Dynamic Verification of Approximate Computing Circuits using Coverage-based Grey-box Fuzzing

Kazuki Yoshisue
Nagoya University
E-mail:yoshi@ertl.jp

Yutaka Masuda
Nagoya University, JST PRESTO
E-mail:masuda@ertl.jp

Tohru Ishihara
Nagoya University

***Abstract*—Approximate computing (AC) has recently emerged as a promising approach to the energy-efficient design of digital systems. For realizing the practical AC design, we need to verify whether the designed circuit can operate correctly under various operating conditions. Namely, the verification needs to efficiently find fatal logic errors or timing errors that violate the constraint of computational quality. This paper proposes a novel dynamic verification methodology of the AC circuit. The key idea of the proposed methodology is to incorporate a quality assessment capability into the Coverage-based Grey-box Fuzzing (CGF). CGF is one of the most promising techniques in the research field of software security testing. By repeating (1) mutation of test patterns, (2) execution of the program under test (PUT), and (3) aggregation of coverage information and feedback to the next test pattern generation, CGF can explore the verification space quickly and automatically. On the other hand, CGF originally cannot consider the computational quality by itself. For overcoming this quality unawareness in CGF, the proposed methodology additionally embeds the Design Under Test (DUT) mechanisms into the calculation part of computational quality. Thanks to the integration of CGF and DUT mechanism, the proposed framework realizes the quality-aware feedback loop in CGF and thus quickly enhances the verification coverage for test patterns that violate the quality constraint. In this work, we quantitatively compared the verification coverage of the approximate arithmetic circuits between the proposed methodology and the random test. In a case study of an approximate multiply-accumulate (MAC) unit, we experimentally confirmed that the proposed methodology achieves the target coverage three times faster than the random test.**

***Index Terms*—Approximate Computing (AC), Coverage-based Grey-box Fuzzing (CGF), Design Under Test (DUT) mechanism, verification, computational quality**

I. Introduction

Approximate computing has recently emerged as a promising approach to the energy-efficient design of digital systems [1]–[7]. While the conventional systems require exact and completely deterministic computation, approximate computing allows some loss of quality or optimality in the computed result. This concept is suitable for a wide range of applications such as digital signal processing, image, audio, video processing, graphics, wireless communications, and machine learning. By exploiting the inherent resilience of those applications, approximate computing techniques substantially improve energy efficiency (e.g. [1]).

After designing the AC circuit, we need to verify whether the designed circuit can operate correctly under various operating conditions. Here, the fundamental assumption for the verification is completely different between the conventional circuit and the AC circuit. For example, in the conventional circuit, all paths except for false paths should not cause logic and timing errors. On the other hand, the AC circuit gives a constraint in the quality of computational results, e.g., inference accuracy in the machine learning domain, and allows the occurrence of errors as long as the circuit satisfies the target constraint. Namely, in the verification of AC design, we need to find fatal logic errors or timing errors that violate the constraint of computational quality. Originating from the difference of target errors in the verification, the verification technique for the AC circuit is still immature compared with the AC design methodology. Therefore, a novel quality verification methodology for AC circuits is strongly desired.

Quality verification methodologies of the AC circuit can be divided into static and dynamic approaches. As a static approach, formal verification based techniques are proposed in recent years [8]–[10]. These techniques solve the satisfiability (SAT) problems and thus find a correction of errors that violate quality constraints. On the other hand, a dynamic verification technique prepares the test patterns and gives them to the target circuit. After running the test patterns on the circuit, we can obtain the computational results and judge whether the obtained results are acceptable or not. Since the dynamic approach is straightforward [11] and easy-to-handle, the dynamic verification through the simulation is one of the most widely deployed approaches [12]. However, in the dynamic approach, the verified hardware components heavily depend on the input test pattern. From this point of view, the efficient generation of test patterns is crucially important.

