一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2014-130,CPSY2014-139,RECONF2014-63(2015-01)

C コンパイラの算術最適化を対象としたテストスイート CF3

日比野佑亮†　　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, 特に定数畳み込みを対象としたテストスイート CF3 を提案する. C コンパイラ用ランダムテスト Orange3 は, GCC や LLVM/Clang の最新版の不具合を検出する強力なテストシステムだが, コンパイラによっては同じ不具合を検出するエラープログラムが多数生成されるため, 不具合の特定が効率的に行えないことがある. 本稿では, Orange3 が GCC の複数のバージョンで検出したエラープログラムを最小化すると, 式数 1, 演算数 3 程度の小さいものになることに着目し, この規模のプログラムをできる限り網羅的に集めることによりテストスイート CF3 を構築する. CF3 は, 1 つの式の形に対して変数の型や値が異なる 100 のテストケースを 1 ファイルに収容したものを, 3 演算で構成可能な全ての式の形 10,985 通りに対して生成したものである. GCC と LLVM/Clang の複数のバージョンで実験を行った結果, 同じ時間で Orange3 よりも多くのパターンのエラーを検出することができた.

キーワード　コンパイラ, テストスイート, ランダムテスト

CF3: Test Suite for Arithmetic Optimization of C Compilers

Yusuke HIBINO† and Nagisa ISHIURA†

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

Abstract　This article presents a compiler test suite “CF3,” which targets arithmetic optimization, especially constant folding, of C compilers. While Orange3 is a random test system which has detected bugs in the latest versions of GCC and LLVM/Clang, it may generate, for some compilers, many long error programs which detect the same bugs repeatedly. A new test suite CF3 is based on an obervation that most of the error programs which Orange3 found on GCCs have been reduced to small programs, each consisting of a single expression with 3 operations. CF3 consists of 10,985 files, which cover all the possible expression patterns build up with 3 operations, and each file contains 100 cases to test optimization for the expression pattern with different types and initial values of 4 variables. Experiments on GCC and LLVM/Clang show that CF3 detects more error program patterns than Orange3 within the same computation time.

Key words　Compiler, Test suite, Randomtest

1. はじめに

コンパイラはソフトウェア開発における基盤ツールであり, その不具合はソフトウェアの開発の工期や信頼性に重大な影響を及ぼす. 信頼性の確保のため, コンパイラには徹底的なテストが必要になる.

コンパイラのテストは, 正しいプログラムをエラーを出さずにコンパイルでき, さらに生成されたコードの実行結果が正しいかどうかを確認することにより行う. コンパイラのテストには, 数千から数十万本のテストプログラムから成るテストスイートを用いるが, テストスイートで発見できない不具合を検出するためにランダムテストが行われることもある.

コンパイラのテストスイートには, 目的やコンパイラの完成度等に応じて様々なものがある. C コンパイラを対象としたものとしては, 開発初期段階のコンパイラを対象とした testgen2 [1], GCC の様々な機能のテストを目的とする GCC 付属のテストスイート [2], C 言語の規格に対応しているかを検証する Plum Hall [3], 商用のコンパイラ評価サービスのテストスイート [4] 等がある.

コンパイラのランダムテストはランダムに生成したプログラムによりテストを行う手法である. テストスイートは限られた本数のプログラムでテストを行うのに対し, ランダムテストでは時間の許す限り多くのプログラムでテストを行うため, テストスイートで検出できなかった不具合を検出できることがある. コンパイラのランダムテストも, テストの対象によって様々なものが提案されている. 関数呼び出しにおける引数や返り値

等の規約を対象とする quest [5], volatile 宣言された変数を対象とする randprog [6], 配列や構造体, for 文や while 文等様々な構文を対象とする Csmith [7], 算術式の最適化を対象とした Orange3 [8] 等がある.

