Vol.2020-SLDM-192 No.27
2020/11/18

ファジングを用いた近似コンピューティング回路の品質検証手法の一検討

吉末 和樹[1]　増田 豊[2]　石原 亨[2]

概要：本研究では，計算品質の制約を阻害するテストパターンの探索に根差した，近似コンピューティング 回路の検証技術を提案する．提案手法の肝は，Coverage-guided Greybox Fuzzing (CGF) と Design Under Test (DUT) 機構の活用にある．CGF により，(1) テストパターンの変異，(2)Program Under Test の実行，(3) カバレッジの集計とテストパターンへのフィードバック，を繰り返し行うことで，テスト対象空間を効率的に自動探索する．また，ハードウェア記述言語内の品質計算箇所に DUT 機構を埋め込み，信号伝搬情報を詳細化することで，計算品質の制約を診断しつつ，制約を違反するテストパターンに対する検証網羅性を評価する．近似演算器を対象とした評価実験により，検証時間とカバレッジの関係から，提案手法とランダムテストの優劣を議論する．

1. 序論

ポストムーア時代において，集積システムの省電力化と高性能化を推進可能な設計技術として，近似コンピューティング (Approximate Computing; AC) に注目が集まっている [1], [2], [3], [4], [5], [6], [7], [8]．AC は，重要な計算を正確に実行し，他の演算を近似的に実行する設計パラダイムである．冗長な計算を内包するアプリケーションと特に親和性が高く [1]，機械学習，ディジタル信号処理，画像処理，音声処理などの多様な分野において，省エネルギー化を促進できると高く期待されている．この期待から，現在までに，上流のソフトウェアレベル (例. プログラミング言語 [5]) から下流のハードウェアレベル (例. アーキテクチャ [6]，タイミング特性 [7]，トランジスタレベル [8]) に渡って，様々な AC 設計手法が研究されている．

設計後の AC 回路は，信頼性保証のために，多様なテストパターンの元で正しく動作するか検証される必要がある．テストパターンの例として，機械学習の推論テストにおける推論用データセット，画像処理における入力画像などが挙げられる．ここで，従来回路と AC 回路では，検証時の制約が大きく異なる．例えば，従来回路では，全てのパスが論理故障とタイミング故障を起こさないことを制約とする．一方，AC 回路は，計算結果の品質 (例. 機械学習の推論精度，画像処理の PSNR など) を制約とし，制約を満足する範囲であれば，論理故障やタイミング故障の発生を許容する．従って，AC 回路の検証では，従来回路と異なり，「計算品質の制約を脅かすテストパターン」を明らかにする必要がある．

AC 回路の検証技術は，静的な手法と動的な手法に大別される．近年，静的な検証技術として，形式的検証を利用した手法 [9], [10], [11] が提案されている．静的な手法は，充足可能性問題などを解き，計算品質の制約を阻害する故障の組を発見する．しかし，回路規模と実行クロックサイクル数に応じて，計算時間が劇的に増加するため，検証速度のスケーラビリティに課題を持つ．他方，動的な検証技術として，論理シミュレーション [12] の利用が考えられる．動的な検証では，AC 設計の仕様に準拠するテストパターンを，回路に入力して実行することで，計算品質の制約を満足するか定量的に議論できる．例えば，画像処理では，入力画像を構成するピクセル情報を，テストパターンとして AC 回路に与えて，出力画像の画質 (PSNR や SSIM など) が許容可能か評価できる．一方，動的な手法では，検証箇所がテストパターンに依存するため，検証の網羅性に課題を持つ．以上をまとめると，既存技術では，検証の速度と網羅性を両立することが困難である．従って，限られた検証時間で AC 回路の信頼性を保証するためには，高速性と高い網羅性を兼ね備えた新しい検証技術が必要不可欠である．

ここで，ソフトウェア・セキュリティの研究領域に目を向けると，近年，Coverage-guided Greybox Fuzzing (CGF) と呼ばれるテスト技術が盛んに研究されている [13], [14], [15]．CGF は，ファジング [16], [17], [18], [19], [20], [21], [22] と