Very recently, for improving the test coverage efficiently, Coverage-guided Grey-box Fuzzing (CGF) has been actively developed in the research field of software security testing [13]–[15]. CGF is one of the testing technique called fuzzing [16]–[22]. CGF automatically generates and executes test patterns that enhance verification coverage by repeatedly executing (1) mutation of test patterns, (2) execution of the program under test (PUT), and (3) tabulating code coverage and feeding back to the next test pattern generation. CGF improves the code coverage rapidly by keeping mutation and coverage aggregation lightweight [16]. This is quite attractive not only for software testing but also for hardware verification

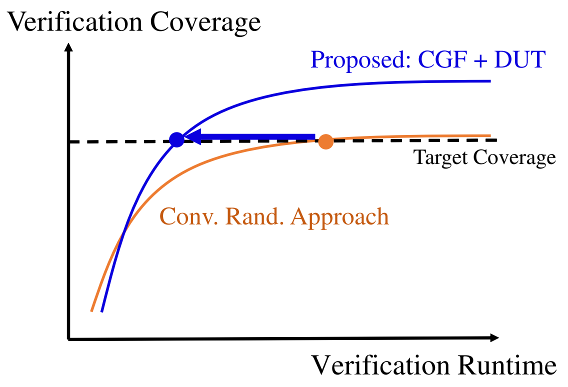

Fig. 1: Expected speed-up effects thanks to the proposed verification framework.

[23].

This paper proposes a novel dynamic verification methodology of the AC circuit. The key idea of the proposed methodology is to incorporate a quality assessment capability into the CGF. We found that CGF originally cannot consider the computational quality by itself. In this case, the mutation could not generate test patterns efficiently that violate the constraint. For overcoming this quality unawareness in CGF, the proposed methodology additionally embeds the Design Under Test (DUT) mechanisms into the calculation part of computational quality. Thanks to the integration of CGF and DUT mechanism, the proposed framework realizes the quality-aware feedback loop in CGF and thus quickly enhances the verification coverage for test patterns that violate the quality constraint.

Figure 1 illustrates the expected coverage improvement thanks to the proposed methodology, i.e., a blue curve. As a baseline, we plotted the random stimulus approach with an orange curve, which is typically used in the dynamic verification domain [12]. When we start the verification, the random test may improve the coverage steadily since a number of *easy-to-activate* paths are unexplored yet. Then, if we continue to run the verification, the random test gradually decelerates since the random approach suffers from finding paths that rarely activate or scarcely affect the computational quality. On the other hand, the proposed approach incorporates the coverage-aware feedback loop into the test pattern generation. Besides, we take into account the computational quality with the DUT mechanisms for the AC verification. Thanks to the CGF and DUT, the proposed framework is expected to accelerate the verification speed compared with the typical random testing.

The main contributions of this work include (1) the verification methodology of AC circuits using CGF and (2) the quality-aware feedback framework thanks to the DUT mechanism. To the best of our knowledge, this is the first work that proposes the dynamic verification framework for AC circuits with CGF. Moreover, for efficiently finding test patterns that violate quality constraints, the proposed methodology utilizes the DUT mechanism. Experimental results show that the proposed methodology contributes to finding logic paths with a low activation probability and thus achieves the target coverage three times faster than the simple random test.

The rest of this paper is organized as follows. Section II describes the assumed verification of AC design and CGF and highlights the challenges for applying CGF to AC circuit verification. Section III proposes a quality verification methodology of AC circuits using the CGF and DUT mechanisms. Section IV compares the coverage between the proposed framework and the random test. Then, this section discusses the speed-up effects of coverage improvement thanks to the proposed methodology in terms of DUT mechanisms. Finally, the concluding remarks are given in Sect. V.

II. Assumption and Challenges

This section explains the assumed verification of AC circuit and CGF and summarizes the challenges for applying CGF to the AC verification. Section II-A highlights the difference between the verification of the AC circuit and the conventional circuit. Then, Section II-A introduces assumed CGF. Section II-B discusses the challenges of CGF for verifying the AC design.

A. Assumed Verification of AC Design and Fuzzing

First, let us introduce the fundamental difference of the verification concept between the conventional circuit and AC circuit using Fig. 2. In this work, we assume the dynamic verification where the computational result can be observed, and the constraint of computational quality is given.