Orange3 は, GCC や LLVM/Clang の最新版の不具合を検出する強力なテストシステムである. しかし, 規模の大きなエラープログラムが多数生成された場合には, その解析に大きな労力を要する. エラープログラムの最小化は自動的に行うことができるが時間を要する. また, 複数のプログラムが同じ不具合を繰り返し検出することがあるため, 完成度があまり高くないコンパイラへの適用は必ずしも効率的ではない.

そこで本稿では, Orange3 が生成するプログラムに基づき, C コンパイラの算術最適化を対象としたテストスイート CF3 を提案する. Orange3 を用いて GCC の複数のバージョンで検出したエラープログラムの分析に基づいて, テストスイートの設計を行った. 検出されたエラープログラムの多くは, 最小化すると式数 1, 演算数 3 程度の小さなものになることに着目し, この規模のプログラムをできる限り網羅的に集めることによりテストスイート CF3 を構築する. CF3 は, 1 つの式のパターンに対して変数の型や値が異なる 100 個のテストケースを 1 ファイルに収容したものを, 3 演算で構成可能な全ての式のパターン 10,985 通りに対して生成したものである.

CF3 を用いて, GCC や LLVM/Clang 等のテストを行った結果, 同じ時間で Orange3 よりも多くのパターンのエラーを検出することができた.

以下, 本稿では, 2 章でコンパイラのテストについて, 3 章で Orange3 によるエラープログラムの分析について述べる. 4 章では提案する CF3 テストスイートについて述べ, 5 章で実験結果を述べる. 最後に 6 章でまとめと今後の課題を述べる.

2. コンパイラのテスト

コンパイラのテストは, テスト対象のコンパイラが対象とする言語で記述されたプログラムをコンパイルし, 実行して得られる結果が期待するものであるかを確認することにより行う. コンパイラの機能をテストするためには, 必然的に膨大な数のテストプログラムが必要となる.

2.1 C コンパイラ用テストスイート

C コンパイラ用のテストスイートには目的に応じて様々なものがある. testgen2 [1] のテストスイートは開発初期段階での使用を想定しており, 基本的な構文のコンパイルをテストする約 13,000 のファイルで構成されている. GCC 付属のテストスイート [2] は, GCC の様々な機能をテストする約 2,500 のファイルから構成され, 不具合として報告されたテストプログラムを順次追加している. Plum Hall [3] とコンパイラ評価サービスのテストスイート [4] は商用のコンパイラ用テストスイートである. Plum Hall [3] はコンパイラが ANSI/ISO C 規格に対応しているかを検証する約 750 本のテストプログラムで構成されている. コンパイラ評価サービスのテストスイート [4] は最終的な品質評価に用いられ, 約 305,000 本のテストプログラムで構成されている.

2.2 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. まずランダムにプログラムを生成し, それをテスト対象のコンパイラでコンパイルし, 実行する. 実行結果が正しくなければエラープログラムとして保存する. この操作を, 指定した回数または時間繰り返し行う.

```
while (指定時間 or 指定回数) {
    テストプログラムをランダムに生成;
    コンパイル & 実行;
    if (エラー) { 記録;}
}
エラープログラムの解析;
```

図 1 コンパイラのランダムテストの流れ

Csmith [7] は, GCC および LLVM/Clang において, 3 年間で 約 325 個の不具合を検出した. 検出した不具合の多くが tree-optimization (SSA 形式を利用した最適化) に関するものであり, 算術最適化は重要なテスト対象である. Orange3 は算術最適化を対象とするテストを行っており, エラープログラムの最小化を容易に行うことができ, エラーの原因を特定しやすい.

2.3 Orange3

Orange3 (注1) [8] は算術最適化を対象とする C コンパイラのランダムテストであり, GCC および LLVM/Clang の最新版で不具合を検出している. 複数の算術式を含むプログラムをランダムに生成し, 実行結果とあらかじめ計算した期待値を比較してエラー判定を行う. 期待値計算の過程で未定義動作を引き起こす部分式を修正することができるため, 未定義動作を含まないプログラムの生成が可能である.

