EATBit: Effective Automated Test for Binary Translation with High Code Coverage

Hui Guo, Zhenjiang Wang, Chenggang Wu*
State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences
{guohui, wangzhenjiang, wucg}@ict.ac.cn

Ruining He
Department of Computer Science and Engineering
University of California, San Diego
r4he@eng.ucsd.edu

***Abstract*—Binary translation makes it convenient to emulate one instruction set by another. Nowadays, it is growing in popularity in various applications, especially the embedded platforms. When it comes to the test of binary translators, traditional methodologies which still mainly rely on manual unit test is costly, labor intensive and often not adequate to test complicated algorithms in the translators. Some standard benchmark suites, like SPEC CPU2006, are compiled with different compilation options for further tests. However, the translation modules still have over 30% of their code unexecuted after such tests, according to our experimental results. Methodologies based on randomization can generate a vast variety of tests, thus improve the code coverage in the translation system. In this paper, we propose such an approach named EATBit. Test binaries are generated with randomly selected instructions and operands. The binaries and a large amount of input data are then refined to exclude invalid ones. Experimental results on a real binary translator demonstrate that EATBit can not only improve code coverage by over 20%, but also find some new bugs in the translator successfully.**

I. Introduction

In recent years, with the rapid development of the semiconductor industry, we have been surrounded by all kinds of portable devices, system-on-chip designs, and multi-purpose appliances with different Instruction Set Architectures (ISAs). Fortunately, binary translation provides a fast, inexpensive way to migrate software from one ISA to another without the need of its source code. Binaries from source ISA are disassembled and translated into native code, which will be executed to conduct the emulation. What's more, when the source and destination ISAs are the same, binary translation is also used for dynamic optimization [1] [2], instrumentation [3] [4], software detection and security enforcement [5].

Testing and debugging a binary translator is challenging because of its special input, binaries. They can be compiled from various programming languages, with all kinds of compilers and optimizations. Some of them are even generated from assembly languages with much more freedom. Therefore, developers have to spend much of the time conducting test cases, trying to represent as much binary characteristics as possible, which is costly and labor intensive. From our experience, testing usually takes more than one-third of a binary translator's development process. With the continuing emergence of novel transformation methods and optimizations in binary translation, it becomes more and more unbearable to design test cases manually. However, even with such efforts, test cases generated in predefined strategies are still inadequate compared with the vast variety of binary characteristics.

Apart from manual test cases, some standard benchmark suites are often used, such as SPEC CPU2006 [6], Super Test [7] and LTP [8]. They are compiled with different compilation options to measure the binary translator's correctness and performance. However, experimental results show that more than 30% of code in the translator remains unexecuted, and therefore untested after such tests. Besides, those benchmarks are typically more complicated and run slower than normal test cases, making it more difficult to trace and debug.

Inspired by the predicament faced by the developers of binary translators, we propose an automated way to generate tester-customized effective test binaries with high code coverage. The key contributions of this paper include:

- Instruction categories and algorithms. Testers can use them to customize test binaries according to their own requirement.
- Operand sliding window mechanism for effective test binary generation. In this paper, we proposed an operand sliding window mechanism to guarantee the generated binaries to be compact and efficient to test.
- Static instruction wrapping and dynamic training. After the wrapping and dynamic training, test binaries with potential exceptions can be screened out successfully.

We also studied the core size and quantity of the generated binaries and their impact on the code coverage. Code coverage is widely used in white box testing in software engineering [9]. Experimental results based on DigitalBridge [10] demonstrate that our approach can improve code coverage by over 20%, compared with traditional test cases or SPEC CPU2006 benchmarks. Additionally, our approach also found some new bugs in this binary translation system.

The rest of this paper is structured as follows. Section II introduces some related works. Section III presents techniques we employed to automatically generate raw test binaries. In section IV, we explain how we refined the raw test binaries and their input data. The experimental results on benchmarks are given in Section V. Section VI discusses the limitations and future work, and Section VII concludes the paper.

*To whom correspondence should be addressed.

978-3-9815370-2-4/DATE14/©2014 EDAA

II. Related Work

Randomized methods have been studied in the field of compiler testing for many years. As early as 1962, Sauder [11] developed a method of generating program test data to test the logic of COBOL compilers. In 1970, Hanford [12] employed a syntax definition in a formal notation to drive the generation of random test cases. Burgess and Saidi [13] designed an automated generator of test cases for FORTRAN compilers.