[1] 名古屋大学情報学部コンピュータ科学科
[2] 名古屋大学大学院情報学研究科

呼ばれるソフトウェアテスト技術の一種であり，(1) テストパターンの変異，(2) Program Under Test (PUT) の実行，(3) コードカバレッジの集計とテストパターンへのフィードバック，を繰り返し行うことで，検証網羅性を高めるテストパターンを自動生成し，実行する技術である．CGF は，変異とカバレッジの集計を軽量に抑えることで高速性を保ちつつ [15]，未知のバグに対する高い発見能力を達成しており [16]，ソフトウェア・システムの品質テストにおいて成功を収めている技術として，特に注目を集めている [14]．

本研究では，CGF の高速性と高い検証網羅性に着目し，CGF を AC 回路の品質検証に応用する．CGF では，PUT 実行時に新たなパスが活性化したか (カバレッジが向上したか) 計測し，カバレッジ向上時に現在の入力パターンを変異に取り入れることで，検証網羅性を高める．換言すれば，CGF の肝は，カバレッジの評価とテストパターンへのフィードバックにある．従って，CGF を AC 回路の品質検証に応用するためには，第一に，AC 回路の検証網羅性に則してカバレッジを評価する必要がある．第二に，計算品質を阻害するテストパターンを効率的に生成するために，PUT 実行後の計算品質をフィードバックし，制約違反の有無情報を変異に取り入れる必要がある．

本稿では，計算品質の制約を阻害するテストパターンの探索に根差した，AC 回路の検証技術を提案する．提案手法の肝は，CGF と Design Under Test (DUT) 機構の活用にある．CGF により，「テストパターンの変異，PUT の実行，カバレッジの集計とテストパターンへのフィードバック」を繰り返し行うことで，テスト対象空間を効率的に自動探索する．また，ハードウェア記述言語 (以下，HDL) 内の品質計算箇所に DUT 機構を埋め込み，信号伝搬情報を詳細化することで，計算品質の制約を診断しつつ，制約を違反するテストパターンに対する検証網羅性を評価する．

本研究の主な貢献は（1）CGF を用いた AC 回路の検証手法と（2）DUT 機構を利用した AC 回路のカバレッジ評価法にある．AC 回路の検証に対してファジングを利用した研究は，著者らの知る限り，本研究が初である．提案手法は，CGF と DUT を活用することで，計算品質の制約を違反するテストパターンを効率的に発見する．近似演算器を対象とした評価実験により，検証時間とカバレッジの関係から，提案手法とランダムテストの優劣を議論する．

本稿の以降の構成は以下の通りである．2 章では，関連研究としてファジングについて説明し，AC 回路に CGF を適用する際の課題を整理する．3 章では，CGF を用いた AC 回路の検証手法を提案する．4 章では提案手法とランダムテストの優劣を，検証時間とカバレッジの観点から定量的に比較する．最後に，5 章で結論を述べる．

2. 関連研究

本章では，まず，2.1 節において，関連研究としてファジングを説明する．次に，2.2 節では，CGF を AC 回路の品質検証に応用する際の課題を整理する．

2.1 ファジング

ファジングは，テストパターンの変形と PUT (テスト対象プログラムでの実行) を繰り返し行う手法である．1990 年初頭に Miller らによって提唱 [18] されて以来，様々な派生手法に展開され，現在では，ソフトウェアの品質テストにおける最重要技術の一つとして挙げられている [19], [20]．

ファジングの手法は，ブラックボックス，ホワイトボックス，グレイボックスの三種類に大別される．ブラックボックス・ファジングは，入力の構文や文法箇所などに対して，ランダムな変更を加えて PUT を実行する手法 [21] である．テストパターンの変異が非常に単純であるため，より多くの PUT を実行できるが，バグの発見能力に課題を持つ．ホワイトボックス・ファジングは，テスト対象のソフトウェアを解析し，分岐やメモリアクセスなどを考慮して PUT を実行する [22]．ソフトウェアの構造と文脈を考慮したテストパターン生成を行うため，バグの発見能力は高い．一方，複雑なソフトウェアに対しては解析時間が爆発的に増加し，計算時間のスケーラビリティに課題を持つ．グレイボックス・ファジングは，前述のブラックボックス・ファジングとホワイトボックス・ファジングの中間に位置する手法となっている．ソフトウェアに対して，軽量かつ近似的な静的解析を行い，PUT 実行時の動的な活性化情報と合わせてフィードバックすることで，バグの発見能力を大きく低減することなく，高速性を保つ狙いがある [16]．本研究では，グレイボックス・ファジングに着眼する．