In Fig. 2, three input signals InA, InB, and InC are given to the adder, and one output signal Y can be observed. Let us suppose the values of three input signals are $1001_{(2)}$, $0001_{(2)}$, and $0_{(2)}$. Also, in this example, the quality constraint is set to the error of Y by less than 15%. In the conventional design, any logic errors and timing errors cannot be allowed, as shown in Fig. 2(a). Hence, the verification goal is to find logic and timing errors as much as possible. On the other hand, in the AC design, the error occurrence is allowed as long as the circuit satisfies the quality constraint. For example, in

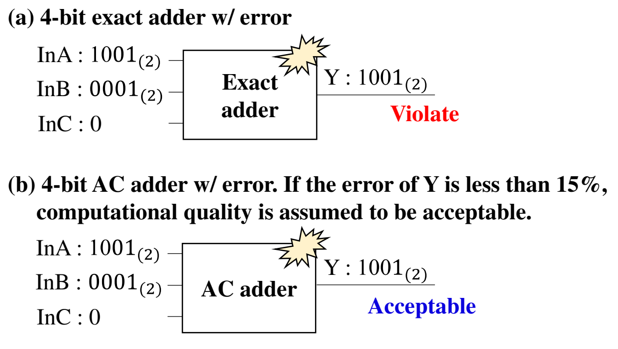
(a) 4-bit exact adder w/ error
InA : 1001(2)
InB : 0001(2)
InC : 0
Exact adder
Y : 1001(2)
Violate
(b) 4-bit AC adder w/ error. If the error of Y is less than 15%, computational quality is assumed to be acceptable.
InA : 1001(2)
InB : 0001(2)
InC : 0
AC adder
Y : 1001(2)
Acceptable

Fig. 2: (a) In the conventional circuit, any error occurrence is prohibited. (b) In the AC design, the error occurrence is allowed as long as the computational quality is acceptable.

Fig. 2(b), Y has the error of 10% (=$\frac{1010_{(2)}-1001_{(2)}}{1010_{(2)}}$), but this error is smaller than 15% and thus acceptable. From the above, the AC verification aims to identify errors that violate the quality constraint, which is totally different from conventional verification.

Next, this section explains the assumed CGF using Fig. 3. In CGF, the test target space is automatically searched by repeating (1) test pattern mutation, (2) PUT execution, and (3) coverage-wise feedback to the next test pattern generation [13]–[15]. As for the mutation, various strategies are developed for exploring the target space efficiently, e.g., bit-flip, byte-flip, arithmetic operation, havoc, and random operations in [13]. In the PUT execution, the activated paths can be measured via the instrumentation codes, which are inserted via the source code compile. When new paths are activated in the PUT execution, CGF adds the current input to the queue as an interesting seed for the mutation. Thanks to the lightweight mutation and coverage feedback loop, CGF improves the code coverage rapidly [16], which is quite attractive for the verification. Here, since CGF has been actively developed in the research field of software security testing [13]–[15], most of the CGF methodologies are developed for high-level languages like C language. Based on this background, this work assumes to verify the AC design starting from the C-flavored hardware description language (HDL), e.g., SystemC, CUDA C, and OpenCL.

B. Challenges of CGF for Verifying the AC Design

As mentioned in the previous section, CGF measures the code coverage through the PUT execution. When the coverage improves, the current input pattern is incorporated into the mutation. Therefore, the heart of CGF lies in the coverage assessment and feedback on test patterns. From this point of view, if we apply CGF to the verification of AC circuits, CGF needs to evaluate the code coverage of the AC circuit, taking into account the computation quality. However, we found that CGF originally cannot consider the computational quality by itself, which will be explained using Listing 1 as an example.

Listing 1: Code example of AC adder written in SystemC

```
1 int sc_main (int argc, char *argv[]) {
2   sc_signal<bool> InA, InB, InC;
3   sc_signal<bool> Y, OutC;
4   AC_Full_Adder *AC_FA1;
5   AC_FA1 = new AC_Full_Adder ("AC_FA1");
6   (*AC_FA1)(InA, InB, InC, Y, OutC);
7
8   return 0;
9 }
```

Listing 1 shows a code example of an AC adder written in SystemC. Lines 2 and 3 are the input and output declaration. Lines 4 to 6 correspond to the instantiation of the AC adder module. For simplicity, the content of AC adder module is omitted in the example. From Listing 1, we can see that the HDL of pure AC adder does not include any mechanism to evaluate how much calculation error occurs at output, e.g., Y, Cout. If CGF fails to acquire the violation condition of quality constraint, the target coverage information cannot be corrected appropriately. In this case, the mutation could not generate test patterns efficiently that violate the constraint. On the other hand, in the verification of AC circuit, we need to find test patterns that violate the computational quality constraint as discussed in Sect. II-A. Therefore, for applying CGF to the quality verification of AC circuits, the countermeasure for evaluating the quality-aware code coverage is crucially important.