In recent years, Lindig [14] proposed a C compiler testing tool named Quest, with a built-in scripting language driving test program generation to find compiler bugs concerning calling conventions. Thirteen new bugs in mature open source and commercial C compilers were uncovered by Quest. Sheridan [15] used a simple technique for testing a C99 compiler, by comparing its output with the output from pre-existing tools. Eighteen bugs in the GNU C compiler, as well as a dozen bugs in versions of GCC for the ARM processor were found by this approach. Zhao et al. [16] built a script-driven test program generator to test an embedded C++ compiler. Through a script, their tool accepts general test requirement, like which optimization to test, and sets up a program template to drive further test case generation. They reported greatly improved statement coverage in tested modules and found several new compiler bugs in GCC. Yang et al. [17] created Csmith, a randomized test case generation tool for C compilers. Csmith can avoid undefined and unspecified behaviors and generate programs which cover a large subset of C. J. Regehr et al. [18] tried to automatically reduce large C programs that trigger compiler bugs into smaller ones, so that they could be inserted into compiler bug reports directly.

Although randomized methodologies have attracted much research attention in the field of compiler testing, to the best of our knowledge, hardly any similar research can be found in the area of binary translation testing. Since binary translation is on a lower level than compilation, the characteristics, theories, and methodologies of binary translation are quite different. This makes common methodologies used by compiler testing not applicable to the automated testing of binary translators.

One thought is to generate test programs by existing compiler testing methods, compile these programs into binaries, and then use them to test binary translators. However, it is difficult for such methods to conduct customized functionality test, because the detailed instruction selection and scheduling are partially and indirectly controlled by compiler options. In practice, new transformation and optimizations are usually continuously added to the binary translator, producing new functionality testing needs. It would thus be inconvenient and cost more without the ability to customize instruction details. Therefore, its necessary to address the specific challenges faced by automatic binary translation testing.

III. Randomized Generation Of Raw Test Binaries

The most important part in a test binary is the *test core*, which is a sequence of randomly selected instructions other than control transfer ones (e.g., jump, branch or call). In order to generate customized cores, we defined some typical categories and algorithms to generate instruction sequences. These can be used directly by testers to express their specific test purposes and guide the randomized generation of raw test binaries. An operand sliding window mechanism is then proposed for operands selection, so that the effect of each instruction can be accumulated into the final result. It makes the test core compact and efficient for testing. Finally, the generated core is compiled using inline assembly along with its input/output facilities.

A. Instruction Category and Algorithm

A test core consisting of arbitrary instructions can be useful, but in many cases, testers may want them within a subset of instructions. For example, a test core should not contain floating-point instructions when the binary translator is simulating a platform without floating-point function units. Therefore, we defined some instruction categories for testers. In some cases, testers need to enumerate some kind of instructions to ensure a full test, so we defined some algorithms for such enumeration need. Table I lists some of them and their typical instructions or sequences on IA32 platform.

Testers can specify the constraints of each instruction with the help of the categories and algorithms. We will give such an example aiming to test the flag optimization, usually the first translation optimization when developing a binary translator.

Flag register is used on x86, ARM, and some other platforms. The naïve emulation of hardware flag calculation will incur large overhead, especially on platforms without such hardware support, e.g., MIPS. The elimination of redundant flag calculation is relatively easy with data-flow analyses, while optimizations on flag calculation algorithms are still mostly rely on programmers' craft. The optimal translation of a flag differs depending on the instructions that define or use the flag, the extension mode of native register operands, etc. Therefore, it is necessary to design some tests aiming on these translation algorithms.

Such a test core consists of some flag definition instructions followed by some flag using ones. In case a flag is used without any previous instruction defining it, we build a *minimal flag full-definition set*. Elements in the set are sequences of instructions in which all flags are defined, and removing any instruction will leave some flag undefined. The test core start with a random element in the set, and then followed by some instructions using any flags. The pseudo code building such a test core is shown as follows.

```
 1: procedure GEN_CORE($size\_limit, min\_ffd\_set$)
 2:     $core\_size \leftarrow 0$
 3:     $opcode\_seq \leftarrow$ ENUMERATE($min\_ffd\_set$)
 4:     for $opcode$ in $opcode\_seq$ do
 5:         $operands \leftarrow$ SELECT_OPERANDS($opcode$)
 6:         EMIT($opcode, operands$)
 7:         $core\_size \leftarrow core\_size + 1$
 8:     end for
 9:     while $core\_size \leq size\_limit$ do
10:         $opcode \leftarrow$ RANDOM($flag\_use\_any$)
11:         $operands \leftarrow$ SELECT_OPERANDS($opcode$)
12:         EMIT($opcode, operands$)
13:         $core\_size \leftarrow core\_size + 1$
14:     end while
15: end procedure
```

