RTL events with a specific timing dependency. The exact time of occurrence of RTL events cannot be calculated, unless very complex MAMs are developed. This is because our test selection approach is based on architecture simulations, which are not cycle accurate simulations.

Despite this being a limitation, the MAM based approach is not completely ineffective. Our approach allows us to verify that tests contain all the events that comprise the targeted behavior. The occurrence of the target behavior can then be bounded to a window between the retirement of two consecutive instructions. For example, suppose we want to trigger a bus contention caused by requests coming from two modules in the same cycle. We can ensure that both modules generate a request within the same retirement window.

Bibliography

[1] A Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A Ziv. Genesys-pro: innovations in test program generation for functional processor verification. *Design Test of Computers, IEEE*, 21(2):84–93, Mar 2004.

[2] A Adir, E. Bin, O. Peled, and A Ziv. Piparazzi: a test program generator for micro-architecture flow verification. In *High-Level Design Validation and Test Workshop, 2003. Eighth IEEE International*, pages 23–28, Nov 2003.

[3] A Adir, E. Marcus, M. Rimon, and A Voskoboynik. Improving test quality through resource reallocation. In *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International*, pages 64–69, 2001.

[4] Inc. Advanced Micro Devices. *AMD64 Architecture ProgrammerâĂŹs Manual Volume 2: System Programming.* Advanced Micro Devices, Inc., 3.23 edition, May 2013.

[5] A. Aharon, A. Bar-David, R. Gewirtzman, E. Gofman, M. Leibowitz, and V. Shwartzburd. Dynamic process for the generation of biased pseudo-random test patterns for the functional verification of hardware designs, April 13 1993. US Patent 5,202,889.

[6] Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, and Gil Shurek. Test program generation for functional verification of powerpc processors in ibm. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 279–285, New York, NY, USA, 1995. ACM.

[7] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.

[8] N. Bamford, R.K. Bangalore, E. Chapman, H. Chavez, R. Dasari, Yinfang Lin, and E. Jimenez. Challenges in system on chip verification. In *Mi-*

croprocessor Test and Verification, 2006. MTV '06. Seventh International Workshop on, pages 52–60, Dec 2006.

[9] M. Benjamin, D. Geist, A Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 970–975, 1999.

[10] D. L. Bird and C.U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[11] M. Bose, Jongshin Shin, E.M. Rudnick, T. Dukes, and M Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 442–448 vol. 1, 2001.

[12] Leo Breiman. *Classification and regression trees.* Chapman & Hall, New York, N.Y, 1993.

[13] via Wikimedia Commons By Wgsimon (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0) or GFDL (http://www.gnu.org/copyleft/fdl.html)]. Transistor count and moore's law - 2011. http://upload.wikimedia.org/wikipedia/commons/0/00/Transistor_Count_and_Moore%27s_Law_-_2011.svg.

[14] Wen Chen, Li C. Wang, Jay Bhadra, and Magdy Abadir. Simulation Knowledge Extraction and Reuse in Constrained Random Processor Verification. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA, 2013. ACM.

[15] Todd Dukes, Jason Yeh, Randall Metzger, and Farhan Rahman. Abstract-state based x86 random stimulus generation. In *Proceedings of the 13th International Workshop on Microprocessor Test and Verification(MTV)*, MTV '13, 2012.

[16] Julia Dushina, Mike Benjamin, and Daniel Geist. Semi-formal test generation with genevieve. In *Proceedings of the 38th annual Design Automation Conference*, pages 617–622. ACM, 2001.

[17] S. Dwarkadas, J. R. Jump, and J. B. Sinclair. Execution-driven simulation of multiprocessors: Address and timing analysis. *ACM Trans. Model. Comput. Simul.*, 4(4):314–338, October 1994.

[18] Kerstin Eder, Peter Flach, and Hsiou-Wen Hsueh. *Towards automating simulation-based design verification using ILP*. Springer, 2007.

[19] Usama Fayyad and Keki Irani. Multi-interval discretization of continuous-valued attributes for classification learning. *In Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[20] Shai Fine, A Freund, I Jaeger, Y. Mansour, Yehuda Naveh, and A Ziv. Harnessing machine learning to improve the success rate of stimuli generation. *Computers, IEEE Transactions on*, 55(11):1344–1355, Nov 2006.

[21] Shai Fine and A Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Design Automation Conference, 2003. Proceedings*, pages 286–291, June 2003.

[22] Laurent Fournier, Yaron Arbetman, and Moshe Levinger. Functional verification methodology for microprocessors using the genesys test-program generator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '99, New York, NY, USA, 1999. ACM.

[23] Onur Guzey and Li-C Wang. Coverage-directed test generation through automatic constraint extraction. In *Proceedings of the 2007 IEEE International High Level Design Validation and Test Workshop-Volume 00*, pages 151–158. IEEE Computer Society, 2007.

[24] Jiawei Han. *Data mining concepts and techniques*. Elsevier Morgan Kaufmann, Amsterdam Boston San Francisco, CA, 2006.

[25] S.K.S. Hari, V.V.R. Konda, V. Kamakoti, V.M. Vedula, and K.S. Maneperambil. Automatic constraint based test generation for behavioral hdl models. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(4):408–421, April 2008.

[26] John Hennessy. *Computer architecture : a quantitative approach*. Morgan Kaufmann, Waltham, MA, 2012.

[27] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):9, 2009.

[28] R.C. Ho, C.H. Yang, M.A Horowitz, and D.L. Dill. Architecture validation for processors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 404–413, June 1995.