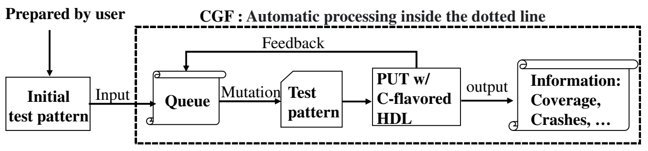

Fig. 3: Assumed CGF for C-flavored HDL. CGF automatically searches the target space by repeating (1) test pattern mutation, (2) PUT execution, and (3) feedback to the next test pattern.

III. Proposed Verification Framework

In this section, we propose a quality verification framework for AC circuits using CGF. The key idea of the proposed framework is to incorporate the quality assessment capability into the CGF. The proposed methodology embeds the DUT mechanism into the quality calculation part of HDL and thus enables CGF to evaluate whether the quality constraint is satisfied or not. Thanks to the integration of CGF and DUT mechanism, the proposed framework realizes the quality-aware feedback loop in CGF and thus quickly enhances the verification coverage for test patterns that violate the quality constraint.

Figure 4 shows the overview of the proposed framework. As inputs, the proposed framework receives the hardware described in C-flavored HDL and the target computational quality. First, the proposed framework inserts the DUT mechanism into the received HDL. Next, the HDL with the embedded DUT mechanism is passed through the CGF. In the CGF, PUT is performed, and then the coverage is derived taking into account the target quality. Finally, the proposed framework outputs the results of CGF, such as the coverage, the number of crashes, and test patterns that violate the constraint. The following subsection describes the detail of DUT mechanism.

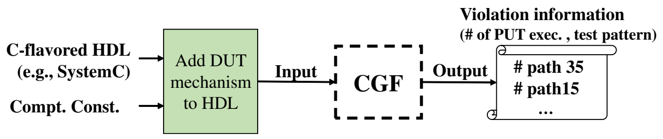

Fig. 4: The overview of the proposed framework. The proposed framework incorporates the quality assessment capability into the CGF via the Design Under Test (DUT) mechanism.

A. DUT Mechanism for CGF

The proposed DUT mechanism consists of two key components; (1) the bridge between the test cases given from CGF and the input test patterns for HDL, (2) the classification of the test pattern with the computational quality. This section explains the implementation and contribution of DUT mechanism using an example of Listing 2, which adds the DUT mechanism to the pure AC adder, i.e., Listing 1.

Listing 2: Code example of AC adder with DUT

```
1  int sc_main (int argc, char *argv[]) {
2
3    char buf[8];
4    if (read(0, buf, 8) < 1) { exit(1);}
5    int value = *(int*) &buf[0];
6    input = value%(1<<n);
7    int a, b, c = 0;
8
9    //Update a, b, c referring to input.
10   TestPatternGenerate(input);
11
12   sc_signal<bool> InA, InB, InC;
13   sc_signal<bool> Y, Cout;
14   sc_signal<bool> EX_Y, EX_Cout;
15
16   AC_Full_Adder *AC_FA1;
17   AC_FA1 = new AC_Full_Adder ("AC_FA1");
18   (*AC_FA1)(InA, InB, InC, Y, Cout);
19
20   EX_Full_Adder *EX_FA1;
21   EX_FA1 = new EX_Full_Adder ("EX_FA1");
22   (*EX_FA1)(InA, InB, InC, EX_Y, EX_Cout);
23
24   InA.write(a); InB.write(b); InC.write(c);
25
26   if ( (Y - EX_Y) > ERROR_TH){
27     if (value%(1<<n) == 0){;}
28     else if (value%(1<<n) == 1){;}
29     ...
30
31     // When the quality is violated,
32     // CGF records using the crash.
33     abort();
34   }
35
36   return 0;
37 }
```

First, we explain the bridge part between the mutated test cases and the input pattern to HDL. We highlight that this bridge is the essential part for constructing the feedback loop, as shown in Fig. 3. Let us see the example of Listing 2. First, the bridge part saves the input character string from CGF (lines 3 and 4). Then, the stored string is converted to an integer type (lines 5 to 7). After that, in line 10, the test pattern to the adder is updated by referring to the integer, e.g., using the pre-defined look-up table. Finally, the test pattern can be successfully given to the input of adders in line 24. In summary, thanks to the bridge, the mutation by CGF can be reflected directly on the input test patterns to HDL.