TABLE I. INSTRUCTION CATEGORY AND ALGORITHM EXAMPLES

Definition		Description	Typical Instructions or Sequences
Category	arithmetic	integer arithmetic instructions	ADD, SUB, ADC
	logical	logical instructions	AND, OR, XOR
	flag_def_any	instructions that define any flag	ADD, POPF, CLC
	flag_def_all	instructions that define all 6 flags	ADD, POPF
	flag_use_cf	instructions that use CF flag	ADC, SETNZ, PUSHF
Algorithm	minimal flag full_definition set	minimal instruction sequences that define all 6 flags	{{INC, STC}, {ADD}, ...}
	CFOF flag some_use set	instruction sequences that read CF and OF flags	{{flag_use_cf - flag_use_of}, {flag_use_of - flag_use_cf}, ...}

B. Operand Sliding Window

In order to make test cores compact and efficient, we try to avoid idle destination operands, i.e., operands that are overwritten before their values are used. Those operands do not affect the test results, so that the correctness of instructions generating such operands are not reliably tested. Therefore, we propose an *operand sliding window* (OSW) mechanism to avoid such cases.

Initially, all registers and specified memory locations are set to the values given by the test binary input, and the OSW contains nothing. As more instructions are generated, the OSW traces and records the destination operands of every instruction, sorted by their last reference time. They are used to direct how the following instructions select source and destination operands. The following algorithm shows how to extend and shrink the register OSW along with the operand selection.

```
 1: procedure SELECT_SRC_OPND($osw$)
 2:     if LEN($osw$) $\geq threshold$ then
 3:         $operand \leftarrow$ HEAD($osw$)
 4:         REMOVE_RELEVANT($osw, operand$)
 5:     else
 6:         $operand \leftarrow$ RANDOM($all\_operands$)
 7:         if $operand \in osw$ then
 8:             while HEAD($osw$) $\in operand$ do
 9:                 REMOVE_HEAD($osw$)
10:             end while
11:             MOVE_RELEVANT_TO_TAIL($osw, operand$)
12:         end if
13:     end if
14:     return $operand$
15: end procedure
16:
17: procedure SELECT_DEST_OPND($osw$)
18:     $operand \leftarrow$ RANDOM($all\_operands - osw$)
19:     if $operand \neq null$ then
20:         INSERT_RELEVANT_TO_TAIL($osw, operand$)
21:         return $operand$
22:     else
23:         return $error$
24:     end if
25: end procedure
```

Figure 1 shows an example of OSW, with the threshold of 4. The first instruction (MOV) selected *eax* as the source operand, while the OSW remained empty because *eax* was not in OSW (line 7 in the algorithm). *ecx* was appended to the OSW as a destination operand (line 20), and was split into fragments (*cl*, *ch* and *cxh*). In the following MUL instruction, *cx* stayed at the tail (line 11) as a source, while *ax* and *dx* were appended as destination, all in fragment format. OSW had more elements than the threshold when selecting source operands for the ADD instruction, so the first element *cxh* was selected (line 3). In this example, the ADD instruction needs 32-bit source operands, so *cl* and *ch* were also removed along with *cxh* (line 4). The number of element still exceeded the threshold, so *eax* was selected and removed similarly as source operand, but was appended back to the OSW immediately as the destination.

