An Efficient Directed Random Instruction Tester for High Coverage

Feng Zou, SHZ1-3-L4, ([feng.zou@intel.com](mailto:feng.zou@intel.com))

Gong Chen, SHZ1-3-K4, ([gong.g.chen@intel.com](mailto:gong.g.chen@intel.com))

Abstract

Traditional random instruction tester generates enormous number of random tests to fully validate the device or software under test. Running these tests requires significant amount of resource and time. This paper introduces an efficient directed random instruction tester – ART (Affordable Random Tester), which achieves same level of coverage with less random instruction tests generated. It employs ECP (Equivalence Class Partitioning) and BVA (Boundary Value Analysis) concepts to generate a subset of instruction mixes covering interesting conditions. It can be specified to generate special instruction sequences, so that optimizations for instruction sequences are covered. It’s used for validating Houdini, which transparently run majority of Android NDK applications on ATOM-based handheld devices. ISA (Instruction Set Architecture) coverage metrics is defined for measuring the quality of ART generated tests.

# Introduction

RIT (random instruction tester) [1] is successfully used by CPU validation. It could generate corner cases to stress conditions and data boundaries. There have many RIT tools widely used in Intel CPU pre-silicon and post-silicon validation team, like Janus and Café[3] [10]. uBT validation team has successful used Café to validate uBT and CMS [7]. There also have some RITs developed outside Intel. TiGeR [6] is used by Transmeta CPU and Code Morphing Software (CMS) validation. RAVEN [12] (Random Architecture Verification Engine) is a robust random test generator for processor validation and it’s claimed to support ARM [8] and x86.

However, traditional RIT generates tremendous amount of tests to fully validate the CPU/software under test and achieve high coverage. For Nehalem post-silicon validation with Café, it takes 10 weeks to generate and run 84 million random tests with 40 hosts, 40 targets and 400 generators (machine for generating random tests). As the limitation of time and resource, it couldn’t be possible to generate and run all of possible instruction mixes and achieve full coverage. So we must define a finite subset of tests from all possible tests and in the meantime, to achieve similar level of coverage.

Equivalence Class Partitioning (ECP) and Boundary Value Analysis (BVA) are the useful methodologies for this goal: achieving high coverage with few tests [4][5][11]. ECP [4] is to divide input variable data into subsets. In a given subset, any element should produce the same results. It could help reducing the number of tests from all possible tests and still provide with high degree of confidence. Test cases should be generated to cover each subset at least once.

BVA [4] is used to generate tests with the boundary conditions. The evidence demonstrates there have lots of problems/bugs occurs at or near the boundary conditions of independent input and output parameters.

In order to validate the optimizer for instruction sequences, we define FSMs (Finite State Machine) to direct ART generating sequences with specific control and data flow. It’s the similar concept with production in TiGeR [6], which is proven useful for validating emulator software.

Architectural/ISA coverage is defined to measure how many conditions are hit, provide feedback to ART for generating effective tests and ensure ART generating enough corner cases to validate Houdini and provide confidence with quality.

The rest of this paper is organized as follows. Section 2 provides an overview of ART generated test. Section 3 describes ECP and BVA methodology. Section 4 explains single instruction generation flow in ART. Section 5 introduces sequence generation flow in ART. Section 6 presents the coverage definition for single instruction and sequence. Section 7 shows the bugs exposed by ART generated tests and the related coverage data. Section 8 summarizes our work and introduces future work.

# Overview of ART generated test

ART generates random instruction test with three parts: setup code, random code and self-checking code. Setup code initializes the context and prepares the running environment for random code. Random code includes number of random instructions being generated. Self-checking code in the end checks the context after random code being executed against the reference context in the data section of random instruction test.



Figure 1 execution flow of a random test

ART generates a random instruction test in three steps. Firstly, an initial random instruction test is generated by ART, only including setup code and random code. Secondly, the initial test is executed on the golden reference model, and then the reference result at the end of random code is collected from the log of golden reference model. Finally, self-checking code is generated to compare the context with the reference result, and is rebuilt together with setup code and random code included in the initial random test. After being generated, the random instruction test is not only used to validate, but also used to collect its ISA coverage statistics to measure the test’s quality.