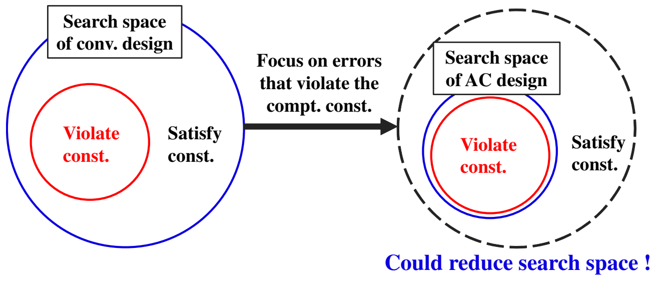
Search space of conv. design
Violate const.
Satisfy const.
Focus on errors that violate the compt. const.
Search space of AC design
Violate const.
Satisfy const.
Could reduce search space !

Fig. 5: By focusing on errors that violate the computational quality, the search space and thus the required runtime in the verification could be reduced.

Next, the classification part of the test pattern is discussed. In the proposed framework, the accurate circuit component is first prepared for calculating the approximation error (Lines 14 and 20 to 22). Then, the computational results in the AC component are compared with the accurate result for deriving the error (Line 26). When the error exceeds the given threshold value, the DUT mechanism starts to identify the test pattern (Lines 27 to 33). More specifically, our framework inserts conditional branches of the input test patterns for recognizing the executed pattern (Lines 27 to 29). For example, in Listing 2, 8 (= 2^3) branches could be enumerated at most. Since the CGF marks each branch at the compile time and traces the executed branch in PUT phase, the above branch description enables the CGF to sum up the branch coverage that violates the computational quality. From the above, thanks to the computational quality evaluation and classification of test patterns, the proposed framework realizes the quality-aware feedback loop in CGF and thus the quality verification of AC circuit.

Here, another interesting consideration is that the quality-aware CGF may reduce the verification time. Let us explain the expectation with Fig. 5. As previously discussed with Sect. II-A, the AC verification aims to identify errors that violate the quality constraint. From this point of view, the total number of errors we need to find could be dramatically reduced compared with the conventional circuit. Consequently, the required time consumption for dynamic verification is expected to be saved significantly. Section IV-C discusses the speed-up effect thanks to the quality-aware verification.

IV. Experimental Evaluation

This section evaluates the trade-off relationship of the proposed framework between the number of PUT execution and the coverage. Section IV-A describes the evaluation setup. Section IV-B shows the speed-up effects thanks to the proposed methodology compared with the conventional random testing. Then, Section IV-C discusses the speed-up effects in terms of the path activation probability and the target error space for verification.

A. Evaluation Setup

In this work, we use the approximate multiplier-accumulator (MAC) unit, where the numbers of input bits and accumulation clock cycles are set to 3 and 5, respectively. As the AC technique, we select the bit-width scaling and then truncate the least-significant bit of input. This circuit was implemented with the SystemC language. Next, we add the DUT mechanism to the approximate MAC unit as explained in Sect. III-A. In this work, we add a quality constraint of the computational error in approximate MAC result. Namely, we used the following equation to derive the error and set the constraint of 0.3 for the error threshold, e.g., ERROR_TH in Listing 2.

$$error = \frac{|R_{ac} - R_{ex}|}{R_{ex}}, \tag{1}$$

where R_{ac} represents the computational results of approximate MAC unit, and R_{ex} is the golden results of exact MAC unit. Note that the above constraint is just an example, and the proposed framework can cope with other settings in the same manner.

Then, for taking into account the quality-aware coverage, the branch insertion is performed, e.g., lines 27 to 29 in Listing 2. Here, if we naively enumerate all the combinations of test patterns, the number of branch descriptions exponentially explodes with the number of input bits and execution clock cycles. For mitigating this issue, this work compresses the number of branch descriptions using the accumulated results. More specifically, our evaluation focuses on the pair of the accumulated results until the fourth clock cycles and the test pattern in the fifth clock cycle. In this case, the maximum number of enumeration can be dramatically reduced, e.g., from $2^{(3+3)\times 5} \simeq 10^9$ to $(7^2 \times 4) \times (2^{(3+3)}) \simeq 10^4$ in this experiment. Since this strategy is heuristic, developing a better compression approach is one of our future works.