グレイボックス・ファジングにおいて，コードカバレッジを入力の変異に取り入れる手法を，特に Coverage-guided Greybox Fuzzing (CGF)[13], [14], [15] と呼ぶ．CGF の動作について，図 1 を用いて説明する．CGF では，「テストパターンの変異，PUT の実行，カバレッジの集計とテストパターンへのフィードバック」を繰り返し行うことで，テスト対象空間を自動探索する．PUT 実行時に活性化したパスを計測し，新たなパスが実行された場合には，カバレッジを向上したテストパターンとしてキューに追加し，次の変異に利用する．動作開始時には，CGF のユーザーがあらかじめ用意した初期テストパターンをキューに格納し，PUT 実行に利用する．CGF は，変異とカバレッジの集計を軽量に抑えることで高速性を保ちつつ [15]，未知のバグに対する高い発見能力を達成しており [16]，ソフトウェア・システムの品質テストにおいて成功を収めている技術として，特に注目を集めている [14]．本研究では，CGF の高速性と高いバグ発見能力に着目し，CGF の AC 回路検証への応用に焦点を絞る．

2.2 CGF を用いた AC 回路検証の課題

前節で述べた通り，CGF は，PUT を実行後，カバレッジを計測する．カバレッジが向上した際には，現在の入力パターンを変異に取り入れることで，検証網羅性を高める．換言すれば，CGF の肝は，カバレッジの評価とテストパターンへのフィードバックにある．従って，CGF を AC 回路の品質検証に応用するためには，第一に，AC 回路の検証網羅性に則してカバレッジを評価する必要がある．第二に，計算品質を阻害するテストパターンを効率的に生成するために，PUT 実行後の計算品質をフィードバックし，制約違反の有無情報を変異に取り入れる必要がある．本節では，Listing 1 の近似加算器を例に，カバレッジの評価と計算品質の考慮に対する，CGF の課題を説明する．

Listing 1: 近似加算器

```
1  int sc_main (int argc, char *argv[]) {
2      sc_signal<bool> InA, InB, Cin; // 入力
3      sc_signal<bool> Y, Cout; // AC 加算結果
4
5      // AC 加算
6      AC_Full_Adder *AC_FA1;
7      AC_FA1 = new AC_Full_Adder ("AC_FA1");
8      (*AC_FA1)(InA, InB, Cin, Y, Cout);
9
10     return 0;
11 }
```

Listing 1 に，SystemC で記述した 1 bit の近似加算器を示す．入力として InA，InB，Cin を受け取り (2 行目)，Y と Cout を出力する (3 行目)．なお，簡単化のため，ヘッダファイルの読み込み，及び，近似加算器の中身の記載は割愛する．この近似加算器を検証する際には，どのような入力組 (InA，InB，Cin) が，計算結果 (Y，Cout) の誤差を大きくするのか明らかにする必要がある．InA，InB，及び，Cin は 1 bit 入力であり，論理値 0 と 1 の 2 種類の値を取り得るため，最大 $2^3 = 8$ 通りのテストパターンが存在する．8 通りのパターンの中から，Y と Cout の誤差が許容値を超えるものを，出来るだけ多く発見することが，検証の目的となる．

一方，既存の CGF 技術は，Listing 1 のような近似加算器に対して，単純には機能しない．第一の課題として，CGF は，if-else 等の分岐を指標にコードカバレッジを評価 [13] しており，検証対象パス数をハードウェアレベルで考慮出来ない．例えば，Listing 1 を CGF に与えると，CGF は分岐のないコードと解釈する．すなわち CGF は，テストパターンを 1 種類入力すれば，100% のコードカバレッジを達成できると考える．しかし，ハードウェアレベルでは，前述の通り，テストパターンの最大数は 8 であるため，両者には大きな乖離が存在する．カバレッジの解釈の乖離は，検証探索空間の誤認識に繋がるため，AC 回路の検証を適切に行うためには補正が不可欠である．第二の課題として，計算品質を考慮した，テストパターンへのフィードバック機構の不足が挙げられる．例えば，Listing 1 では，近似加算結果 (Y, Cout) がどれ程の計算誤差を持つか評価する機構が存在しない．計算品質の違反情報を正確に集計出来ない場合，品質を違反するパスの集合に対するカバレッジ評価や，変異機構に対するフィードバックを適切に実行することが出来ない．以上より，AC 回路の品質検証に CGF を応用するためには，上記の 2 点の課題を解決し，計算品質を考慮しつつ，カバレッジを適切に評価することが必要不可欠である．

