社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

算術式の最適化を対象としたCコンパイラのランダムテスト

粟津　裕亘†　　石浦　菜岐佐†

† 関西学院大学理工学部　〒669-1337 兵庫県三田市学園 2–1

あらまし　本稿では, C コンパイラの誤りを検出する一手法として, 算術式の最適化を対象としたランダムテストを提案する. 本手法では, 種々の整数型の変数と暗黙の型変換を含む整数型算術演算に対するコード生成, 特に最適化が正しく行われていることを, ランダムに生成したプログラムによりテストする. ランダムに決定するものは, 変数の型, 初期値, 算術式である. 演算結果の期待値は生成時に計算し, テストプログラム内で比較する. この際, ゼロ除算, オーバーフロー, 負の値の左シフト等未定義の動作を引き起こすプログラムを生成しないようにする. 本手法に基づくランダムテストシステムのプロトタイプを実装した結果, x86 用コンパイラ GCC-4.1.2 の誤りを 1 つ検出することができた.
キーワード　C コンパイラ, ランダムテスト, 最適化, 算術式

Random Testing for Arithmetic Optimization of C compilers

Hironobu AWAZU† and Nagisa ISHIURA†

† School of Science and Technology, Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　This article presents random testing of C compilers focusing on arithmetic optimization. It tests if code generation and optimization are properly performed for integer arithmetic expressions containing various integer type variables and implicit casting. The types and the initial values of the variables are randomly selected and the arithmetic expressions are randomly composed to form a test program. The expected values of the expressions are precomputed by the random test program generator so that the comparison between the computed values and the expected values is done within the test program. During the preparation of the expected values, all the intermediate values are tested so that the generator do not produce programs that results in undefined behavior by zero division, overflow, nor left-shift of negative values, etc. An implemented random test system successfully detected a bug in GCC 4.1.2 for x86.
Key words　C compilers, random test, optimization, arithmetic expressions

1. まえがき

近年, 携帯電話, 家電製品, 自動車などのシステムの多機能化により, プロセッサには益々高い処理能力が要求されるようになっている. 一方で, サイズ, 携帯性, コストの制約から, この性能を省ハードウェアで実現することが要求される. 汎用プロセッサや汎用 DSP で消費電力, 性能, 面積の要求を満たせない場合には, ASIP (Application Specific Instruction-set Processor) や, コンフィギュラブルプロセッサ (configurable processor) [1] などが利用される.

特定用途向けに ASIP やコンフィギュラブルプロセッサが数多く開発されるのに伴い, それだけコンパイラの開発も必要になる. コンパイラは他のソフトウェア開発に用いられるため, そのテストは極めて重要な課題となる. 一般に, コンパイラの開発の初期段階では, コンパイラ開発者がテストプログラムを作製して, コンパイラの動作確認を行う. コンパイラが完成段階に近づくと, テストスイートを用いたテストを行う. C コンパイラのテストスイートには, GCC (GNU Compiler Collection) 付属のテストスイート (注1), testgen テストスイート [2] , Plum Hall (注2) , ACTEST (注3) , SuperTest (注4) などがある. これらのテストとデバッグによってコンパイラの信頼性が高められるが, このテストを経てもなお誤りが潜在する問題があり, 実際に GCC でも誤りが多く報告されている(注5). この問題を解決する一手法として, 時間の許す限りランダムテストを行う手法がある.

コンパイラのランダムテストとは, プログラムをランダムに生成して, 開発者やテストスイートが想定しない誤りをテスト

(注1)：http://gcc.gnu.org/install/test.html.
(注2)：http://www.a-qual.com/ph/index.html.
(注3)：http://www.a-qual.com/compiler/service/test_program.html.
(注4)：http://www.ace.nl/compiler/supertest.html.
(注5)：http://gcc.gnu.org/bugzilla/duplicates.cgi.

