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.

Cコンパイラの算術最適化のランダムテストにおける式生成の強化

永井絵里子[†]　橋本　淳史[†]　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 コンパイラの算術最適化を対象とするランダムテストにおける不具合検出能力の向上を目的として, 式生成を強化する手法を提案する. 本手法では, 従来手法よりも長く, かつ C 言語の未定義動作を引き起こさない算術式を生成する. これは, 式の期待値計算の過程で未定義動作を検出すると, 式を変形することによりその未定義動作を回避するという方法による. また, 1 つのプログラムに複数の式を出現させることにより, 不具合検出能力の強化を図る. 本手法に基づくランダムテストシステムを実装した結果, GCC 4.7.2 (x86_64-apple-darwin10), GCC 4.5.4 (i686-pc-linux) などの不具合を検出することができた.

キーワード　コンパイラ, ランダムテスト

Scaling the Size of Expressions in Random Testing of Arithmetic Optimization of C Compilers

Eriko NAGAI[†], Atsushi HASHIMOTO[†], and Nagisa ISHIURA[†]

† Kwansei Gakuin University, 2-1 Gakuen, Sanda, Hyogo, 669-1337, Japan

Abstract　This paper presents an enhanced method of testing validity of arithmetic optimization of C compilers using random programs. It generates arithmetic expressions which are longer than a previously proposed method but yet does not lead to undefined behavior . This is achived by modifying the expressions on detecting undefined behavior during the computiation of the correct values of the expressions. The detection ability is further improved by incorporating multiple expressions into each program. Experimental results show that a random testing system based on our new method has higher capabilities of finding compiler bugs; it has detected more bugs than previous method in earlier versions of GCCs and it has also revealed flaws in some latest versions of GCCs which have been unknown so far.

Key words　Compiler, Randomtest

1. はじめに

コンパイラはソフトウェア開発の基盤ツールであり, 極めて高い信頼性が要求される. 組み込みシステムにおいては, 厳しい性能/省電力要求を満たすために, 新しい機能や特定用途向けの命令セットを持ったプロセッサが開発される [1] が, そのためにコンパイラの開発も必要になる. また, 既存のプロセッサに対してもコンパイラに新しい最適化が実装されるため, これらのコンパイラのテストは重要な課題となる.

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ことがあり, 実際に GCC でも多くの誤りが報告されている(注5).

この問題を解決する一手法がランダムテストである. ランダムテストは, ランダムに生成したプログラムによるコンパイラのテストを時間の許す限り行うことにより, 開発者やテストスイートが想定しない誤りを検出しようとするものである. コンパイラのランダムテストに関する研究としては, 関数呼び出し

(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.

における引数と返り値の受け渡しのテストを行う Quest [3] や, volatile 宣言された変数を最適化が正しく扱うことをテストする randprog [4], また配列や構造体, for 文や while 文など様々な構文を含むテストを行う Csmith [5] などがある.

Csmith は, GCC 4.5 や LLVM 2.8 などのコンパイラに対し, 3 年間で約 325 個の不具合を検出しており, これらのコンパイラの信頼性向上に貢献してきた. この手法は differential testing [6] に基づいており, 1 つのテストプログラムを複数のコンパイラによってコンパイルし, その実行結果を比較することによりエラー判定を行っている. ランダムに生成されたプログラムに対して期待される出力 (期待値) をあらかじめ計算する必要はないが, C プログラムの未定義動作をプログラムの静的解析だけに基づいて回避しなければならない. またエラーを検出したプログラムの最小化が難しいという課題があった.

これに対し, 文献 [7–9] の算術式を対象とするランダムテスト手法では, プログラムの期待値計算に基づいてエラー判定を行っている. 未定義動作は期待値計算の過程で検出できるため, 変数の初期値の再設定や式の再生成よりこれを回避できる. またエラープログラムの最小化も容易に行える.

この手法は, GCC 4.2.1 (apple-darwin10), GCC 4.3.4 (i686-pc-cygwin), GCC 4.4.1 (i686-pc-linux) 等の不具合を検出しているが, より新しい GCC 4.5 以降のバージョンでは不具合を検出できていない. この手法の未定義動作回避方法では長い式が生成できないこと, また 1 つのテストプログラムに 1 つの式しか生成されないことがその一因と考えられる.