As the CGF tool, we select AFL 2.52b [13] and incorporate the AFL in the proposed design. By running the PUT in CGF, the quality-aware branch coverage can be obtained. Note that other CGF tools [14], [15] can be similarly utilized in the proposed framework. As a comparison, we select a typical random approach as discussed with Fig. 1 in Sect. I, and implement the approach using the C++11 random number library. Since the implementation difference between the random approach and AFL may cause an unfair comparison, we do not directly compare the runtime but the number of PUT execution between the proposed and random approaches. Note that the upper bound of the number of PUT executions is set to 10 million times. Consequently, the trade-off relationship between the number of PUT executions and quality-aware branch coverage is quantitatively evaluated. In this evaluation, we use a computer machine equipped with Ubuntu 16.04 LTS and AMD Ryzen Threadripper 3990X 64-Core Processor.

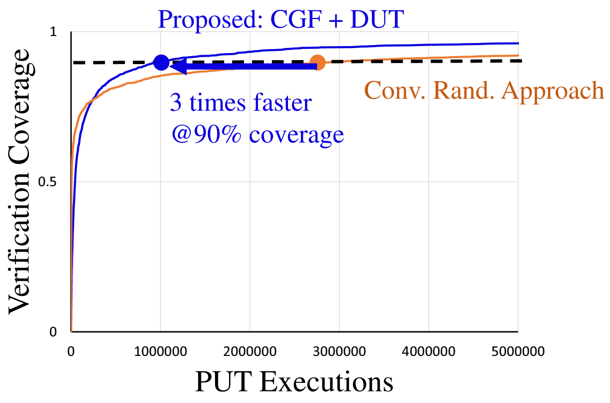

Fig. 6: Trade-off comparison results between the proposed and random approach. The proposed methodology achieves the 90% coverage three times faster than the random approach.

B. Evaluation Results

Figure 6 shows the comparison results of the trade-off relationship of the number of PUT executions and coverage between the random test and the proposed methodology. From Fig. 6, we can see two interesting observations. The first finding is that the random test improves the coverage well at the beginning of verification. For example, when we look at the line where the coverage equals 50%, the random test repeats the PUT executions 230 times while the proposed approach requires 1,200 times. This result leads us to the consideration that the random test has enough capability to find *easy-to-activate* errors, which will be further discussed in Sect. IV-C.

On the other hand, when we continue to run the PUT, we can observe the second tendency. That is, the proposed methodology improves the coverage steadily, whereas the random test rapidly decelerates. For example, if we focus on the 90% coverage line in Fig. 6, the proposed framework takes the PUT execution 1.0×10^6 times while the random approach requires 3.0×10^6 times. In other words, at this coverage point, the proposed approach is three times faster than the random approach. From these results, we experimentally confirm that the proposed methodology finally achieves the significant speed-up effects compared with the typical random approach.

C. Discussion

The evaluation results in Sect. IV-B showed that the proposed methodology achieves the speed-up effects. Let us investigate the results in detail.

First, we examine the coverage improvement by the proposed and random approaches in terms of the path activation probability. As previously explained with Sect. III-A, the test pattern to the AC circuit is determined by the strings from CGF, e.g., lines 3 to 10 in Listing 2. Therefore, by modifying the mapping function from the strings to test patterns, we can control the average activation probability. Based on this consideration, We prepare the six mapping functions where the input probability of logical value 1 in each input bit is

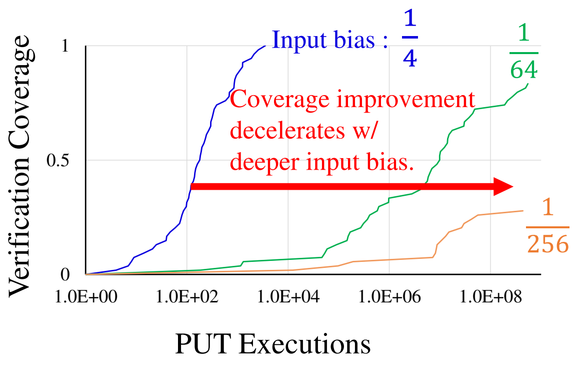

Fig. 7: The trade-off relationship of random test with different input bias settings. The smaller value means the deeper bias, i.e., the logic value 1 is unlikely to be inputted. In random test, the coverage improvement significantly decelerates with the deeper input bias.