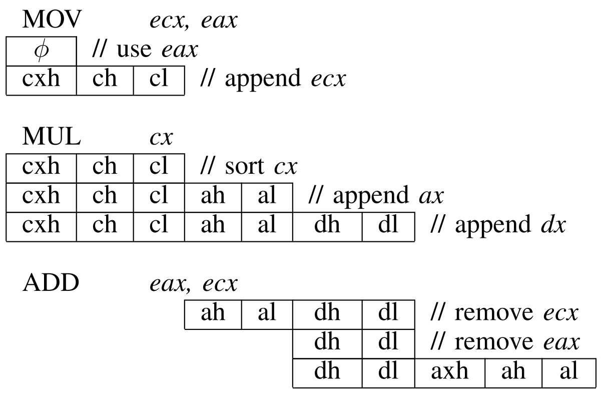

Fig. 1. An example of operand sliding window

In Figure 1, the OSW always extends to the right side, while shrinks from the left side. This is why the window is called *sliding*. It guarantees an operands to be used not long after it was defined. The threshold of OSW must not be larger than the number of registers, so that the oldest element can be used and removed before all registers come into the OSW. In our experiment, the threshold is equal to the number of registers. We use two OSWs for registers and memory locations, respectively. Memory OSW works in a similar way, with all memory operands limited within a 64-byte region.

C. Compilation

The test core is generated in text format. It is then compiled by GCC as inline assembly in the $main$ function, together with some input/output facilities. The input data is specified as the arguments of the binary executable. It contains the initial values of operands, and is set to registers and memory locations before the test core is executed. After the test core ends, the values of registers and memory locations are printed as output.

IV. REFINEMENT OF RAW TEST BINARIES

A bunch of randomly selected instructions, along with randomly selected input data, will probably trigger exceptions or generate undesired results. Therefore, we refined raw test binaries with static and dynamic methodologies. Static refinement can avoid certain exceptions by instruction wrapping, while a large amount of inputs are tested dynamically to distinguish those valid ones.

A. *Static Instruction Wrapping*

Exception might be triggered by some instruction with certain input. For example, a divide error (#DE) occurs when the divisor operand of instruction DIV is zero. Therefore, we conduct instruction wrapping for such instructions to ensure their validity. An instruction wrapping example is shown as follows.

$$\text{DIV } \textit{ebx} \quad \longrightarrow \quad \begin{array}{ll} \text{TEST} & \textit{ebx}, \$0 \\ \text{CMOVE} & \textit{ebx}, \$1 \\ \text{DIV} & \textit{ebx} \end{array}$$

Temporary operands used in the wrapping are selected from outside the OSW, and they do not enter the OSW although they are defined. It keeps the OSW the same as that without wrapping. In cases where the change of OSW is inevitable, this wrapping will be dropped. In our experiments, we logged related information during the wrapping generation, and checked the equality before and after wrapping.

B. *Dynamic Input Filter*

Instruction wrapping can avoid some exceptions but may result in code bloat, especially when checking NaN. Besides, it resets dangerous values to a pre-defined safe one, which weakens the test ability. Therefore, we wrapped just a few instructions like DIV, and left other checks to the dynamic input filter.

The number of possible input is quite large (8 32-bit registers and 64-byte memory), so we use an input generator to produce inputs. The value of each register or memory is either a 32-bit random number, or randomly selected from predefined boundary values such as 0x8000 or 0x7f7f7f7f.

A test binary is executed on source architecture, fed with generated inputs one by one. Invalid inputs will cause exceptions or undesired results, so they can be filtered out. Although such exceptions could be used to test signal handling modules in binary translator, we exclude them from the final test because EATBit's focus is on the translation modules, and signal handling can be tested with other specifically designed test suites.

Valid inputs are saved and will be used in the final test. The dynamic input filtering for a test binary stops when its valid inputs reach a threshold. Since larger test core usually makes an input more difficult to be valid, its threshold is smaller than that of a smaller test core. For some test binaries, most of the inputs are invalid because they have some improper instruction sequences. Such test binaries will be discarded if they did not collect enough valid inputs in given time.

V. Evaluation

A. *Experiment Setup*

The source ISA we use is IA-32. We generated and refined test binaries on a 48-way 1.86 GHz Intel Xeon server with 16GB memory, running Linux 2.6.

DigitalBridge [10] is used as the binary translator to be tested. It is a dynamic binary translator migrating x86 binaries to MIPS64 architecture. It supports static/dynamic translation and employs almost all common translation optimizations

TABLE II. Refinement for different core sizes

Group No.	Core Size	Refinement Time	Survival Rate
1	40	10 mins	100%
2	50	∼1 day	81.7%
3	60	∼2 days	52.0%
4	70	∼2.3 days	22.5%
5	140	∼3 days	19.9%