# ECP and BVA methodology

In this section, we present ECP (Equivalence Class Partitioning) and BVA (Boundary Value Analysis) techniques with an example. The example below is a function to be validated.

int foo (int a)

{

if ((a>=0) && (a<=12)) {

return 1;

}

return -1;

}

With exhaustive testing technique, tests with all possible inputs are used to validate this function. The total number of tests is 4,294,967,296 (232), as the size of int type is 32-bit. Obviously, there have lots of redundant tests, which would take a lot of time and resource.

ECP [4] helps systematically reducing the number of tests from all possible tests and still provides similar level of coverage. The variables or variable combinations in the same subset produce the same expected result. With ECP technique, the test inputs are divided into 3 partitions:

[-231 ~ -1], [0, 12], [13, 231-1]

That is, three tests for covering three partitions are enough to validate the function.

BVA (Boundary Value Analysis) is also a functional testing technique that targets data values at, immediately above, and immediately below a specific boundary condition [4]. Historical experience demonstrates that bugs are always occurred at or near the boundary conditions of independent input or output parameters. With BVA technique, the test inputs have 10 boundary values:

-231, -231+1, -1, 0, 1, 11, 12, 13, 231-2, 231-1

Combining ECP and BVA techniques could be more effective as it could reduce the number of tests and contain the boundary values easy to expose bugs. With ECP and BVA techniques together, the test inputs are divided into 10 boundary values and 3 partitions:

-231, -231+1, [-231+2, -2], -1, 0, 1, [2, 10], 11, 12, 13, [14, 231-3], 231-2, 231-1

The total number of ECP/BVA tests is 13, including the inputs with boundary values or random values in 3 partitions. ECP/BVA techniques illustrate the number of tests for possible inputs is reduced from 4,294,967,296 possible tests to 13 tests.

# Single instruction generation

ART generates an instruction by following these steps below:

1. Firstly, ART randomly selects an instruction opcode from the internal instruction description. ART keeps an internal instruction description for each instruction. For each instruction, it describes the opcode, opcode mask, and ECP/BVA values for each operand.
2. Secondly, ART generates a random 32-bit number, and set it to the selected opcode by applying the opcode mask.
3. Thirdly, ART sets the instruction fields for source registers, destination register and immediate based on ECP/BVA techniques. For example, the source register is classified into six subsets: [R0, R3], [R4, R7], [R8, R12], R13, R14, R15. Source registers are selected randomly from the six subsets defined above. If [R0, R3] subset is selected, the source register number is selected randomly from one of R0, R1, R2 and R3.
4. Finally, ART filters out invalid instructions and adds the preamble instructions. The preamble instruction is used to make the generated memory instruction accessing the valid memory range, or to set the register using by the generated instruction with the random value from ECP/BVA subsets.

ART uses binary code to represent the random instruction generated, instead of using assembly instruction, in order to cover all possible encoding formats, including valid and invalid formats. The limitation of assembly instruction is that the instruction has multiple possible encoding formats and only one of them would be used and tested. For example, the immediate value in data-processing instruction is encoded into a base number with a shift rotation. Some constant values may have different combinations of base and rotation values and the assembler only chooses the encoding with lowest rotation value.

# Sequence generation

ART defines FSMs (Finite State Machine) to generate specific instruction sequence for validating optimization techniques, as it couldn’t generate specific instruction sequence to validate optimization effectively without any direction. The FSM concept in ART is similar with production in TiGeR [6], FSM and recipe in café [9], but the FSMs are defined for the specific optimizations, including control flows and data flows.

Figure **3** lists all types of control flow types currently supported in ART, including by-pass, hammock, loop and irreducible loop and call-ret flow. All of these different flows are defined as sub-classes of a basic block so that it’s easy to generate code with different kinds of control-flow nested together. Among them, the call-ret flow is implemented to support special interwork function call between ARM and Thumb instruction set [8].

Figure 3 control flows defined in ART

ART also defines some FSMs for data flows to validate data optimization, which could be used together with FSMs for control flows to guide instruction sequence generation. Below is some examples for data-flow FSM.