そこで本稿では, [7–9] の手法に対し, 算術式の生成法を改良することによって不具合検出能力の強化を図る手法を提案する. これは式の修正による未定義動作の回避によって長い式の生成を可能にし, さらに 1 つのプログラムに複数の算術式を生成することにより実現する.

本手法に基づくランダムテストシステムを実装した結果, [7–9] の手法では検出できなかった GCC 4.7.2 (x86_64-apple-darwin10), GCC 4.5.4 (i686-pc-linux) 等の不具合を新たに検出することができた.

以下, 2 章ではコンパイラのランダムテストおよびに [7–9] の手法を概観した後, 3 章で提案する生成式の強化手法について述べる. 4 章で検出したエラープログラムの最小化手法, 5 章では実装と実験結果を示した後, 6 章で結論を述べる.

2. 算術式を対象とするコンパイラのランダムテスト

2.1 コンパイラのランダムテスト

コンパイラのランダムテストの流れを図 1 に示す. プログラムをランダムに生成し, 生成したテストプログラムをコンパイル・実行する. その結果が正しくないと判定した場合は, そのプログラムを保存する. この処理を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の分析を行うが, この際にエラー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) を自動または手動で行うことが必要になる.

```
while(規定の回数 or 時間内) {
  ランダムプログラム生成;
  コンパイル & 実行;
  if(エラー) {記録;}
}
```

図 1　コンパイラのランダムテストの流れ

コンパイラのランダムテストにおける一つの重要な課題は, 生成されるプログラムに未定義動作が含まれないようにすることである. 未定義動作 (undefined behavior) とはゼロ除算や NULL ポインタによるオブジェクトへのアクセス等, 「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」と C99 [10] 等の C 言語の規格で定められているものである. 未定義動作を 1 つでも含むプログラムは不正なプログラムであるため, その生成を回避することが必要になる.

未定義動作が発生するプログラムの例を図 2 に示す. 10 行目の `t0` の値が 670 となるため, 11 行目の式では, `/` 演算子の右オペランドの値 `a > (c * t0)` が 0 となり, ゼロ除算が発生する. また `<<` 演算子の右オペランドの値 `b * d - c` は 40 となり, 左オペランドの幅を超える左シフト演算が発生する. C99 にはこの他にも, 符号付き整数型のオーバーフローや配列インデックスの範囲外指定等, 全部で 191 の未定義動作が規定されている.

未定義動作にはプログラムの静的解析で検出できるものもあるが, 値に依存するものは実行時にしか検出できない. このため, プログラムの期待値計算を行わない Csmith [5] 等では, 生成でするプログラムに何らかの制限を課す必要がある.

```
 1: #define MAX = 10
 2:
 3: int main (void)
 4: {
 5:   int a = 60;
 6:   int b = 10;
 7:   int c = 30;
 8:   int d = 7;
 9:
10:   int t0 = b * (a + d);
11:   int t1 = (a << (b * d  - c)) /  a > (c * t0));
12:
13:   return 0;
14: }
```

図 2　未定義動作を含むプログラムの例

2.2 算術式を対象とする C コンパイラのランダムテスト

文献 [7–9] は, 算術式のコード生成およびその最適化を対象とするランダムテスト手法を提案している. この手法では, 算術式の期待値計算を行い, その期待値と実行結果の比較よるエラー判定を行っている. 未定義動作が発生した場合は, 変数の初期値の再設定や式の再生成によりこれを回避することができる. またエラープログラムの最小化により, エラーの原因の解析も容易にできる.

この手法で生成されるテストプログラムの例を図 3 に示す. 3-10 行目では変数の宣言, 12 行目ではそれらの変数を参照する算術式があり, 14-20 行目で期待値との比較を行っている. 使用

する変数は, 型, 初期値, 有効範囲 (局所変数か外部変数か), 型修飾子 (`const`, `volatile`, `const volatile`, または無し), 記憶域クラス指定子 (`static`, または無し) をランダムに選択したものである. 式の生成は図 4 の make_expression を呼び出すことにより行っている.

```
 1: #include <stdio.h>
 2:
 3: const volatile unsigned char x1  = 2U;
 4: const volatile signed long long x6  = 1476669LL;
 5: static const unsigned short x8  = 35U;
 6:
 7: int main (void)
 8: {
 9:     int rc = 0;
10:     signed long long test = 0;
11:
12:     test = (((x8*(x6<<x8))>=x1)/x6);
13:
14:     if (test == 0LL) {
15:        printf("OK, %lld\n",test);
16:     }
17:     else {
18:        rc = 1;
19:        printf("NG, %lld\n",test);
20:     }
21:     return rc;
22: }
```