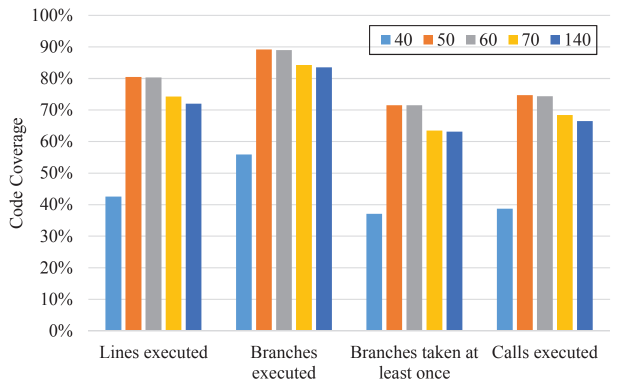

Fig. 2. Code coverage for different core sizes

on flag calculation, indirect control transfer, misaligned data access, etc.

Line, branch and function coverage are popular code coverage criteria in white-box testing [9]. Higher coverage means more test ability in some degree, at least for previously uncovered regions. Therefore, these indicators are our focus, and are used to evaluate different test approaches in our experiment. We instrumented DigitalBridge by GCOV [19], and collected the reported code coverage after each execution, including *lines executed*, *branches executed*, *taken at least once*, and *calls executed*.

B. *Core Size*

Core size of a test binary is an important factor for testing ability. A larger test core may cover more code lines and paths, but is harder to collect enough valid inputs in short time, time, and is also harder to debug for testers. We set 5 different core sizes and tested their impact. The largest core size is 140, which exceeds the limit of a basic block set by DigitalBridge. We generated and refined 1000 raw test binaries for each group with the methods introduced in Section III and IV. The survival rate of raw test binaries is shown in Table II. We found that a larger core size always needs longer dynamic filtering time and has a lower rate to survive.

Test binaries that have survived are used to test DigitalBridge. Figure 2 shows the code coverage of the 5 groups. A line, branch, or call is covered if it is executed at least once. The 2nd group (core size 50) achieved the best code coverage, slightly better than the 3rd group. This is reasonable since too large test core has lower survival rates, so that less test binaries become available for the final test, as shown in Table II.

We also compared the code coverage in each translation modules of DigitalBridge, using the test binaries from the 2nd group (BEST) and all the 5 groups (ALL). The results are shown in Table III. We found that the 2nd group alone is almost as good as all the groups. Therefore, we take 50 as the core size of test binaries in later experiments.

TABLE III. CODE COVERAGE FOR THE 2ND AND ALL GROUPS

Module	Group	Lines	Branches	Taken	Calls
Arith	**2nd**	60.83%	85.71%	73.81%	58.01%
	All	61.67%	85.71%	76.19%	58.36%
Arich2	**2nd**	88.91%	93.75%	76.88%	72.63%
	All	88.91%	93.75%	77.50%	72.63%
Logical	**2nd**	55.95%	73.68%	47.37%	54.00%
	All	56.49%	73.68%	52.63%	54.00%
Logical2	**2nd**	94.15%	94.95%	86.87%	91.52%
	All	94.15%	94.95%	86.87%	91.52%
Flag	**2nd**	90.77%	93.43%	79.81%	85.22%
	All	90.94%	93.43%	80.75%	85.37%
Pattern	**2nd**	89.97%	98.35%	80.58%	62.24%
	All	90.83%	98.35%	80.99%	63.07%
Reduction	**2nd**	92.63%	100.00%	87.14%	86.15%
	All	92.63%	100.00%	87.14%	86.15%
Extension	**2nd**	69.44%	82.21%	63.19%	47.53%
	All	69.44%	82.21%	63.19%	47.53%