1. sequence for validating register allocation

Register allocator is a common optimization technique, which is used to map the virtual register used by optimization to the physical register. For validating this functionality, ART defines three types of code sequence below: write\_reg\_seq, reg\_joint\_seq and read\_reg\_seq.

write\_reg\_seq{

dst\_regs: [R0, R3], [R4, R7], [R8, R12], R13, R14

}

reg\_joint\_seq{

src\_regs: NOT write\_reg\_seq.dst\_regs,

dst\_regs: NOT write\_reg\_seq.dst\_regs,

}

read\_reg\_seq{

src\_regs: write\_reg\_seq.dst\_regs,

}

FSM\_regalloc{

default{ next\_state{ write\_reg\_seq } }

write\_reg\_seq{ next\_state{ reg\_joint\_seq } }

reg\_joint\_seq{ next\_state{ read\_reg\_seq } }

read\_reg\_seq{ next\_state{ reg\_joint\_seq } }

}

In this FSM for validating register allocator, the instructions in write\_reg\_seq sequence use couple of registers as the destination registers, and the following instructions in read\_reg\_seq sequence use the same destination registers in write\_reg\_seq as the source registers. The instructions in reg\_joint\_seq sequence don’t touch any of them at all. The FSM creates the dependency on reading and writing registers, and could be used to direct ART generating the corresponding sequence.

By combining hammock control flow and this data flow, ART would generate the corresponding sequences: write\_reg\_seq, reg\_joint\_seq, reg\_joint\_seq and read\_reg\_seq for block 1 to 4. It would lead to long live range for selected registers and is helpful to register allocation validation.

1. Sequence for validating condition code optimization

Conditional execution is an important feature provided by ARM architecture [8]. A conditionally executed instruction takes effect if the flags satisfy the condition specified in the instruction. Otherwise, it acts as a NOP without taking any effect. Most instructions can also be chosen to update these flags or not depending on the value of S bit.

This optimization for conditional execution is to map flags in source architecture into flags in target architecture and update the flags into memory if necessary. So we also define three types of sequences: def\_flag\_seq, use\_flag\_seq and joint\_flag\_seq. Instructions in def\_flag\_seq with S bit set will modify flags; instructions in use\_flag\_seq will be conditionally executed; for joint\_flag\_seq, instructions will not update flags and be executed under all conditions. While a test case is generated in this scenario, these different types of sequences are chosen in each block of specific control flow.

1. Sequence with stressing the same kind of instructions

Stressing the same kind of instructions is to generate the same type of instructions continuously. In practice, except generating instructions with same kind of instructions such as memory instructions or multiply instructions, ART also generate other special types. One is using multiply instructions and shift instructions together. The reason for this is that these two kinds of instructions are using fixed register implemented in optimizer. Another is generating memory instructions with adjacent or discontinuous addresses. The memory address is selected randomly from the options: random address, last address+4, last address-4, last address+8, last address-8. This is used for validating the peephole optimization for memory load/store with continuous address.

# Coverage definition

Coverage metrics is used to measure how many conditions are hit and how many tests are enough to hit all of the conditions. It also provides feedback to random instruction tester on the efficiency of the generated tests. Based on ECP/BVA techniques, we define two kinds of ISA coverage for ART generated tests: single instruction and sequence coverage. They’re same with the coverage definition in eArchSpec[2] . Single instruction coverage is same with the coverage defined by cross items, and sequence coverage is same with the coverage defined by event.

Single instruction coverage is the sum of all cross items, and the latter is the combination of basic items. Usually, an instruction might consist of conditional execution bits, s bit, several source registers, one destination register, and immediate. By means of ECP/BVA technology, basic items for these fields are defined as below (Figure 4).



Figure 4 basic item definitions

Here “ADD (register)” is taken as an example:



Figure 5 definition of “ADD (register)” instruction [8]

These instruction fields in ADD (register) are classified into the relevant kinds of basic items in Figure 4. In Figure 5, Rn, Rm are source registers (src\_regs basic item), Rd is the destination register (dst\_regs basic item), type is shift type (shift\_type basic item), imm5 is the immediate with 5-bit long (s\_bit basic item), cond is the bits specified for instruction execution condition (cond\_bits basic item) and S is the s bit indicating updating the status register or not (s\_bit basic item).