[29] Hsiou-Wen Hsueh and K. Eder. Test directive generation for functional coverage closure using inductive logic programming. In *High-Level Design Validation and Test Workshop, 2006. Eleventh Annual IEEE International*, pages 11–18, Nov 2006.

[30] Hiroaki Iwashita, Satoshi Kowatari, Tsuneo Nakata, and Fumiyasu Hirose. Automatic test program generation for pipelined processors. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 580–583. IEEE Computer Society Press, 1994.

[31] Vinayak Kamath, Farhan Rahman, and Li-C Wang. Analyzing efficacy of constrained test program generators - a case study. In *Microprocessor Test and Verification (MTV), 2012 13th International Workshop on*. IEEE Computer Society Press, 2013. ©[2013] Reprinted, with permission.

[32] Y. Katz, M. Rimon, A. Ziv, and G. Shaked. Learning microarchitectural behaviors to improve stimuli generation quality. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 848–853. IEEE, June 2011.

[33] E.J. Koldinger, S.J. Eggers, and H.M. Levy. On the validity of trace-driven simulation for multiprocessors. In *Computer Architecture, 1991. The 18th Annual International Symposium on*, pages 244–253, 1991.

[34] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[35] Shubu Mukherjee. *Architecture design for soft errors.* Morgan Kaufmann, 2011.

[36] Andrew Piziali. *Functional verification coverage measurement and analysis.* Kluwer Academic Publishers, Boston, 2004.

[37] John Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[38] John Ross Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan Kaufmann, 1993.

[39] Barbara Ryden. *Introduction to Cosmology*. Addison-Wesley, 2002.

[40] Arnold S. Tran, Richard A Forsberg, and Jack C. Lee. A vlsi design verification strategy. *IBM Journal of Research and Development*, 26(4):475–484, July 1982.

[41] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 175–180, 1999.

[42] Wen-Hann Wang and Jean-Loup Baer. Efficient trace-driven simulation method for cache performance analysis. *SIGMETRICS Perform. Eval. Rev.*, 18(1):27–36, April 1990.

[43] Y. Malka Y. Lichtenstein and A. Aharon. Model-based test generation for processor design verification. In *The American Association of AI's 6th Innovative Applications of Artificial Intelligence Conference (IAAI)*. AAAI Press, 1994.

[44] Hu-Hsi Yeh and Shao-Lun Huang. Automatic constraint generation for guided random simulation. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 613–618, Jan 2010.

[45] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-Based Verification*. Springer, 2006.

Appendix A

Pseudocodes

A.1 Approximation of speculative code execution behavior

```
// This subroutine returns the target cacheline linear address of speculatively
// executing code from a given linear address. This is done by searching the
// branch history for branches after the starting address. The return address
// is guaranteed to be outside of the current cacheline. If a loop is found,
// it returns the next (sequential) cacheline
        sub calc_spec_exec_tgt (CurrLinAd)
                SET TargetLinAd to zero
                //  Find the closest 64-byte-aligned boundary
                SET CurrCL = CL(CurrLinAd);

                // Finds the first branch after the CurrLinAd
                // in the current cacheline
               FOR EACH (branch in current cacheline)
                        // Look for branches starting from CurrLinAd
                        IF (CurrBrn.LinAd< CurrLinAd) THEN
                                NEXT
                         // if a branch was taken previously,
                         // assume it will be taken again
                        IF (CurrBrn.Type !~  "not_predicted"
                                                and Prev1_IsBrnTkn) THEN
                                SET FirstBrnInCL = CurrBrn.LinAd
                                SET TgtLinAd =  CurrBrn.Prev1Tgt
                                BREAK

                  // If there are no branches, target LA is the next cacheline
                IF  (TgtLinAd == 0) THEN
                        TgtLinAd = CurrCL+ 64
```

```
          // If we loop back to the same cacheline,
          // find out if there are other branches
         SET LastBranch to zero
          IF (CL(TgtLinAd) == CurrCL) THEN
                IF (TgtLinAd <= CurrLinAd) THEN //Ensure no infinite loops
                        FOR EACH (branch in current cacheline)
                                IF  (CurrBrn.LinAd > FirstBrnInCL)
                                        SET LastBranchInCL =  CurrBr.LinAd
                                        SET TgtLinAd =  CurrBrn.Prev1Tgt

                        IF (LastBranch != 0) THEN
                                        // A smaller fwd jmp within a bkwd
                                        // jmp can cause an infinite loop
                                        IF (TgtLinAd < FirstBrnInCLl) THEN
                                                SET LastBranch to zero
                                ELSE
                                         // There is a branch between the
                                         // target and the source that
                                         // hasn't been discovered yet.
                                         // So it appears like an infinite
                                         // loop We assume naively that it
                                         // will take us to the next cacheline
                                        RETURN CurrCL + 64

                                // We did find a branch that takes us
                                // out of the cacheline
                                IF (CL(LastBr) != CurrCL) THEN
                                        RETURN TgtLinAd
                                // If the branch keeps us in the current CL,
                                // ensure its not backward
                                ELSIF ( CL(TgtLinAd) == CurrCL &&  TgtLinAd
                                                < LastBracnh) THEN
                                        // Skip branch if it is backward
                                        RETURN calc_spec_exec_tgt
                                                        (FirstBrInCL + 1)
                                ELSE
                                // Fwd jmp within cacheline. Find another
                                // way out
                               RETURN calc_spec_exec_tgt(TgtLinAd)
                        ELSE
                         // Fwd jmp within cl. Find another way out
                        RETURN calc_spec_exec_tgt(TgtLinAd)
                 ELSE // Target is some other cacheline
                        RETURN TgtLinAd

END
```