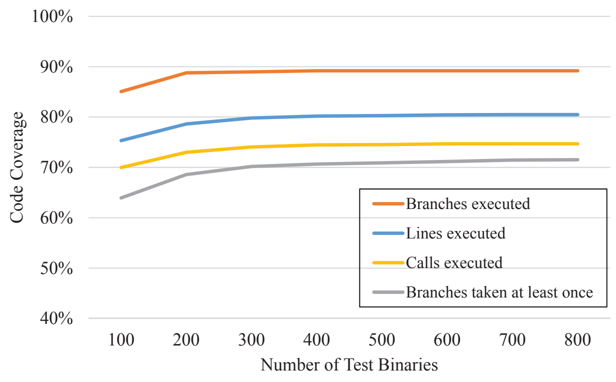

Fig. 3. Code coverage for different amount of test binaries

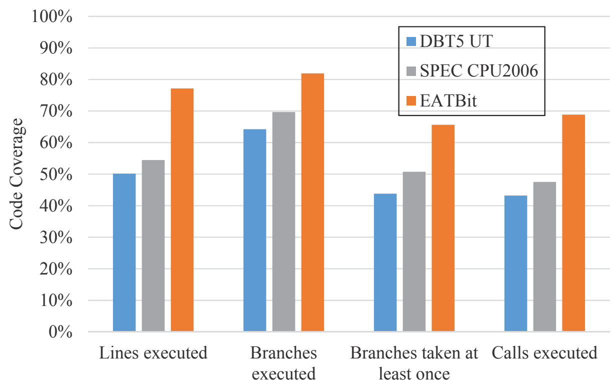

Fig. 4. Overall code coverage

TABLE IV. CODE COVERAGE IMPROVEMENT

Code Coverage		DBT5 UT	SPEC CPU2006	EATBit
Flag Pattern Optimization				
Lines Executed		77.36%	81.09%	89.97%
Branches	Executed	97.11%	98.76%	98.35%
	Taken$\geq$1	71.90%	74.79%	80.58%
Calls Executed		52.28%	56.43%	52.24%
Optimizations Based on History and Future				
Lines Executed		45.79%	50.48%	76.65%
Branches	Executed	59.20%	65.50%	80.27%
	Taken$\geq$1	39.66%	47.33%	64.37%
Calls Executed		42.09%	46.86%	71.67%
Overall Translations and Optimizations				
Lines Executed		50.16%	54.46%	77.16%
Branches	Executed	64.23%	69.66%	81.94%
	Taken$\geq$1	43.80%	50.74%	65.62%
Calls Executed		43.21%	47.53%	68.87%

C. *Quantity of Test Binaries*

More test binaries may bring more code coverage and expose more bugs, but too many of them can also be labor intensive and harvest little after reaching a certain point. Therefore, we test with various amount of test binaries ranging from 100 to 800, and the result is shown in Figure 3. More test binaries do not provide obvious code coverage improvement after the number of test binary exceeds 200.

D. *Overall Code Coverage*

We compared the overall cover coverage of EATBit with DBT5 UT and SPEC CPU2006. DBT5 UT is an x86 instruction test suite that we use during the development of DigitalBridge. In each test case, a single x86 instruction is fed with all kinds of typical boundary values, and the results are checked. There are 116, 50 and 168 test cases for integer, floating point and SIMD instructions respectively. We used the reference input set for SPEC CPU2006 runs. All these benchmarks were compiled by the Intel C++ Compiler 11.0, using the fast compiler optimization option.

Figure 4 presents code coverage of different test approaches. The test ability of DBT5 UT is limited due to the fixed context of the instruction to be tested. For example, the instruction is always followed by a PUSHF instruction, saving all flags to the stack for later examination. This would disable some flag optimization in binary translator, so that they could not be covered. SPEC CPU2006 has a better coverage, but the compiler gives the executable too much similarity among different test cases, so it still leaves more than 30% of code unexecuted. EATBit has the best result, which means a more sufficient test.

More detailed results are demonstrated in Table IV. Due to limited space, only two specific kinds of optimizations are shown. Optimizations based on future and history instructions include *Flag Reduction*, *Extension Propagation* and some other optimizations. Overall, the randomized approach can improve line coverage by 22.7% and branches taken by 14.9% compared with SPEC CPU2006. The improvement is 27.0% and 21.8% respectively compared with DBT5 UT.

EATBit also discovered some new bugs in DigitalBridge, which had passed the test of DBT5 UT and SPEC CPU2006. In our experiment, we generated 3468 test binaries. DigitalBridge triggered 4 different assertion errors with 485 of them, and produced wrong result with 350 of them. Since bugs reside in small test cores, not in complicated real-world programs, fixing them is relatively easy by comparing the execution traces of binary translator and real machine.

VI. LIMITATIONS OF THE APPROACH AND FUTURE WORK

There are some limitations in the current version of EATBit, and one of them is the selection of memory operands. Unlike registers, random memory operands would probably access unexpected locations, such as unprivileged region (which makes the program fail) or uninitialized region in the heap or stack (which makes final results uncertain). Therefore, in our current implementation, the memory locations are limited in a statically allocated 64-byte region, and register indirect addressing mode is not allowed in test cores. This weakens the test ability of EATBit as some instructions and operand formats cannot be tested. We are planning to improve it in the next version.