図 3　文献 [7–9] の手法で生成されるテストプログラム

```
 1: // R は 0 < R < 1 の実数定数 (0.45 程度)
 2: node make_expression
 3: {
 4:    r = 0.0〜1.0 の乱数;
 5:    if (r <= R)   { return 変数節点; }
 6:    else  { op = 演算子をランダムに決定;
 7:       left = make_expression;
 8:       right = make_expession;
 9:       return 演算節点 (op, left, right);
10:    }
11: }
```

図 4　式の生成法 (従来手法)

未定義動作の回避は, 次の方法で行っている.

1) ランダムに式を生成する.
2) ランダムに変数の初期値を設定する.
3) 式の期待値を計算する.
4) 未定義動作無しならプログラムを生成して終了.
   未定義動作有りなら 2) に戻る.
   ただし, 同じ式に対して 2) を 100 回行っても未定義動作が回避できなければ, 式を破棄して 1) に戻る.

図 4 の式生成手法では種々の長さの式が生成されるが, 長い式には未定義動作が発生する確率が高いため破棄される確率も高くなり, 短いものだけが残る傾向がある. 10,000 回のランダムテストを行った結果, 生成された算術式の平均サイズは 4 演算子であり, 最大でも 50 演算程度であった. また 1 つのテストプログラムに 1 つの式しか生成されないことも不具合検出能力が限定される一因と考えられる.

3. 式生成の強化

本手法では文献 [7–9] において, 長い算術式を生成し, さらに 1 つのプログラムに複数式を生成することにより, ランダムテストの不具合検出能力の強化を図る.

3.1 長い式の生成

変数の初期値の再設定や式の再生成により未定義動作を回避する方法では長い式の生成は難しい. そこで本手法では, 未定義動作を引き起こす部分式を修正することによって生成した式から未定義動作を排除するというアプローチをとる. 具体的には, 期待値計算時に未定義動作を検出すると, 演算を挿入してこれを回避する.

(1) ゼロ除算および不正なシフト幅の回避

除算や剰余算の右辺の値が 0 であれば, 加算を挿入して 0 以外の値にする. 図 5(a) に例を示す. 算術式 `(x1+x2)/(x2<(x3*x4))` に, `x1=8, x2=15, x3=4, x4=5` という初期値が設定されているため, ゼロ除算が発生する. この場合, 加算を挿入して新たに生成した変数 (`x5=10`) を加算する. これにより式は `(x1+x2)/((x2<(x3*x4))+x5)` となりゼロ除算は発生しなくなる. 加算する値は当該の除算・剰余算の型の範囲内で 0 以外の数からランダムに選択する. シフト演算の右オペランドが許容範囲の値ではない場合も同様に, 加算を挿入して値が範囲に収まるようにする.

(2) オーバーフローの回避

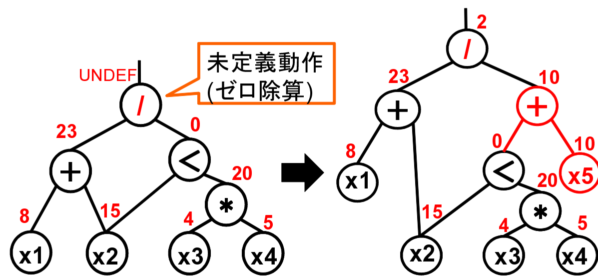
ある演算で符号付き整数のオーバーフローが発生した場合には, その演算のいずれか一方のオペランドに加算を挿入してオーバーフローを回避する. 図 5(b) に例を示す. 算術式 `x1+x2` に, `x1=INT_MAX, x2=1` という初期値が設定されているため, オーバーフローが発生する. この場合には, 右オペランドに加算を挿入し, 新たに生成した変数 (`x3=-6`) を加算する. これにより式は `(x1+x2)+x3` となりオーバーフローは発生しなくなる. 加算する値は変数の型の範囲内で, 演算結果が型の範囲内に収まるような値からランダムに選択する.