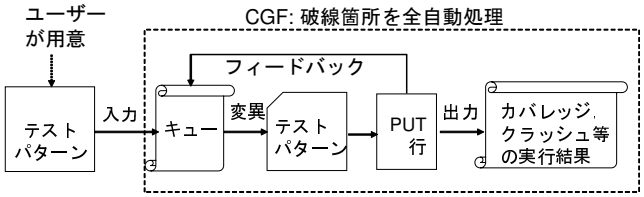

図 1: CGF の概要: 「テストパターンの変異，PUT の実行，カバレッジの集計とテストパターンへのフィードバック」を繰り返し行うことで，テスト対象空間を自動探索

3. 提案手法

本章では，CGF を利用した AC 回路の品質検証手法を提案する．提案手法の肝は，CGF と Design Under Test (DUT) 機構の活用にある．提案手法では，HDL の品質計算箇所に DUT 機構を埋め込むことで，計算品質の制約を診断しつつ，制約を違反するテストパターンに対する検証網羅性を評価する．

図 2 に，提案手法の概要を示す．入力として，C 言語を基軸とした HDL (SystemC，Cuda C，OpenCL 等) により記述されたハードウェアと目標の計算品質を与える．提案手法では，まず初めに，入力された HDL に対して DUT 機構を追加する．次に，DUT 機構を埋め込んだ HDL を CGF に渡し，対象ハードウェアの PUT を行うことで，検証網羅性および品質制約の違反情報 (違反時のテストパターンなど) を導出する．3.1 節 では，Listing 1 に対比する形式で，DUT 機構の実装法と貢献について説明する．

3.1 DUT 機構の挿入

2.2 節で前述した通り，CGF は計算品質の考慮とカバレッジの評価に課題を持つ．この課題を解決し，CGF を AC 回路の品質検証に応用するため，本研究では，DUT 機構を追加実装する．DUT の肝は，(1) 分岐命令を利用した信号伝搬経路の観測機構と，(2) 計算誤差の算出機構，及び，許容誤差を超過するテストパターンの特定機構の 2 点にある．本節では，Listing 1 に対して DUT を追加した Listing 2 を用いて，DUT 機構について詳述する．

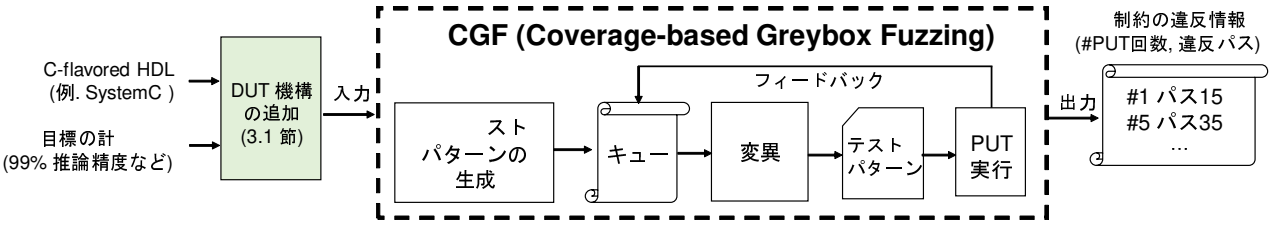

図 2: 提案手法の概要: 入力された HDL に対して DUT 機構を追加後，DUT 機構を埋め込んだ HDL を CGF に渡し，対象ハードウェアの PUT を行うことで，検証網羅性および品質制約の違反情報を導出

Listing 2: 近似加算器 (DUT 追加後)

```
1  int sc_main (int argc, char *argv[])  {
2
3      // CGF のテストパターンを受け取る機構
4      char buf[8];
5      if (read(0, buf, 8) < 1) { exit(1);}
6
7      sc_signal<bool> InA, InB, Cin; // 入力
8      sc_signal<bool> Y, Cout; // AC 加算結果
9      sc_signal<bool> EX_Y, EX_Cout; // 正解値
10
11     // カバレッジ診断用コード: 1 条件のみ記載
12     if (buf[0] == 'a'){
13         InA = true; InB = true; Cin = true;
14     }
15     // 残りの 7 条件は記載省略
16
17     // AC 加算
18     AC_Full_Adder *AC_FA1;
19     AC_FA1 = new AC_Full_Adder ("AC_FA1");
20     (*AC_FA1)(InA, InB, Cin, Y, Cout);
21
22     // 正確な加算
23     EX_Full_Adder *EX_FA1;
24     EX_FA1 = new EX_Full_Adder ("EX_FA1");
25     (*EX_FA1)(InA, InB, Cin, EX_Y, EX_Cout);
26
27     // 計算誤差の診断
28     if ( (Y - EX_Y) > ERROR_TH){
29         // 制約を違反するパターンの特定
30         if (buf[0] == 'a')
31         // 記載省略
32     }
33
34     return 0;
35 }
```