をするものであり, 誤りの発生しやすい特定の言語要素に的を絞ったものが提案されている. 例えば, 関数の呼出し規約を対象とした Quest [(注6)]や, volatile 宣言された変数を対象とした randprog [3] などがある. しかし GCC では, この言語要素以外にも, 算術式の最適化において多くの誤りが報告されている.

そこで本稿では, 算術式の最適化を対象とした C コンパイラのランダムテストを提案する. 本手法では, ランダムに選択した変数の型と初期値および算術式を含むプログラムを出力し, 最適化によって誤りが生じないかをテストする. ゼロ除算, オーバーフロー, 負の値の左シフト等を含むプログラムを生成しないようにする. 本手法に基づくランダムテストのプロトタイプを実装した結果, x86 用コンパイラ GCC-4.1.2 の誤りを 1 つ検出することができた.

以下, 2 章ではコンパイラの開発と C 言語の未定義/未規定/処理系定義の動作について述べる. 3 章では提案するランダムテスト手法を示し, 4 章ではランダムテストのプロトタイプの実装, およびこれを用いた結果と考察について述べる.

2. コンパイラの開発とテスト

2.1 コンパイラのリターゲッティング

ASIP を開発する都度コンパイラを一から開発するには膨大な労力を要するため, 既存のコンパイラのリターゲッティング (コンパイラの機械依存部分を変更し, 目的とするアーキテクチャに対応させること) による開発が主流になっている. このため, 字句解析や構文解析等のフロントエンドよりも, コード生成や最適化等のバックエンド, 中でも機械に強く依存する機能を重点的にテストすることが有効と考えられる.

2.2 未定義/未規定/処理系定義の動作

C99 [4] には, 計算結果が言語仕様上一意に定まらない, 未定義/未規定/処理系定義の動作がある.

未定義の動作 (undefined behavior) とは, 言語仕様がプログラムの実行結果に何ら要求を課していない動作である. 例えば, ゼロ除算, オーバーフロー, 負の値の左シフト, 左オペランドの幅以上のシフトに対する動作は未定義である. 未定義の動作に対して処理系は予測不可能な結果を返しても良いので, 未定義の動作を含むプログラムはテストとして用いるべきではない. 例えば, 図 1 の 3 行目では `a+1` の計算でオーバーフローが起こるので, このような文はテストに用いるべきではない. ただし, 4 行目の文はオーバーフローを起こさない [(注7)]のでこの限りではない.

```
1:          int a = INT_MAX;
2: unsigned int b = UINT_MAX;
3: if(a+1 != INT_MIN) abort(); /* 不適 */
4: if(b+1 != 0)       abort(); /*   適 */
```

図 1: 未定義の動作

未規定の動作 (unspecified behavior) とは, 言語仕様が二つ以上の可能性を提供し, どの可能性を選択するかに関して何ら要求を課さない動作のことである. 例えば, 副作用完了点間で関数を呼び出す順序や, 関数の引数の評価順序は未規定である. 未規定の動作により計算結果が一意に定まらないようなプログラムはテストとして用いるべきではない[(注8)]. 例えば, 図 2 の 12 行目の `i` の値は 2 か 3 のいずれになるか規定されていないので, このような文はテストに用いるべきでない. 一方, 13 行目における j の値は一意に定まるので, テストに用いても差し支えない.

```
 1:  static int i;
 2:  int f(int x) { i = x; return x;}
…
11:  int j = f(2)+f(3);
12:  if (i!=3) abort(); /* 不適 */
13:  if (j!=5) abort(); /*   適 */
```