整数型の算術演算の際に発生する未定義動作は C99 規格では全部で 8 個ある (表 1) が, 全てに同様の操作を行う.

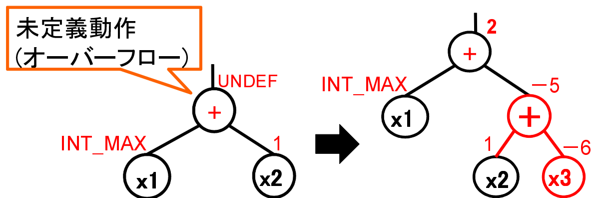
表 1　算術式に現れる未定義動作

演算子	条件
/	右オペランドの値が 0
%	右オペランドの値が 0
<<	右オペランドの値が負
	右オペランドの値が左オペランドの bit 幅以上
	左オペランドの値が符号付き整数型で値が負
>>	右オペランドの値が負
	右オペランドの値が左オペランドの bit 幅以上
	結果の型が符号付き整数型, オーバーフロー

また, 2.2 節の式の生成法では, 生成する算術式の演算子数を制御することはできなかった. そこで本手法では図 6 の make_expression によって指定された演算子数の算術式の生成を行う. この関数は演算子数 n と式の深さの上限 d を受け取り, 式を表す木を生成してその根の節点を返す. 左辺と右辺の節点数をその和が $n-1$ となるようにランダムに決定し, 再帰的に make_expression を呼び出して式を生成する. 深さを制限しているのは, C99 の規格では 1 つの完結式における括弧で囲



(a) ゼロ除算



(b) オーバーフロー

図 5　未定義動作回避処理の例

まれた式の入れ子のレベル数が 63 までと翻訳限界が定められているためである．

生成する式の規模は，1〜10,000 演算子程度を想定している．

```
 1: node make_expression(n, d)
 2: // 演算子数 n, 深さ d を受け取って, 節点を返す
 3: {
 4:   if (n==0 || d==0)  { return 変数節点; }
 5:   else  {
 6:    ln = 左辺の演算子数 (0〜n-1) をランダムに決定;
 7:    rn = n - 1 - ln;
 8:    op = 演算をランダムに決定;
 9:    lnode = make_expression(ln, d-1);
10:    rnode = make_expression(rn, d-1);
11:    return 演算節点 (op, left, right);
12:  }
13: }
```

図 6　式の生成法 (本手法)

3. 2　複数式を含むプログラムの生成

本手法では，1 つのプログラムに複数式生成することによりテストプログラムを強化を図る．生成するプログラムの例を図 7 に示す．17-19 行目のように複数の式 (代入文) を生成している．式の計算結果を代入する変数は，それ以後に現れる式で参照されることを許す．21-23 行目では各式の計算結果をそれぞれの期待値と比較することによりエラー判定を行っている．

生成する式の数は，1〜10,000 式程度を想定している．

4.　エラープログラムの最小化

エラープログラムの最小化は，エラーの分析を行う上で不可欠である．本稿では [9] の最小化手法を複数式に対応させるとともに，長い式の最小化が高速に行えるように二分探索を導入する．またプログラム中に現れる定数や変数の型をできるだけ簡潔なものに変換する．

エラープログラムの最小化は，(1) 式の削除，(2) トップダウン最小化，(3) ボトムアップ最小化，(4) 値と型の最小化という

```
 1: #include <stdio.h>
 2: #define OK()  printf("@OK@")
 3: #define NG()  printf("@NG@")
 4:
 5: static signed int x5 = 10;
 6: const volatile signed long x6 = 8L;
 7: static signed int x8 = 2;
 8: signed long t0 = 70L;
 9: signed int t1= 820;
10: signed long t2 = 9;
11:
12: int main (void)
13: {
14:   signed long x1 = 100L;
15:   signed int x3 = 32;
16:
17:   t0 = (((x8 * (x6 << x8)) >= x1) / x6;
18:   t1 = ((t0 + x3) * (x5 << x8));
19:   t2 = ((x1 + t0) - t1) * x6;
20:
21:   if (t0 == 0L) { OK(); }   else{ NO(); }
22:   if (t1 == 1280) { OK(); }   else{ NO(); }
23:   if (t2 == -9440L) { OK(); }   else{ NO(); }
24:
25:   return 0;
26: }
```