Orange3 が生成するテストプログラムの例を 図 2 の (a) に示す. 806–1139 行目に算術式があり, 1141–1473 行目の部分で期待値と比較を行っている. 5–579 行目と 582–804 行目では変数の宣言を行い, 初期値を設定している.

Orange3 が生成するプログラムは, 表 1 に示した要素を含む複数の算術式で構成される. 変数の型は, 8 bit から 64 bit の符号付き/符号なしの整数型と, 32 bit から 128 bit の浮動小数点型である. 変数の初期値は, 各変数の型の最小値から最大値, 修飾子は, 「const」, 「volatile」, 「const volatile」, 「付加しない」 の 4 つからランダムに選択する. また, 使用する演算子は表に示す 18 個である.

表 1 Orange3 が生成するプログラムの要素

変数の型	(signed/unsigned) char, short, int, long, long long, float, double, long double
値の範囲	各変数の型の最小値～最大値
修飾子	const, volatile, const volatile
演算子	+, -, *, /, %, <<, >>, ==, !=, <, >, <=, >=, &&, ||, |, &, ^

2.4 Orange3 におけるエラープログラムの最小化

ランダムテストで生成されるプログラムは, 数千行～数万行

(注1) : **Orange3** は, http://github.com/ishiura-compiler/orange3 で公開されている.

表 2　testgen2 と Orange3 のエラー検出

コンパイラ	testgen2 [1]	Orange3 [8] (24h)	
	#err	#pat	(#err)
GCC-4.3	0	8	(52)
GCC-4.4	0	6	(43)
GCC-4.5	0	5	(29)
GCC-4.6	0	16	(37)
GCC-4.7	0	5	(41)
GCC-4.8	0	2	(4)
GCC-4.9	0	0	(0)

Target: x86_64-apple-darwin13

に及ぶことがあるため, 不具合の原因の特定を容易にするためには, テストプログラムの最小化が不可欠である. 最小化とは, エラープログラムにおいて, エラーに関係する箇所以外を削除してプログラムをできる限り小さくすることである.

Orange3 は, エラープログラムの最小化を自動で行う. これは, 以下の操作をそれ以上適用できなくなるまで繰り返すことにより実現している.

- 式の数を減らす.
- 定数畳込みを行って演算を減らす.
- 定数の絶対値をできる限り小さくする.
- static, volatile 等の修飾子をできる限りする削除する.
- グローバル変数をできる限りローカル変数にする.

図 2 の (b) は, (a) のプログラムを最小化した例である. 最小化によって 1475 行あったプログラムが 13 行に削減されている. 最小化前には 390 個あった算術式が, 最小化後の (b) では 9 行目の 1 個だけになっており, 1 式中の演算数も 3 つにまで減っている. また, 定数の絶対値も小さくなっており, 不要な static や volatile 等の修飾子も削除されている.

エラープログラムの最小化には, 1 つのプログラムにつき数十秒から数十分の時間がため, エラープログラムが多い場合には非常に時間がかかってしまう. また, 多くのエラープログラムを最小化した結果が同じ形のプログラムになる, 即ち, ランダムテストで同じ不具合が何度も検出されることも多い.

表 2 は, testgen2 と Orange3 で, GCC のバージョン 4.3 〜 4.9 (ターゲットはいずれも x86_64-apple-darwin13) のテストを行った結果である. 「testgen2」の「#err」は, testgen2 により検出されたエラー数を示す. testgen2 では GCC のこれらのバージョンでエラーは検出されない. 「Orange3」の「#pat」は Orange3 で 24 時間テストを行った際にエラーを検出したプログラムのパターン数, 「#err」はエラーを検出したプログラムの総数を示している. GCC の新しいバージョンのように完成度が高いコンパイラでは, 検出されるエラーが少ないため, 得られたエラープログラムを 1 つ 1 つ最小化して解析すればよい. しかし, これより低いバージョンのように, 同じパターンのエラープログラムが多数生成される場合には, テストの効率もエラー解析の効率も悪くなってしまう.