図 2: 未規定の動作

処理系定義の動作 (implementation-defined behavior) とは, 未規定の動作のうち、処理系が選択した動作を文書化することになっているものである. 例えば, 符号付き整数を右シフトした場合の最上位ビットの伝播法は処理系定義である. 処理系定義の動作を含むプログラムはテストとして用いることができるが, 計算結果の期待値はその定義に従って計算する必要がある. 例えば, 図 3 の 2 行目は `v >> 1` が算術右シフトと定義されていれば適切なテストである. また, 4 行目は unsigned int 型が 16 ビットと定義されていれば, ラップアラウンドより `a+1` は `0` になるので適切なテストである (17 ビット以上の場合は不適となる) .

```
1: int v = -2;
2: if(v>>1 != -1) abort();  /* >> が算術右シフトなら適 */
3: unsigned int a = 0xffff;
4: if(a + 1 != 0) abort();     /* int が 16bit なら適 */
```

図 3: 処理系定義の動作

3. 算術式を対象とした C コンパイラのランダムテスト

3.1 テスト手法の概要

本稿では, コンパイラのテストスイートで検出できない誤りを検出する一手法として, 算術式の最適化を対象とした C コンパイラのランダムテストを提案する.

テストの流れを図 4 に示す. ランダムテストプログラムジェネレータが, テストプログラムを生成する. 期待値はランダムテストプログラムジェネレータが計算し, テストプログラムの内部で実行結果と比較する. 生成したテストプログラムを最適化オプションを用いてコンパイル・実行した結果, 正しくないことが確認された場合は, そのテストプログラムを保存し, 新たなテストプログラムを生成する. このテストは, ユーザが意図的にプログラムを中止するまで, 時間が許す限り行う.

テスト後は, エラーを起こしたプログラムに対して手動で解

(注6) : http://www.st.cs.uni-sb.de/~ lindig/src/quest/.

(注7) : 符号無し整数の場合はラップアラウンドにより値が一意に定まる

(注8) : 可能な全ての期待値を準備して比較する場合はこの限りではない

析を行い, 誤りの原因の特定を行う. このためには, プログラムの「最小化」 (エラーが起こる状態を維持しつつプログラムをできる限り縮約すること) の作業が必要になる.

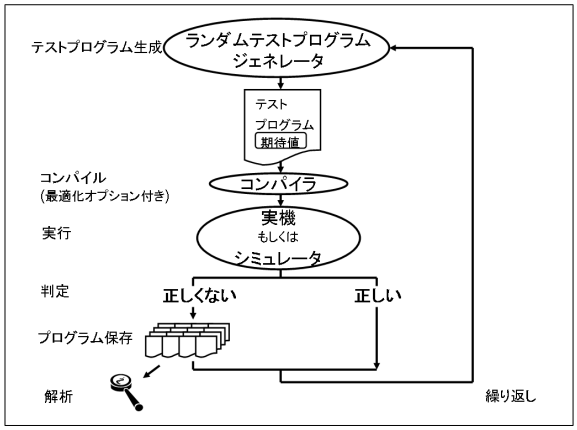

図 4: ランダムテストの流れ

3.2 テストプログラム

本稿のランダムテストジェネレータが出力するテストプログラムの例を図 5 に示す. 一つのテストプログラムは, 平均 70 行程度である. プログラムは, 宣言/初期値設定部, 演算部, テスト部から成る.

2〜10 行目の宣言/初期値設定部において, 使用する変数の型をランダムに宣言し, この型の範囲内の数値で初期化する. 変数の型は全ての標準符号付き整数型, 符号無し整数型であり, 非修飾型のうち volatile, 記憶域クラス指定子のうち static を対象とする. 変数の数はランダムに決定する.

14〜16 行目の演算部はランダムに生成した算術式より成る. 演算子は, 全ての二項演算子であり, 変数と組み合わせてランダムな算術式を生成する. 代入演算子は, 一文当たり一度のみ用いる.

20〜21 行目のテスト部では期待値と演算部で計算された結果を比較する. 20 行目の 9 が期待値である. 20〜21 行目の printok, printno は, 結果を確認するための関数である. 標準出力を使用できる環境であれば, 結果が正しい/正しくないことを表す文字列を出力する.