分岐命令を利用した信号伝搬経路の観測機構について説明する．まず，CGF から入力された文字列を格納する (3 行目から 5 行目)．次に，格納された文字列を元に，AC 回路の入力信号に対して，論理値 (0/1) を振り分ける (11 行目から 15 行目)．この振り分けにより，CGF により生成・変異されたテストケースを，HDL 上で取り扱える論理値に変換する．また，AC 回路の入力テストパターン組数の条件分岐を用意することで，分岐カバレッジを指標とする CGF に対して，AC 回路の探索空間を認識させることが出来る．例えば，Listing 2 の例では，(InA, InB, InC) = (0, 0, 0)···(1, 1, 1) の計 8 通りの条件分岐を用意することで，CGF は 8 種類の実行経路の活性化を目指して，テストパターンを生成することが出来る．なお，Listing 2 の例では，テストパターンは最大 8 通りあるが，簡単化のため 1 条件のみ記載している．

次に，計算誤差の算出機構，及び，許容誤差を超過するテストパターンの特定機構について説明する．9 行目と 22 行目から 25 行目は，近似誤差を計算するために用意した，正確な演算機構である．27 行目から 32 行目は，計算誤差の診断機構である．AC 演算結果と正確な演算結果を比較し，演算誤差がしきい値 (ERROR_TH) を超えているかどうか，診断する．演算誤差がしきい値より大きい場合は，演算器への入力値の組をトレースし (30 行目) テストパターンを特定する．

以上をまとめると，DUT を実装することで，AC 回路の検証探索空間に則してカバレッジを評価しつつ，計算品質を診断できる．従って，CGF と DUT を組み合わせることで，AC 回路の計算品質とカバレッジを考慮して，「テストパターンの変異，PUT の実行，テストパターンへのフィードバック」のループを構築できる．

4. 評価実験

本章では，提案手法の効果を PUT 実行回数とカバレッジの観点から評価する．4.1 節では評価環境を説明する．4.2 節では実験結果とその考察について述べる．

4.1 評価環境

本評価実験では，文献 [23] で提案されている近似全加算器を対象回路として選択し，3 bit 入力の近似加算器を，SystemC により実装した．次に，3 章の提案手法に従い，近似加算器に DUT 機構を追加した．計算誤差を以下の式により算出し，計算品質の制約として，誤差 0.1 を設定した．カバレッジは，「誤差 0.1 を持つ総テストパターン数

に対する，発見できたパターン数の割合」と設定した．

$$\text{誤差} = \frac{|\,\text{近似加算結果} - \text{正解値}\,|}{\text{正解値}}. \quad (1)$$

また，本比較実験では，パス間の活性化確率のばらつきや，到達確率の低い (活性化されにくい) パスを疑似的に考慮するために，近似加算器に入力される 0/1 の平均確率を仮想的に操作した．具体的には，CGF が格納した文字列を論理値 (0/1) に振り分ける機構 (例. Listing 2 の 12 行目) において，0，1 に変換する文字列の集合に偏りを持たせた．本実験では，近似加算器の各入力 bit に対して，論理値 1 が入力される確率として，$\frac{1}{4}$，$\frac{1}{8}$，$\frac{1}{16}$，$\frac{1}{64}$，$\frac{1}{128}$，$\frac{1}{256}$ の 6 種類を設定した．

CGF ツールとして，AFL 2.56b[13] を選択した．また，比較対象として，C++11 の乱数ライブラリを用いて，ランダムにテストパターンを生成する手法を実装した．AFL とランダムテストを用いて，PUT の実行回数とカバレッジのトレードオフ関係を，近似加算器を対象に定量的に評価した．評価実験には，Ubuntu 16.04 LTS と AMD Ryzen Threadripper 2920X 12-Core Processor を搭載した計算機マシンを用いた．本実験では，ランダムテストと AFL の実装法の差異を考慮して，公平な比較をするために，PUT の実行時間ではなく，PUT 回数を指標として選択した．