3. Orange3 によるエラープログラムの分析

本稿では, Orange3 が生成するテストプログラムを元に, テストスイートを作成することを目指す. この目的のために, まず, Orange3 が不具合を検出したエラープログラムの分析を行った.

```
0001: #include<stdio.h>
0002: #define OK() printf("@OK@")
0003: #define NG() printf("@NG@")
0004:
0005: volatile unsigned char x0 = 15U;
0006: static volatile signed int x2 = -7421;
0007: unsigned long long x3 = 428827LLU;
...                ...
0580:
0581: int main(void) {
0582:   static signed char x1 = 2;
0583:   static const signed char x5 = 1;
0584:   static volatile signed long x9 = -3533585L;
...                ...
0805:
0806:   t0 = ((x1044<<((x942%(((x1140+k2232)
0807:       >>x1885)+k2233))+k2234))*t352);
0808:   t1 = (((x494+k1627)>>x1626)%((x1113
0809:        +k1629)>>((x478>>x800)+k1628)));
0810:   t2 = (((x252%(x387-x1550))+k1729)
0811:        >>(((x995+k1727)>>x1027)+k1728));
...                ...
1138:   t390 = (t277>>((x876+k1812)>>
1139:          ((x613<<((x456/x142)+k1810))+k1811)));
1140:
1141:   if (t0 == 24U) { OK(); } else { NG(); }
1142:   if (t1 == 0) { OK(); } else { NG(); }
1143:   if (t2 == 0LU) { OK(); } else { NG(); }
...                ...
1473:   if (t390 == 0) { OK(); } else { NG(); }
1474:
1475:   return 0;
1476: }
```

(a) Orange3 が出力するプログラムの例

```
01: #include<stdio.h>
02: #define OK() printf("@OK@")
03: #define NG(fmt,val) printf("@NG@")
04:
05: int main(void) {
06:   volatile int x1430 = 31;
07:   volatile int x495 = 1;
08:
09:   t136 = (x495%(((int)2>>x1430)+(int)2147483647))
10:
11:   if (t136 == 1) { OK(); } else { NG(); }
12:   return 0;
13:}
```

(b) (a) を最小化して得られるプログラム

図 2　Orange3 のテストプログラム

表 3　事前実験に用いた C コンパイラ

コンパイラ	Version	Target
LLVM-GCC	4.2	i686-apple-darwin10
GCC	4.0	i386-apple-darwin10
GCC	4.2	i386-apple-darwin10
GCC	4.4	i686-linux-gnu
GCC	4.4	x86_64-apple-darwin13.4

分析は, 表 3 の 5 つのコンパイラに対して行った. これらのコンパイラは, 本稿で提案するテストスイートがターゲットとして想定するレベル完成度のものである.

Orange3 が生成する 1 万本のプログラムでテストを行い, 検出したエラープログラムの最小化を行った. 分析結果を表 4 に示す. 表中の「#pat」は最小化したエラープログラムのパターン数を, 「(#err)」はエラーを検出したプログラムの総数を示す. エラープログラムのパターン数は, エラープログラム中の

表 4　事前実験 (Orange3 が検出した不具合の分析結果)

コンパイラ (ターゲット)	#pat (#err)	式数 1	式数 他	op 数 2	op 数 3	op 数 4	op 数 他	変数の初期値 (頻出順)	static の割合	volatile の割合	global の割合
LLVL-GCC-4.2 (i686-apple-darwin10)	53 (150)	100 %	0 %	12 %	40 %	25 %	23 %	1, 3, 0, 2, 8, ⋯	23%	10%	18%
GCC-4.0 (i386-apple-darwin10)	79 (162)	100 %	0 %	8 %	35 %	20 %	37 %	1, 2, 0, 3, 31, ⋯	2%	6%	11%
GCC-4.2 (i686-apple-darwin10)	21 (37)	100 %	0 %	8 %	42 %	32 %	18 %	1, MAX, 0, 3, 2, ⋯	0%	10%	15%
GCC-4.4 (i686-linux-gnu)	37 (94)	100 %	0 %	15 %	46 %	22 %	17 %	1, 2, 31, 0, 8, ⋯	25%	50%	25%
GCC-4.4 (x86_64-apple-darwin13.4)	14 (37)	100 %	0 %	10 %	45 %	33 %	12 %	1, 0, 2, 8, −1, ⋯	10%	30%	23%