図 7　複数式を含むテストプログラム

4 つの基本操作をエラーの出る限り繰り返し適用することにより行う．

(1)　算術式の削除

図 8 のように，式の 1 つ (この例では t1 の式) を削除し，式を代入している変数の初期値を式の期待値で書き換える．

```
int t1 = 52;
  ...
t1 = ((x8 * x0) + x2) << x4;   /*期待値 256*/
t2 = x3 < (x5 * (x4 % x1));
```

⇓

```
int t1 = 256;
  ...
t2 = x3 < (x5 * (x4 % x1)):
```

図 8　算術式の削除

(2)　トップダウン最小化

図 9 のように，式をトップの演算の一方のオペランドで置き換える (これに伴い期待値も変更する)．

```
int x1 = 5; int x2 = 7;
int t = ( x1 + x2 ) / x1;
if ( t == 2 ) { OK(); }
else { NG(); }
```

⇓

```
int x1 = 5; int x2 = 7;
int t = ( x1 + x2 );
if ( t == 12 ) { OK(); }
else { NG(); }
```

図 9　トップダウン最小化

(3)　ボトムアップ最小化

図 10(a)，(b) のように，それぞれ変数への値の代入，演算の評価を行う．

```
int x1 = 2; int x2 = 3;
int t = ( x1 + x2 ) * x1;
if ( t == 10 ) { OK(); }
else { NG(); }
```

⇓

```
int x1 = 2; int x2 = 3;
int t = ( 2 + x2 ) * x1;
if ( t == 10 ) { OK(); }
else { NG(); }
```

(a) 値の代入

```
unsigned int x3 = 1;
unsigned int t = ( -3 + 2 ) * x3;
if ( t == 4294967295U ) { OK(); }
else { NG(); }
```

⇓

```
unsigned int x3 = 1;
unsigned int t = -1 * x3;
if ( t == 4294967295U ) { OK(); }
else { NG(); }
```

(b) 演算の評価

図 10　ボトムアップ最小化

(4)　値と型の最小化

値の最小化は図 11(a) のように, 変数の初期値や式中の定数の絶対値を小さくするものである. 型の最小化は図 11(b) のように, 値が収容できる型への変更, 外部変数から局所変数への変更, 型修飾子の削除, 記憶域クラス指定子の削除等を行うものである.

```
long long x1 = 4223372036854775 80;
int x2 = 100;
int t = x1 + x2 << (x1 > 0);
if ( t == 4223372036854776 80) { OK(); }
else { NG(); }
```

⇓

```
long long x1 = 192056;
int x2 = 100;
int t = x1 + x2 << (x1 > 0);
if ( t == 192156) { OK(); }
else { NG(); }
```

(a) 値の最小化

```
long long x1 = 1;
```

⇓

```
long x1 = 1;
```

```
volatile int x2 = 4;
```

⇓

```
int x2 = 4;
```

(b) 型の最小化

図 11　値と型の最小化

最小化の処理の流れを図 12 に示す. (1) を行った後, (2)(3) を適用できなくなるまで繰り返し交互に行う. (2)(3) の一方で効果があった場合は再び (1) に戻り, なければ (4) を行う. (4) で効果があれば再び (1) に戻り, なければ終了する. ただし (1) と (2) は初回のみ二分探索を行うが, 2 回目以降は線形探索を行う.

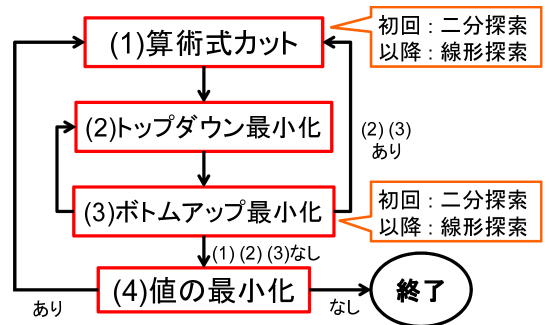


図 12　エラープログラムの最小化

5. 実装と実験結果

5.1 実　　装

提案手法に基づくランダムシステムを Perl 5.10.1 で実装した. 式の規模と式数は, 生成するプログラム毎にランダムに決定する (式の規模と式数の積を定数として与える). 動作環境は, Windows Cygwin, Mac OS X, Ubuntu Linux である.