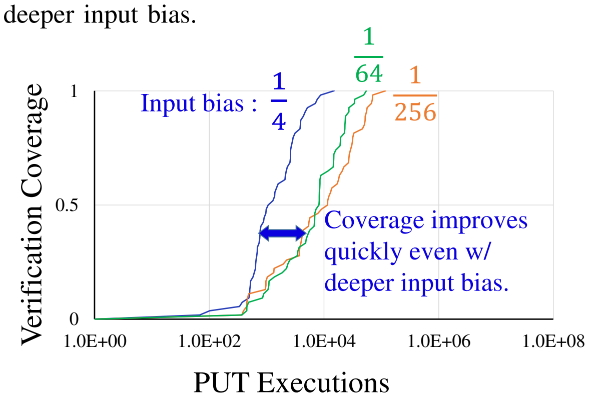

Fig. 8: The trade-off relationship of the proposed framework with different input bias settings. In the proposed framework, the coverage improves quickly even with the deeper input bias.

biased to $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{64}$, $\frac{1}{128}$, or $\frac{1}{256}$. Then, with these mapping functions, we evaluate the trade-off relationship between the number of PUT executions and coverage with the random test and the proposed methodology. In this evaluation, we use an approximate adder, which is family of adders in the literature of [24].

Figures 7 and 8 show the evaluation results of random test and proposed framework, respectively. For simplicity, both figures plot the three results with the bias of $\frac{1}{4}$, $\frac{1}{64}$, and $\frac{1}{256}$. From Fig. 7, we can see that the coverage improvement in the random approach decelerates significantly when the input bias becomes deeper. For example, when we look at the 40% coverage line, the numbers of PUT executions are 135 with the bias of $\frac{1}{4}$ but 5.7×10^6 with the bias of $\frac{1}{64}$. On the contrary, from Fig. 8, the proposed methodology significantly mitigates the increase of PUT execution even with the deeper input bias. For example, with the bias of $\frac{1}{4}$ and $\frac{1}{64}$, the required PUT executions at the 40% coverage are 800 and 5,400, respectively. These results clearly indicate that the proposed methodology achieves a better trade-off relationship in a circuit system, including a path with a low path activation probability.

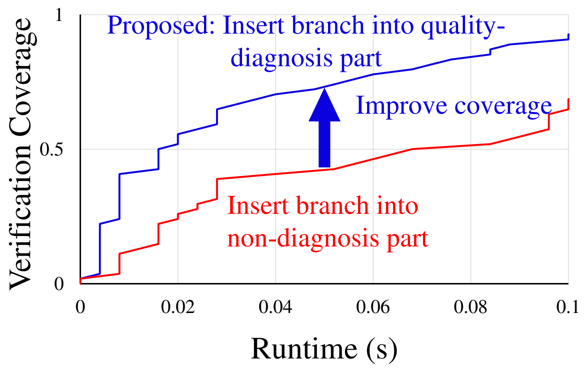

Fig. 9: Coverage improvement thanks to the branch insertion into the diagnosis part of computational quality.

Lastly, we investigate the effectiveness of DUT mechanisms in terms of the branch insertion strategy. As previously discussed with Fig. 5 in Sect. III-A, the required time consumption for dynamic verification is expected to be reduced by focusing on errors that violate the quality constraint. Based on this expectation, we implement the DUT mechanism, which inserts the branch into non-diagnosis part of computational quality, e.g., line 25 in Listing 2. Then, we compared the trade-off relationship between the runtime and coverage for discussing the effectiveness of branch insertion into the diagnosis part, e.g., lines 27 to 29 in Listing 2. Note that the identical CGF is utilized in the comparison. Figure 9 shows the comparison result. From Fig. 9, we can see that the proposed framework achieves a better trade-off between the runtime and coverage. For example, at the runtime of 0.05 seconds, the proposed improves the coverage from 43% to 74% by 31%. Therefore, we experimentally confirm that the proposed framework improves the coverage efficiently thanks to the quality-aware DUT mechanism.