算術式の形が同じであれば 1 つと数えたものである．「式数」は, 最小化の結果式数が 1 あるいはその他となったエラープログラムの割合を示す．「op 数」は, 最小化したエラープログラム中の式に含まれる演算子の数が 2, 3, 4, およびそれより多いものの割合を示す．「変数の初期値」は, エラープログラム中に出現した変数の初期値のうち, 出現頻度が高いものを順に並べたものである．static, volatile, globel の割合は, エラープログラム中に現れた全変数のうち, それぞれが出現した割合を示している．

表 4 より以下が観測される．

- 最小化により全てのプログラムの式数は 1 に削減され, 約半数の不具合は 3 演算子までの算術式で検出されている．
- 変数の初期値は最小化すると, 1, 2 等の小さい値やそれぞれの変数の型の最大値になることが多い．
- static 変数, volatile 修飾, グローバル変数は, 全変数の 20 %程度に出現する．

4. C コンパイラ用テストスイート CF3

本稿では, Orange3 が生成するテストプログラムに基づくテストスイート CF3 を提案する．CF3 は, 3 演算の算術式を含むテストプログラムによって C コンパイラの算術最適化をテストするものであり, GCC のバージョン 4.2 〜 4.4 のレベルの完成度のコンパイラのテストの効率化を目的とする．

CF3 のテストプログラムの例を図 3 (a) に示す．CF3 テストスイートでは, 1 テストプログラムにつき 3 演算の組合せ 1 パターンのテストを 100 ケース収容する．このプログラムでは, 118, 129, 139, ⋯, 1308 行目のように, `((x1+x2)%x3)<x4` という式のパターンに対して, 様々な変数の型や値でテストを行っている．関数 1 つに テストケースを 1 つ 収容しており, 1313–1415 行目のメイン関数からそれを呼び出している．テストプログラムをこのような構成にすることにより, エラーを検出した際に, その算術式のパターンを容易に特定することができ, 同じパターンの算術式で何度もエラーを検出することを防げる．ヘッダファイル `test1.h` は, 不具合を検出した際のマクロ `NG` を定義するものであり, 例えば図 3 の (b) のように書ける．

テストスイート中の式および変数の詳細を表 5 に示す．式中に出現する変数の型は, 8, 16, 32, 64 bit の符号付きと符号なしの整数型を用いる．float 等の浮動小数点型は用いない．変数の初期値は, 1, 2, 31, および各整数型の最大値と最小値の 5 個を用いる．また, static 及び volatile は, 各変数に 20 %の確率で付加し, 各変数は 20%の確率でグローバル変数として宣言する．

表 5　CF 3 テストスイートの諸元

ファイル数	10,985
1 ファイル中のテスト数	100
1 テストケース中の式	1
1 式中の演算子数	3
演算子	`+, -, *, /, %, <<, ==, !=, <, <=, &, |, ^`
変数の型	`int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t`
初期値	1, 2, 31, 各整数の型の最大値, 最小値
static, volatile	各変数に 20 % で付加
グローバル変数	各変数に 20 % で出現

テストに用いる演算子は 13 通りなので, 3 演算の全組合せは 13^3 通り存在する．図 4 に示すように, 演算子 3 つの組合せ 1 つにつき 5 通りの式のパターンが構成可能なので, $13^3 \times 5 = 10{,}985$ ファイルのテストプログラムを構成する．