3.3 算術式の生成法と未定義動作への対応

2.2 節でも述べた通り, コンパイラのテストを行う際には未定義/未規定/処理系定義の動作に留意する必要がある. 本稿で提案する手法では, これらの動作を次のように扱う.

- 未定義の動作を引き起こす算術式は出力しないようにする.
- 未規定の動作を引き起こす構文は本手法のテストプログラムには含まれない.
- 処理系定義の動作を含むプログラムは定義に従って期待値を計算する.

テストプログラムの算術式および変数の初期値は次の方法により生成する.

(1) 算術式を表す木をランダムに生成する. 葉となる節点に `var1, var2`…と変数名を与える.

```
1:  /* 宣言/初期値設定部 */
2:      int test=1;
3:      unsigned int var0 = 59332;
4:      unsigned short var1 = 0;
5:      signed char var2 = 139;
6:      signed short var3 = -7;
7:      unsigned int var4 = 52;
8:      unsigned short var5 = 1;
9:      signed char var6 = 19;
10:     signed short var7 = -433;
11:     …
12:
13: /* 演算部 */
14:     test += test + var0 % var2 || …
15:     test += s2.var1 >= var3 > var5 …
16:     test += var0 + test < var4 …
17:     …
18:
19: /* テスト部 */
20:     if(9 == test) printok();
21:     else printno();
```

図 5: テストプログラムの例

(2) 各変数に対して型をランダムに決定する.

(3) 各変数に対して初期値をランダムに決定する.

(4) 葉から根に向かって期待値を生成する. この際未定義動作 (ゼロ除算, オーバーフロー, 負の値の左シフト, 左オペランドの幅以上のシフト等) が検出されれば, 現在の初期値を破棄して (3) に戻る. ただし, 100 回初期値の再設定を行っても未定義動作が起こる場合には, 現在の算術式を破棄して (1) に戻る.

算術式の木構造の高さは, P_L, P_R という 2 つのパラメータによって制御する. P_L, P_R は節点がそれぞれ左の子, 右の子を持つ確率である.

変数の値の初期値の乱数に一様乱数を用いると, ビット数の小さい数が生成される確率が小さくなる問題がある. 本手法では, まずビット数をランダムに決定し, 次にそのビット数だけ `0` または `1` をランダムに生成して得られる二進数を乱数として用いる.

4. ランダムテストの実装と評価

4.1 実　　装

提案手法に基づくランダムシステムのプロトタイプを Perl 5 で実装した. 動作環境は, Windows 上の cygwin, Mac OS X, Linux である. 算術式の結果が期待値と一致すれば printok 関数によって `@OK@` を, そうでなければ printno 関数で `@NG@` を出力するようにした.

4.2 評　　価

4.2.1 x86 用コンパイラのテスト

x86 用コンパイラ GCC-4.1.2 (kubuntu, x86) のテストを行った. テストを開始してから約 24 時間経過した時点で最適化オプション-O1 でエラーが検出された.

図 6 は, このプログラムを最小化したものである. このプロ

グラムの期待値は 1 で, 最適化オプションなしでは期待値が得られるが, -O1 オプション以上でコンパイル・実行すると 2 が出力される.

```
1: #include <stdio.h>
2:
3: int main (void)
4: {
5:   int test=1;
6:   volatile int a = 1829;
7:
8:   test += (0 == a);
9:   test += ((test + test) <= -69);
10:  printf("%d\n",test);
11:  return 0;
12:}
```

図 6: 最小化したテストプログラム (1)

図 7 は, 図 6 のプログラムの 5〜10 行目に対するアセンブリコードである. 7 行目の即値 2 が 9 行目の printf で直接表示されており, 定数伝播および定数の畳込みの最適化の誤りによってエラーが生じていると考えられる.

```
1:  movl        $1, -8(%ebp)
2:  movl        -8(%ebp), %eax
3:  movl        $.LC0, (%esp)
4:  testl       %eax, %eax
5:  sete        %al
6:  movzbl      %al, %eax
7:  addl        $2, %eax
8:  movl        %eax, 4(%esp)
9:  call        printf
```

図 7: アセンブリコード

図 6 の 9 行目の`-69` を `-70, -68` などにした場合には, エラーは起こらない. また, volatile 宣言を削除した場合もエラーは起こらない. これにより, この誤りは様々な値や型に依存すると考えられる.