4.2 実験結果

本節では，PUT 実行回数とカバレッジのトレードオフ関係を，ランダムテストと提案手法の間で比較する．**図 3**，及び，**図 4** に，近似加算器における，トレードオフ評価結果を示す．各 bit に論理値 1 が入力される平均確率は，図 3 では $\frac{1}{4}$，図 4 では $\frac{1}{64}$ である．両図において，青色の線は提案手法，橙色の線は ランダムテストのトレードオフ関係を示す．

図 3 より，各パスの活性化確率が高く，到達確率の高い回路系においては，ランダムテストが高い性能を発揮していることが読み取れる．例えば，PUT 実行回数 2,000 において，提案手法ではカバレッジが 56% であるのに対して，ランダムテストでは 95% である．一方，図 4 では，提案手法が高いカバレッジを達成している．例えば，PUT 実行回数 2,500 において，提案手法ではカバレッジが 38% であるが，ランダムテストでは僅か 6% である．以上より，到達確率の低いパスを含む回路系において，提案手法がより良いトレードオフ関係を達成することを観測した．

次に，各 bit に論理値 1 が入力される平均確率を変更し，トレードオフ関係をより詳細に比較した．到達確率の高低と偏りが，AFL とランダムテストの検証範囲に与える影響を，実験的に評価することが目的である．3 bit 近似加算器の各入力 bit に対して，論理値 1 が入力される平均確率を $\frac{1}{4}$ から $\frac{1}{256}$ まで変更し，AFL 間とランダムテスト間のトレードオフ関係を比較した．評価結果を **図 5** と **図 6**