5.2 実 験 結 果

GCC の 8 つのバージョンについて, 従来手法 [9] と本手法によるテストを行った. 結果を表 2 に示す. テストした最適化オプションは `-O0` と `-O3` の 2 つである. 表中の「CPU」はテストを実行するのに使用した計算機であり, 「size」は式の規模と式数の積, 「テスト数」は生成したテストプログラムの数, 「エラー」は検出したエラーの数, 「エラーの種類」は検出したエラーの種類数である. 従来手法でエラーが検出できていたコンパイラでは, より短い時間で多くのエラーを検出できている. また, 従来法ではエラーが検出できないコンパイラに対しても, エラーを検出することができている.

GCC の 3 つのバージョンについて Csmith [5] と本手法によるテストを行った結果を表 3 に示す. テストはオプションなし, `-O0`, `-O3` の 3 通りについて行った. 本手法は, 同じ時間内に Csmith より多くの不具合を検出することができている.

GCC の 6 つのバージョンについて, 長い式の生成, および複数式の生成の効果を調べるために, 従来手法, 長い式のみ生成する手法, 複数式のみ生成する手法, 長い式と複数式の両方を用いる手法の 4 通りについて実験を行った. 結果を表 4 に示す. テストしたオプションは `-O0` と `-O3` の 2 つである. 表から, 式を長くすることも, 式を複数にすることも, いずれも不具合検出能力の向上に寄与していると言える. コンパイラにより違いはあるが, 全般的に両方を適用した方が効果は大きいと言える.

表 2 において GCC 4.7.2 の不具合を検出した 3 つのプログラムを分析した結果は下記の通りである.

1)　最小化の結果, 計 38 演算式を含む 16 の算術式から成るプログラムが得られた. 最適化オプション `-O2` と `-O3` でエラー (期待値の不一致) が発生したが, `-O0`, `-O1` オプションでは発生しなかった. エラーが発生する最小のオプションセットは `-O1 -foptimize-register-move` であった.

2)　最小化の結果, 計 26 演算式を含む 18 の算術式から成るプログラムが得られた. エラーが発生する最小のオプションセッ

表 2　実験結果 (従来手法との比較)

コンパイラ (ターゲット)	CPU	size	従来手法 [9]			本手法		
			時間 [h]	テスト数	エラー (種類)	時間 [h]	テスト数	エラー (種類)
LLVM-GCC 4.2.1 (apple-darwin10)	A	10,000	106.8	200,000	4 (3)	12.0	3,383	33 (13)
GCC 4.2.1 (apple-darwin10)	A	10,000	105.5	200,000	4 (3)	12.0	2,772	3 (3)
GCC 4.4.1 (m32r-elf)	B	1,000	6.0	47,836	68 (4)	6.0	5,038	428 (4)
GCC 4.4.1 (arm-elf)	B	5,000	12.0	24,632	0 (0)	12.0	1,718	20 (8)
GCC 4.4.4 (i686-pc-linux)	B	3,000	12.0	38,981	0 (0)	12.0	4,505	21 (18)
GCC 4.5.3 (i686-pc-cygwin)	B	3,000	12.0	26,047	0 (0)	12.0	4,296	30 (29)
GCC 4.5.4 (i686-pc-linux)	B	5,000	12.0	23,674	0 (0)	12.0	1,977	26 (26)
GCC 4.7.2 (x86_64-apple-darwin10)	A	1,000	144.0	713,851	0 (0)	68.1	50,000	3 (3)

テストしたオプション: `-O0`, `-O3`　　CPU　A　Core 2 Duo 2.12GHz
B　Core i5-2540M 2.60GHz

表 3　実験結果 (Csmith [5] との比較)

コンパイラ (ターゲット)	時間 [h]	Csmith [5]		本手法	
		テスト数	エラー (種類)	テスト数	エラー (種類)
GCC 4.4.4 (i686-pc-linux)	12.0	9,291	1 (1)	4,179	21 (18)
GCC 4.5.3 (i686-pc-cygwin)	12.0	10,643	0 (0)	4,172	29 (28)
GCC 4.5.4 (i686-pc-linux)	12.0	12,389	1 (1)	2,236	30 (30)

テストしたオプション: なし, `-O0`, `-O3`　　CPU　Core i5-2540M 2.60GHz

表 4　実験結果 (長い式, 複数式の効果)