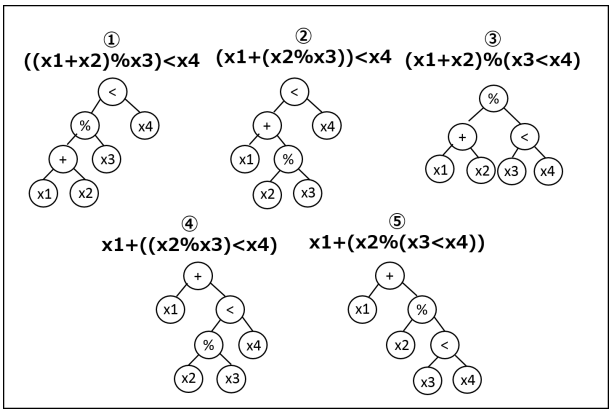

図 4　3 演算子で構成される式のパターン

1 つの式に変数は 4 つ現れるので, 変数の型 (8 個) と値 (5 個) の組合せは $8^4 \times 5^4$ の 256 万通り存在する．この 256 万通り全てを網羅するのは現実的ではないので, 1 つの式のパターンにつき, 100 通りのテストを行うことにする．このため, 1 ファイルは約 1,400 行となる．変数の型と初期値の組合せは, 図 5 のような表を用いて未選択の組合せを選択し, できる限り多くの型と初期値の組合せをカバーするようにする．

テスト生成中, 算術式の中に未定義動作が出現した場合は, そのテストケースを破棄し, 別の変数の型と値の組合せを選択して新たな算術式を生成する．

```
0001: #include<stdio.h>
0002: #include<stdint.h>
0003: #include"test1.h"
0004:
0005: static volatile int32_t t0 = 0;
0006: uint64_t x6 = 31LLU;
0007: uint64_t x10 = 31LLU;
0008: int32_t t2 = 1;
0009: int8_t x17 = 127;
0010: volatile int8_t x23 = 2;
0011:              ...
0111:
0112: void f0(void) {
0113:    int32_t x1 = -1;
0114:    volatile int64_t x2 = 31LL;
0115:    uint8_t x3 = 255U;
0116:    uint8_t x4 = 1U;
0117:
0118:    t0 = (((x1+x2)%x3)<x4);
0119:
0120:    if (t0 != 0) { NG(); }
0121: }
0122:
0123: void f1(void) {
0124:    volatile int8_t x5 = 2;
0125:    uint8_t x7 = 255U;
0126:    uint16_t x8 = 1U;
0127:    static int32_t t1 = 1514369;
0128:
0129:    t1 = (((x5+x6)%x7)<x8);
0130:
0131:    if (t1 != 0) { NG(); }
0132: }
0133:
0134: void f2(void) {
0135:    uint64_t x9 = 2LLU;
0136:    static int16_t x11 = 2;
0137:    static int64_t x12 = -1LL;
0138:
0139:    t2 = (((x9+x10)%x11)<x12);
0140:
0141:    if (t2 != 1) { NG(); }
0142: }
0143:          ...
1302: void f99(void) {
1303:    signed long long x437 = 1LL;
1304:    signed char x438 = 31;
1305:    volatile signed char x439 = 31;
1306:    signed short x440 = -1;
1307:
1308:    t99 = (((x437+x438)%x439)<x440);
1309:
1310:    if (t99 != 0) { NG(); }
1311: }
1312:
1313: int main(void) {
1314:    f0();
1315:    f1();
1316:    f2();
1317:     ...
1413:    f99();
1414:    return 0;
1415: }
```

(a) テストプログラムの例

```
#define NG() printf("@NG@ /%s", __func__)
```

(b) ヘッダファイル `test1.h`

図 3　CF3 のテストプログラム

5. 実 験 結 果

本稿で提案するテストスイート CF3 を, Orange3 のライブラリを用いて Perl スクリプトにより生成した.