The cross item for this instruction is defined as:

cross ADD(register), cond\_bits, s\_bit, src\_regs, src\_regs, dst\_regs, shift\_type, imm5

For ADD (register) instruction as the example above, the coverage for this cross item is equal to the multiplication of every basic items.

Sequence coverage includes the coverage for all FSMs, and the full coverage for each FSM is the multiplication of every FSM states’ coverage, which is the sum of cross items for the instructions satisfying the definition in the FSM state. The sequence including the same kind of instructions is described as below (ALU\_OP, LOG\_OP and others are the instruction categories):

same\_instrs\_seq {

instrs{ // OR - use one kind of instructions

OR ALU\_OP LOG\_OP MOV\_OP …,

}

}

FSM\_sameinstrs {

default {

next\_state { same\_instrs\_seq }

}

}

# Results

ART is proved to an important validation tool in Houdini project. It greatly cuts down the validation cycle by using ECP/BVA techniques, as it reduces the total number of test cases to be generated but produce a subset of all possible input values covering all of boundary conditions. It exposes many bugs in Houdini as it implements the specific FSMs to direct sequence generation and effectively validate optimization existing in Houdini.

Apparently, ECP/BVA coverage is much less than the coverage without these techniques. It speeds up the validation and covers all of boundary conditions and subsets. For the example of ADD (register) instruction above, without employing ECP/BVA techniques, the coverage of this instruction is equal to the multiplication of the number of possible values for every instruction fields.

cross\_item\_coverage(ADD) = sum1(cond\_bits) \* sum1(s\_bit) \* sum1(src\_regs) \* sum1(src\_regs)

\* sum1(dst\_regs) \* sum1(shift\_type) \* sum1(imm5)

= 24 \* 21 \* 24 \* 24 \* 24 \* 22 \* 25 = 16,777,216

The result of sum1() function is the number of element included in the corresponding instruction fields.

With using ECP/BVA methodology, the coverage for this cross item is equal to the multiplication of every basic items:

cross\_item\_coverage(ADD) = sum2(cond\_bits) \* sum2(s\_bit) \* sum2(src\_regs) \* sum2(src\_regs)

\* sum2(dst\_regs) \* sum2(shift\_type) \* sum2(imm5)

= 2 \* 2 \* 6 \* 6 \* 6 \* 4 \* 5 = 17,280

The result of sum2() function is the number of element included in the corresponding basic item.

The full coverage for single instruction is the sum of cross items for all instructions, which is 1,930,685 in total. We have the tradeoff to categorize the instructions and sum up the maximum number of cross items in each kind of instruction as the goal for our first validation stage with ART. The sum is 352,708, which could be achievable in millions of tests.

The sequence coverage includes the coverage for all FSMs defined in ART, while coverage for a FSM is defined as the multiplication of the cross items for the instructions qualified to the FSM state. For example, the full coverage of FSM\_sameinstrs described in section 6 is calculated as below:

full\_coverage(FSM\_sameinstrs) = full\_coverage (same\_instrs\_seq) = 902244.

But the number is a little big and might not be achieved in millions of tests, as there have other FSMs to be selected. Here we have a tradeoff to use the cross item value of the instruction, having maximum value of cross item and qualified to the FSM state, as the coverage number for the state. The coverage goal is set to the sum of every key states’ coverage.

coverage(FSM\_sameinstrs) = coverage (same\_instrs\_seq)

= maximum cross item value of the instruction qualified to same\_instrs\_seq state

= 130000.

In Houdini v1.0 release, 2,000,000 tests with about 100 instructions are generated by ART to validate Houdini. The coverage for single instruction is 100%, including 100% for ARM instructions and 100% for Thumb instructions. And the coverage for sequence coverage is 100%. For achieve the acceptable coverage goal, the time for generating and running ART tests is 40 hours on Intel Xeon X5650 2.67GHz with 10 threads, much less than 10 weeks for Nehalem post-silicon validation with Café - 40 hosts, 40 targets, 84 million seeds, 400 machines for generating tests.