なお, GCC-4.2.2 以降ではこの誤りは修正されている.

4.2.2 Brownie32STD 用コンパイラのテスト

32 ビット RISC プロセッサである Brownie32STD [1] 用コンパイラ GCC-4.2.2 (Cygwin, x86) のテストを行った. このコンパイラは, testgen テストスイートテスト [2] ではエラーが無いことが確認されている.

テストを開始してから約 34 時間経過した時点で最適化オプション-O2 でエラーが検出された. 図 8 は, このプログラムを最小化したものである. 期待される出力は `@OK@` 3 つである. 最適化オプションなしの場合には期待通りに動作するが, -O2 オプションを用いた場合, `@OK@` が 2 つしか出力されない.

12〜13 行目の式を少しでも短くするとエラーは発生しなくなる. 式のこのような組み合わせは組織的には作り難いため, ランダムテストは有用であると考える.

```
 1:  #include <stdio.h>
 2:
 3:  #define printok() printf("@OK@\n")
 4:  #define printno() printf("@NG@\n")
 5:
 6:  int main (void)
 7:  {
 8:      int test=1, i;
 9:      unsigned long var1 = 2605;
10:      volatile short var0 = 4048;
11:
12:      test += var1 != 1 || 1 % var1 - var1 <
13:              var1 % var1 % 4048 > var0;
14:
15:      if(2605 == var1) printok();
16:      else printno();
17:      if(4048 == var0) printok();
18:      else printno();
19:      if(2 == test) printok();
20:      else printno();
21:
22:      return 0;
23:  }
```

図 8: 最小化したテストプログラム (2)

5. 結　　論

本稿では, 算術式の最適化に焦点を絞った C コンパイラのランダムテスト手法を提案した. 本手法に基づくランダムテストのプロトタイプを実装した結果, x86 用コンパイラ GCC-4.1.2 の誤りを 1 つ検出した.

今後の課題としては, 誤りを検出したプログラム・最適化オプションの最小化の自動化などが挙げられる. また, 浮動小数点演算を含む算術式のランダムテストも重要な課題と考える.

謝辞　本研究に関して, 多くの御助言を頂きました株式会社 SRA の引地信之氏, SRA OSS, Inc. 日本支社の矢吹洋一氏に深く感謝申し上げます. 本研究に際し, 多くの助言や協力を頂いた元 関西学院大学吉田昌平氏 (現 株式会社アックス), 内山裕貴氏 (現 株式会社ケイ・オプティコム), 関西学院大学の多賀惣一郎氏, 森本和志氏に心より感謝申し上げます.

文　　献

[1] 岩戸宏文, 稗田拓路, 田中浩明, 佐藤淳, 坂主圭史, 武内良典, 今井正治: “ASIP 短期開発のための高い拡張性を有するベースプロセッサの提案,” 信学技報, VLD2007–92 (Nov. 2007).

[2] 内山裕貴, 引地信之, 石浦菜岐佐, 永松祐二: “C コンパイラ用テストスイートおよびその生成ツール testgen,” 信学技報, VLD2006–95 (Jan. 2007).

[3] Eric Eide and John Regehr: “Volatiles Are Miscompiled, and What to Do about It,”in Proceedings of the 7th ACM International Conference on Embedded Software, pp.255–264(Oct. 2008).

[4] 日本規格協会: JIS X 3010 プログラム言語 C (Oct. 1993).