コンパイラ (ターゲット)	CPU	時間 [h]	エラー (種類)			
			従来手法 [9]	長い式のみ	複数式のみ	長い式+複数式
LLVM-GCC 4.2.1 (apple-darwin10)	A	12.0	0 (0)	15 (11)	15 (3)	33 (13)
GCC 4.2.1 (apple-darwin10)	A	12.0	0 (0)	1 (1)	14 (5)	3 (3)
GCC 4.4.1 (m32r-elf)	B	6.0	68 (4)	11 (1)	571 (6)	428 (4)
GCC 4.4.1 (arm-elf)	B	12.0	0 (0)	0 (0)	35 (9)	20 (8)
GCC 4.4.4 (i686-pc-linux)	B	12.0	0 (0)	2 (2)	4 (4)	21 (18)
GCC 4.5.3 (i686-pc-cygwin)	B	12.0	0 (0)	19 (19)	4 (3)	30 (29)

テストしたオプション: `-O0`, `-O3`　　CPU　A　Core 2 Duo 2.12GHz
B　Core i5-2540M 2.60GHz

トは`-O1 -foptimize-register-move -fexpensive-optimizations -fcaller-saves -frerun-cse-after-loop` であった.

3)　最小化の結果得られたプログラムを図 13 (a) に示す. プログラムは 3 演算子の式 1 つからなる. このプログラムは, GCC 4.7.2 以外の広範なバージョンと x86_64-apple-darwin10 以外の広範なターゲットでも, 最適化オプション `-O0`, `-O1`, `-Os`, `-O2`, `-O3` の全てで期待値とは異なる結果 (`t = 0`) を出力した.

```
 1: #include <stdio.h>
 2: #define OK() printf("@OK@")
 3: #define NG(fmt,val) printf("@NG@ (t = "fmt")",val)
 4:
 5: int main (void)
 6: {
 7:     unsigned int x104 = 6U;
 8:     unsigned int t = 1U;
 9:
10:     t = (((unsigned int)0U-(x104/(int)6))
11:           /(unsigned int)5U);
12:
13:     if (t == 858993459U) { OK(); }
14:     else { NG("%u", t); }
15:     return 0;
16: }
```

図 13　GCC 4.7.2 で検出されたエラープログラム (最小化後)

6. 結　　論

本稿では, C コンパイラの算術最適化のランダムテストにおける式生成の強化を提案した. 実験の結果, GCC 4.7.2 (x86_64-apple-darwin10), GCC 4.5.4 (i686-pc-linux) の不具合を検出することができた.

今後の課題には, 浮動小数型への対応, ポインタ, 配列, 構造体, 共用体, if 文, while 文への拡張などの他, それに伴う最小化の高速化等がある.

謝　　辞

本研究を行うにあたり, 多くの助言や協力を頂きました 中橋 昌俊 氏をはじめ, 関西学院大学理工学部 石浦研究室の諸氏に感謝いたします.

文　　献

[1] M. Imai, Y. Takeuchi, K. Sakanushi, and N. Ishiura: "Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP)," IPSJ Trans. System LSI Design Methodology, vol. 3, pp. 161–178 (Aug. 2010).
[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] C. Lindig: "Find a Compiler Bug in 5 Minutes," ACM International Symposium on Automated Analysis-Driven Debugging, pp. 3–12 (Sept. 2005).
[4] E. Eide and J. Regehr: "Volatiles Are Miscompiled, and What to Do about It," in Proc. ACM International Conference on Embedded Software, pp. 255–264 (Oct. 2008).
[5] X. Yang, Y. Chen, E. Eide, and J. Regehr: "Finding and Understanding Bugs in C Compilers," in Proc. PLDI, pp. 283–294 (June 2011).
[6] W. M. McKeeman: "Differential Testing for Software," Digital Technical Journal, vol. 10, no. 1, pp. 100–107 (Dec. 1998).
[7] 粟津, 石浦: "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," 信学技報, VLD2008-127 (Mar. 2009).
[8] 武田, 粟津, 石浦: "定数畳み込みを対象とした C コンパイラのランダムテスト," 情処関西支部大会, A-13 (Sept. 2009).
[9] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda: "Random Testing of C Compilers Targeting Arithmetic Optimization ," in Proc. SASIMI 2012, pp. 48–53 (Mar. 2012).
[10] 日本規格協会: "JIS X 3010 プログラム言語 C," (Oct. 1993).