V. Conclusion

This paper proposed the novel dynamic verification methodology of the AC circuit. The key idea of the proposed methodology is to incorporate the quality assessment capability into the CGF via the DUT mechanism. Thanks to the integration of CGF and DUT mechanism, the proposed framework realizes the quality-aware feedback loop in CGF and thus quickly enhances the verification coverage for test patterns that violate the quality constraint. In this work, we quantitatively compared the verification coverage of the approximate arithmetic circuits between the proposed methodology and the random test. In a case study of the approximate MAC unit, we experimentally confirmed that the proposed methodology improves the coverage by three times faster than the random test.

Acknowledgement

This work was partially supported by JSPS KAKENHI Grant Number JP20K19767 and JST, PRESTO Grant Number JPMJPR20M9, Japan.

References

[1] J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design,” *Proc. ETS*, pp. 1-6, 2013.

[2] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, “Analysis and characterization of inherent application Resilience for Approximate Computing,” *Proc. DAC*, pp. 1-9, 2013.

[3] Q. Xu, T. Mytkowicz, and N. S. Kim, “Approximate computing: A survey,” *IEEE Design & Test*, vol. 33, no. 1, pp. 8-22, 2016.

[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Architecture support for disciplined approximate programming,” *Proc. ASPLOS*, pp. 301-312, 2012.

[5] R. Hegde and N. R. Shanbhag, ”Soft digital signal processing,” *IEEE TVLSI*, vol. 9, no. 6, pp. 813-823, 2001.

[6] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, “Slack redistribution for graceful degradation under voltage overscaling,” *Proc. ASP-DAC*, pp. 825-831, 2010.

[7] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, “Low-power digital signal processing using approximate adders,” *IEEE TCAD*, vol. 32, no. 1, pp. 124-137, 2013.

[8] S. Froehlich, D. Große, and R. Drechsler, “One method–all error-metrics: A three-stage approach for error-metric evaluation in approximate computing,” *Proc. DATE*, pp. 284–287, 2019.

[9] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, “Precise error determination of approximated components in sequential circuits with model checking,“ *Proc. DAC*, pp. 1–6, 2016.

[10] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, “MACACO: Modeling and analysis of circuits for approximate computing,” *Proc. ICCAD*, pp. 667–673, 2011.

[11] A. Bosio et al., “Design, Verification, Test and In-Field Implications of Approximate Computing Systems,” *Proc. ETS*, pp. 1-10, 2020.

[12] M. Zhou, W. N. N. Hung, X. Song, M. Gu, and J. Sun, “Temporal coverage analysis for dynamic verification,” *IEEE TCAS-II*, vol. 65, no. 1, pp. 66-70, 2018.

[13] M. Zalewski, “American Fuzzy Lop,” http://lcamtuf.coredump.cx/afl/.

[14] C. Lemieux and K. Sen, “FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage,” *Proc. ASE*, pp. 475-485, 2018.

[15] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “VUzzer: Application-aware evolutionary fuzzing,” *Proc. NDSS*, 2017.

[16] V. J. M. Manès et al., “The art, science, and engineering of fuzzing: A survey,” *IEEE TSE*, Oct. 2019.

[17] H. M. Le, D. Groβe, N. Bruns, and R. Drechsler, “Detection of hardware trojans in SystemC HLS designs via coverage-guided fuzzing,” *Proc. DATE*, pp. 602-605, 2019.

[18] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[19] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, “REDQUEEN: Fuzzing with input-to-state correspondence,” *Proc. NDSS*, 2019.

[20] A. Takanen, J. D. DeMott, and C. Miller, Fuzzing for software security testing and quality assurance. Artech House, 2008.

[21] J. E. Forrester and B. P. Miller, “An empirical study of the robustness of Windows NT applications using random testing,” . *Proc. USEC*, Aug. 2000.

[22] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” *Proc. NDSS*, pp. 151–166, 2008.

[23] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, “RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs,” *Proc. ICCAD*, pp. 1-8, 2018.

[24] C. Liu, J. Han, and F. Lombardi, “A low-power, high-performance approximate multiplier with configurable partial error recovery,” *Proc. DATE*, pp. 1-4, 2014.