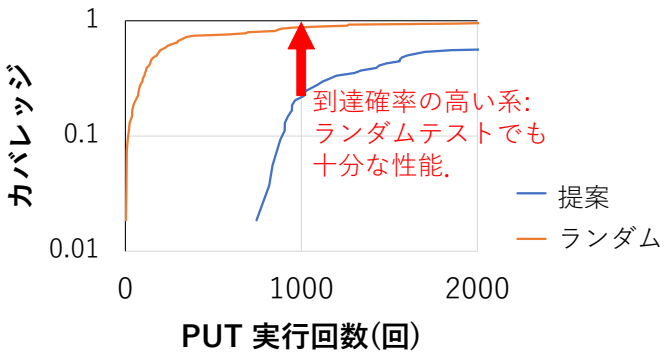

図 3: ランダムテストと提案法の比較（論理値 1 の確率 $\frac{1}{4}$）

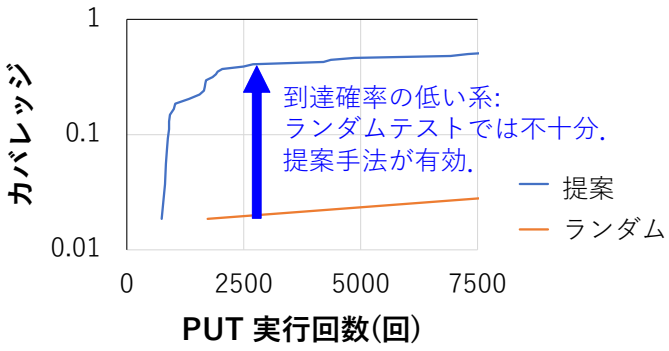

図 4: ランダムテストと提案法の比較（論理値 1 の確率 $\frac{1}{64}$）

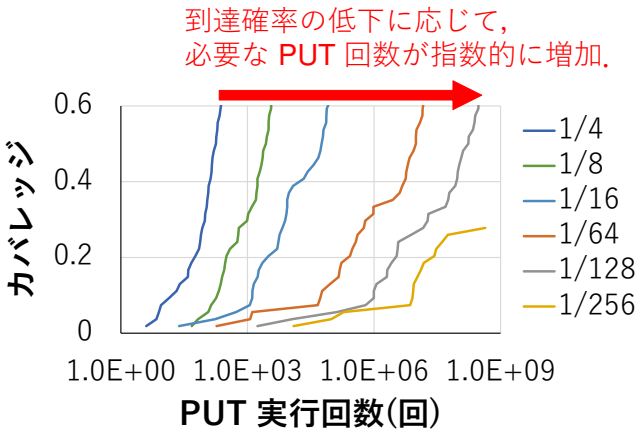

図 5: ランダムテスト: 到達確率の低いパスの探索が弱点

にそれぞれ示す．図 5 より，ランダムテストでは，平均確率が低下するにつれて，カバレッジの増加に要する PUT 実行回数が大幅に増加することが読み取れる．例えば，平均確率 $\frac{1}{4}$ では，PUT 実行回数 135 回 において，40% のカバレッジを達成するが，平均確率 $\frac{1}{128}$ では，9.45×10^7 回の実行を要する．一方，図 6 より，提案手法では，平均確率の低下が PUT 回数の増加に及ぼす影響を，大幅に緩和している．例えば，平均確率 $\frac{1}{4}$ と $\frac{1}{128}$ において，それぞれ 1.43×10^3 回と 2.70×10^3 回の PUT 実行で，カバレッジ 40% を達成している．以上より，到達確率の低いパス

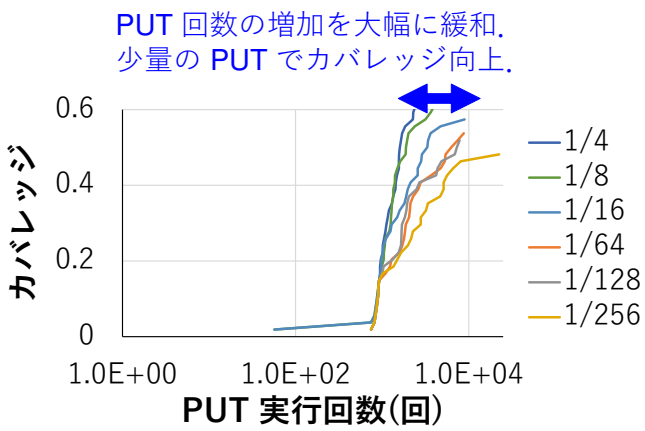

図 6: 提案手法: 到達確率の低いパスも早期に発見

を含む回路系において，提案手法がより良いトレードオフ関係を達成することを実験的に確認した．

5. 結論と今後の課題

本研究では，計算品質の制約を阻害するテストパターンの探索に根差した，近似コンピューティング 回路の検証技術を提案した．提案手法の肝は，CGF と DUT 機構の活用にある．CGF と DUT の併用により，(1) テストパターンの変異，(2) PUT の実行，(3) カバレッジの集計とテストパターンへのフィードバック，のループを実現した．PUT 実行回数とカバレッジのトレードオフ関係について，ランダムテストと提案手法を比較したところ，到達確率の低いパスを含む回路系において，提案手法がより良いトレードオフ関係を達成することを実験的に確認した．

本実験では，最も基本的な AC 回路の一つである，近似加算器を対象に評価実験を行った．今後は，近似ニューラルネットワークなど，より大規模な回路への適用に向けて，提案手法を洗練する．具体的には，(1) 初期テストパターンを PUT の実行クロックサイクル数に合わせて生成し，順序回路としての PUT 動作を実現すると共に，(2) DUT 機構を改良し，推論精度や PSNR 等の，異なる計算品質を評価する機構を実装する．他の CGF ツールを利用した検証手法の検討，タイミング検証法への拡張なども，今後の課題の一部である．

謝辞 本研究の一部は JSPS 科研費 (20K19767) の助成を受けたものである．

参考文献

[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] M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio, “A survey of testing techniques for approximate integrated circuits,” *Proc. IEEE*, pp. 1-17, 2020.
[5] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Architecture support for disciplined approximate programming,” *Proc. ASPLOS*, pp. 301-312, 2012.
[6] R. Hegde and N. R. Shanbhag, ”Soft digital signal processing,” *IEEE TVLSI*, vol. 9, no. 6, pp. 813-823, 2001.
[7] 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.
[8] 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.
[9] 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.
[10] 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.
[11] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, “MACACO: Modeling and analysis of circuits for approximate computing,” *Proc. ICCAD*, pp. 667–673, 2011.
[12] D. M. Lewis, “A hierarchical compiled code event-driven logic simulator,” *IEEE TCAD*, vol. 10, no. 6, pp. 726-737, 1991.
[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] 後藤敏宏, 山下茂 “Approximate Computing を用いた乗算器の実装および検証,” *Proc. IPSJ SIG Technical Report*, vol. 115, pp. 199-204, 2016.