ART is a very effective validation tool, as it exposes 37 bugs for Houdini v1.0 release, which percentage is 29.2% of total bugs. Two examples of bugs below are described to demonstrate the effectiveness of ART by employing ECP/BVA methodology.

A bug exposed by ART is found in MSR\_a\_T1 instruction definition for decoder. Originally, the definition doesn’t set the correct flag mask and the instruction doesn’t write the flag register. ART generates an MSR instruction with the binary code - 0xf38e8c00. The relevant assembly code is:

MSR APSR, R14

Before this instruction, R14 is equal to 0x12008020. According to the semantics in ARM manual [8], v flag in APSR register is changed to 1, as the value in R14<28>. The incorrect mask for flag value in definition files leads to the incorrect flag after the instruction being executed.

Another bug is exposed by the instruction sequence generated:

i1:     subs          r6, r5, r2          // update n, z, c and v flags

i2:     bics           r2, r0                  // update n and z flags

i3:     adcs          r3, r1                  // use c flag; update n, z, c and v flags

i4:     ldr              r6, [pc, #64]

In this sequence with four instructions above, register r6 is written by instruction i1 and i4 but isn’t touched between them. So instruction i1 is treated as a redundant operation and optimized. But in fact, instruction i1 updates the c flag which is used by instruction i3. So the sub operation should be kept to update the flag.

# Summary and Future Work

ART employs ECP/BVA techniques to generate less random tests but achieve the same coverage with traditional RITs. It’s more effective for bug-hunting, as it takes less time to hit most of boundary conditions and expose bugs. It implements specific FSMs to direct sequence generation and validate optimization, which is proved by exposing many bugs in optimization.

We encountered some issues during ART development. The golden simulator used for generating reference results might have bugs and produces inconsistent results against real CPU. We try to run the failed tests on real CPU to exclude the simulator issue before analyzing and reporting the bugs. The instruction description using for generation contains bugs and might lead to generate wrong tests. However, another verification tool is applied to validate the instruction description. ART generated test is executed in single thread. But it’s enough for us to validate Houdini as there have some focus tests for validating multi-thread scenarios.

In the future, we would analyze the generated tests and check what the coverage holes of single instruction and sequence are. This work would greatly help the efficiency of the generated tests and exposing more optimizer bugs. We need to review the definition for single instruction and sequence coverage. The single instruction coverage might be defined as the average percentage of every kinds of instructions. The sequence coverage should be referred to the sequence coverage defined in eArchSpec [2].

Acknowledgments

We’d like to thank Sophie Chen, Frank Zhang, and Young Wang for leading the validation tool development. We also would like to thank Jianping Chen, Eric Lin, and Jianhui Li for providing valuable feedback. Thanks to Honesty, Young and other people in BiTS team providing supports.

Bibliography/References

1. The Art of microarchitecture validation, June 2011
2. Yiling He, Seetharaman Balasubramanian, Hooi Koon Ooi, eArchSpec: New IA32 Architecture Coverage Tool, DTTC, 2003
3. Keith Mac Donald, Café Biasing innovation – Ideas For Other RIT Developers, DTTC, 2007
4. Alan Page, Ken Johnston, Bj Rollison, How We Test Software at Microsoft
5. Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler, The Art of Software Testing
6. Anshuman S Nadkarni, Tom Kenville, TiGeR, the Transmeta Instruction GEneratoR: A production based, pseudo random instruction, x86 test generator, IEEE 2004
7. Jiong Gong, Pandit Amol, Christopher Lishka, et al, Validate Software with Café: Challenges and BKMs, DTTC, 2011
8. ARM Architecture Reference Manual, 2010
9. Café User’s Guide Vol. 1 and 2, Version 2.3, Jan 2007
10. D. C. Henkemeyer, Using Café in Pre-Silicon Validation, DTTC, 2008
11. Young Wang, Xiangning Ma, Haihao Shen, et al, Tracking Software Quality with Health Metrics, DTTC, 2011
12. RAVEN: <http://www.arm.com/community/partners/display_product/rw/ProductId/5171/>