Control flow instructions are also not fully tested in EATBit because the test core does not contain such instructions. One reason is that a controlled jump or branch makes no difference from those outside test cores, while a random one has little chance to be valid. Besides, bugs in such instructions usually does not rely on their operand values, unlike arithmetic or logical instructions. Therefore, we exclude such instructions from test cores.

In the current implementation, the threshold of operand sliding window is hand-tuned, equal to the number of registers to make a full use of all register operands. A different threshold will affect the final code coverage, so we will analyze their relation and try auto-tuning in the future.

Existing random approaches [14][15][16][17] for compiler testing are mostly on source level. However, an operation in the source programming language is converted to intermediate representation (IR) with a few typical patterns, and compiler back-end has been familiar and reliable for such patterns. Therefore, we believe that randomly generated IR may be useful to test compiler back-ends, but a few issues should be addressed before that, such as reconstructing the high-level information needed.

VII. Conclusions

This paper proposed a novel randomized approach, EATBit, to generate and refine effective test binaries for automated testing of binary translation. EATBit defined some typical instruction categories and algorithms for testers to easily customize their own test binaries. Besides, an operand sliding window mechanism is also adopted to make a test binary compact and efficient for testing. To avoid potential instruction exceptions inside raw test binaries, we conducted instruction wrapping and input filter to refine the raw binaries. We also analyzed the impact of test core size and quantity on code coverage. Experimental results show that EATBit can successfully improve code coverage by more than 20% compared with SPEC CPU2006 and DBT5 UT. The approach also exposed some new bugs in the binary translator.

Acknowledgment

This paper is supported in part by the National Natural Science Foundation of China (NSFC) under the grant 60925009, 61332009, 61303051, 61303052, 61303053 and 61100011, the National High Technology Research and Development Program of China (2012AA010901), and the Innovation Research Group of NSFC (61221062).

References

[1] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham, "Dynamic helper threaded prefetching on the sun ultrasparc cmp processor," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 93–104.

[2] J. Lu, H. Chen, P.-C. Yew, and W. chung Hsu, "Design and implementation of a lightweight dynamic optimization system," *Journal of Instruction-Level Parallelism*, vol. 6, p. 2004, 2004.

[3] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[4] P. P. Bungale and C.-K. Luk, "Pinos: A programmable framework for whole-system dynamic instrumentation," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 137–147.

[5] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *Proceedings of the International Conference on Dependable Systems and Networks*, ser. DSN '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 269–280.

[6] "Spec cpu2006." [Online]. Available: http://www.spec.org/osg/cpu2006/

[7] "Supertest test and validation suite." [Online]. Available: http://www.ace.nl/compiler/ supertest.html

[8] "Linux test project." [Online]. Available: http://ltp.sourceforge.net/

[9] K. Burr and W. Young, "Combinatorial test techniques: Table-based automation, test generation and code coverage," in *Proc. of the Intl. Conf. on Software Testing Analysis & Review*. Citeseer, 1998.

[10] J. Li, C. Wu, and W.-C. Hsu, "An evaluation of misaligned data access handling mechanisms in dynamic binary translation systems," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 180–189.

[11] R. L. Sauder, "A general test data generator for cobol," in *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, ser. AIEE-IRE '62 (Spring). New York, NY, USA: ACM, 1962, pp. 317–323.

[12] K. V. Hanford, "Automatic generation of test cases," *IBM Syst. J.*, vol. 9, no. 4, pp. 242–257, Dec. 1970.

[13] C. J. Burgess and M. Saidi, "The automatic generation of test cases for optimizing fortran compilers," *Information and Software Technology*, vol. 38, no. 2, pp. 111–119, 1996.

[14] C. Lindig, "Random testing of c calling conventions," in *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, ser. AADEBUG'05. New York, NY, USA: ACM, 2005, pp. 3–12.

[15] F. Sheridan, "Practical testing of a c99 compiler using output comparison," *Softw. Pract. Exper.*, vol. 37, no. 14, pp. 1475–1488, Nov. 2007.

[16] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang, "Automated test program generation for an industrial optimizing compiler," in *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, 2009, pp. 36–43.

[17] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294.

[18] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 335–346.

[19] "Gcov, a test coverage program." [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html