CF3 を用い, 予備実験に用いたコンパイラと用いなかったコンパイラをテストする実験を行った. 結果を表 6 に示す. 動作環境は, Mac OS X, Ubuntu Linux である. 「コンパイラ (Target)」は, テスト対象のコンパイラを示し, 「time」は, テストの実行に要した時間 (秒) を示す. 「CF3 #pat」は, CF3 により検出されたエラープログラムのパターン数を示す. 「Orange3 #pat (#err)」は, CF3 と同じ時間 Orange3 を用いてテストを行った際に検出したエラープログラムのパターン数とエラープログラムの総数を示している.

予備実験に用いたコンパイラでは, 5 個中 3 個のコンパイラで Orange3 よりも多くのエラーのパターンを検出できた. また, 予備実験に用いなかったコンパイラ LLVM/Clang-3.2 と LLVM/Clang-3.3 でも Orange3 より多くのエラーのパターンを検出できた. TCC では, CF3, Orange3 ともエラーを検出できなかったが, これは TCC がほとんど最適化を行っていないためと考えられる.

	1	2	31	MIN	MAX
int8_t					
uint8_t	✓		✓		
int16_t	✓				
uint16_t					
...					

図 5　変数の型と初期値の網羅表

表 6　実 験 結 果

コンパイラ (Target)	CPU	time [s]	CF3 #pat	Orange3 #pat	(#err)
LLVM-GCC-4.2 (a)	A	10896	18	22	(42)
GCC-4.0 (a)	A	5396	11	13	(40)
GCC-4.2 (a)	A	5562	9	7	(10)
GCC-4.4 (b)	C	5893	8	7	(23)
GCC-4.4 (c)	B	6264	13	2	(6)
LLVM/Clang-3.2 (d)	C	1962	18	9	(30)
LLVM/Clang-3.3 (a)	C	2887	13	4	(5)
TCC-0.9.26 (d)	C	129	0	0	(0)
GCC-4.9 (c)	B	8988	0	0	(0)

Target　a: i686-apple-darwin10,　b: i686-pc-linux
c: x86_64-apple-darwin13,　d: i386-linux
CPU　A: Core 2 Duo 2.12GHz,　B: Core i7 3.90GHz
C: Core i5 2.60GHz

6. む す び

本稿では, C コンパイラの算術最適化を対象としたテストスイート CF3 を提案した. 提案したテストスイートを用いて GCC-4.4 や LLVM/Clang-3.3 等で実験を行った結果, 同じ時間で, Orange3 より多くのパターンのエラーを検出することができた.

テストスイートのエラー検出能力は, 1 ファイル中のテスト数や変数の初期値として用いる値の集合などに依存する. これらの様々な要素を調整し, テストスイートの効率を高めることが今後の課題である.

謝辞　本研究を進めるにあたり, 多くのご助言やご協力を頂いた関西学院大学理工学部 石浦研究室の諸氏に感謝します. 本研究は一部科学研究費補助金 (25330073) による.

文　　献

[1] 森本和志, 内山裕貴, 石浦菜岐佐, 引地信之: “C コンパイラ用テストスイート生成システム testgen2,” 情報処理学会関西支部大会, A-6 (Oct. 2010).

[2] http://gcc.gnu.org/install/test.html.

[3] http://www.plumhall.com/suites.html.

[4] http://www.jnovel.co.jp/service/compiler/.

[5] Christian Lindig: “Find a Compiler Bug in 5 Minutes,” in Proc. ACM International Symposium on Automated Analysis-Driven Debugging, pp. 3–12 (Sept. 2005).

[6] 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).

[7] X. Yang, Y. Chen, E. Eide, and J. Regehr: “Finding and Understanding Bugs in C Compilers,” in Proc. ACM PLDI, pp. 283–294 (June 2011).

[8] E. Nagai, A. Hashimoto, and N. Ishiura: “Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions,” IPSJ Trans. SLDM, vol. 7, pp. 91–100 (Aug. 2014).

[9] 日本規格協会: プログラム言語 C JIS X 3010:2003 (ISO/IEC9899:1999) (2003).