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
VLD2014-139,CPSY2014-148,RECONF2014-72(2015-01)

ランダムテストによるCコンパイラの算術最適化機会の検出

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の性能の向上を目的とした, ランダムテストによる最適化機会の検出手法を提案する. 本手法では, ランダムに生成した C プログラムと, これに C 言語レベルで算術最適化を施したプログラムをそれぞれコンパイルし, 生成されたアセンブリコードを比較することによって, コンパイラが最適化を行っているかどうかをテストする. また, volatile 変数に異なる初期値を与えた 2 つの C プログラムから生成されるアセンブリを比較することにより, volatile 変数に関する最適化が意図通り行われているかどうかのテストを行う. 本手法に基づくランダムテストシステムの実装を行った結果, GCC-5.0.0, LLVM/Clang-3.6 (ともに, 2014 年 12 月現在における最新バージョンの開発版) 等で最適化の改良の機会を検出できた.

キーワード　コンパイラ, ランダムテスト, 最適化, volatile

Detecting Missed Arithmetic Optimization Opportunities Using Random Testing of C Compilers

Atsushi HASHIMOTO† and Nagisa ISHIURA†

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

Abstract　This article presents new methods of detecting missed arithmetic optimization opportunities of C compilers by random testing. For each iteration of random testing, a C program containing arithmetic expressions is generated, from which another program, with the expressions optimized in the C program level, is also generated. Lack of optimization on the first program is detected by comparing the two assembly codes compiled from the two C programs. A method of detecting erroneous optimization or insufficient optimization involving volatile variables is also proposed, in which two random programs differing only on the initial values for volatile variables are generated, and the resulting assembly codes are compared. Random test systems implemented based on the proposed methods have detected missed optimization opportunities on several compilers, including the latest development versions of GCC-5.0.0 and LLVM/Clang-3.6.

Key words　Compiler, Randomtest, Optimization, Volatile

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, その信頼性確保は重要な課題である. コンパイラの不正コードによって開発するソフトウェアに不具合が生じた場合, その原因の特定は非常に難しいため, コンパイラが誤りのないコードを生成することは必須である. 一方で, コンパイラには高い性能, すなわち生成されるコードが速度やメモリー使用量等の点で優れていることも要求される. このため, コンパイラが意図通りの最適化を行っているかどうかも重要なテスト項目になる.

コンパイラの生成するコードが正しいことをテストする手法は多く提案されている. Plum Hall [1], SuperTest [2], GCC (GNU Compiler Collection) test suite [3], testgen2 test suite [4] 等のテストスイートは, 膨大な数のテストプログラムの集合によって, コンパイラのテストをしようとするものである. しかし, テストスイートによるテストを経てもなお不具合が潜在することがあり, そのような不具合を検出する手法として, ランダムテストが用いられる. quest [5] は関数呼び出し規約を対象とし, Csmith [6] は広範な構文の C プログラムを対象とし, Orange3 [7] は算術式を対象としている.

これらの手法はいずれも, プログラムをコンパイルして実行し, 期待通りの結果が得られるかどうかによりテストを行うため, 計算結果に直接反映されない機能をテストすることはできない. 例えば, コードが意図通りのメモリーアクセスを行っていなかったり, 最適化が行われていなくても, 上記のテストにはパスする.

randprog [8] は, volatile 宣言された変数へのアクセスが最適化により不正に除去されないかを, ランダムテストにより検査する. 最適化を適用したコードとしないコードの両方をシミュレータで実行し, メモリアクセストレースを比較することによっ

て不具合を検出する．randprog は不正な最適化を検出するが，最適化が行われているかどうかのテストは行っていない．

コンパイラの最適化をターゲットとしたテストとしては，NULLSTONE [9] がある．NULLSTONE は，約 6,500 本のテストによりコンパイラの最適化の効果を測定するテストスイートであり，40 以上の最適化処理を対象としている．しかし，限られた数のプログラムによるテストであるため，検出能力には限界があると考えられる．

そこで本稿では，算術最適化を対象に，コンパイラが正しく最適化を行っているかどうかをランダムに生成したプログラムによりテストする手法を提案する．ランダムに生成した C プログラムと，これに C 言語レベルで算術最適化を行ったプログラムをそれぞれコンパイルし，生成されたアセンブリコードを比較することによって，コンパイラが最適化を行っているかどうかのテストを行う．また，volatile 宣言された変数に異なる初期値を与えた 2 つの C プログラムから生成されるアセンブリを比較することにより，volatile 修飾された変数に関する最適化が意図通り行われているかどうかのテストを行う．

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-5.0.0, LLVM/Clang-3.6 (ともに，2014 年 12 月現在における最新バージョンの開発版) 等において，算術最適化に改善の余地があることを検出できた．

以下，2 章ではコンパイラのランダムテスト及びに Orange3 の手法を概観した後，3 章で提案する算術最適化の機会検出テストの手法を述べ，4 章で volatile 変数に関する最適化テストの手法を述べる．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. コンパイラの算術最適化のランダムテスト

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は，ランダムに生成さたテストプログラムをテスト対象のコンパイラでコンパイルして実行し，実行結果が正しいかどうかを判定するという処理を，設定した時間もしくは回数だけ行うものである．

Csmith [6] は，配列や構造体，ループ文等広範な C 言語の構文を対象としたランダムテストである．3 年間で GCC-4.5 や LLVM/Clang-2.8 等の不具合を約 325 個検出している．GCC や LLVM/Clang では，最適化処理の強化が継続的に行われているため，最新版に新たな不具合が発生する可能性がある．最適化に重点を置くコンパイラでは，最適化処理が非常に多くのパスから構成されており，これらが複雑に関係することも不具合発生の要因と考えられる．このような不具合は，限られたテストケースのテストスイートで検出することは難しいため，ランダムテストが有用となる．

Csmith が，GCC-4.4 や GCC-4.5 で検出し，原因を特定した不具合の 36 件のうち 26 件が tree-optimization (SSA 形式を利用した最適化) に関連するものであり，算術最適化は重要なテスト対象である．Orange3 (注1) [7] は，算術最適化を対象としたランダムテストであり，GCC, LLVM/Clang の最新版で不具合を検出している．

2.2 Orange3

2.2.1 テスト生成

Orange3 は，複数の算術式を含むランダムなプログラムを生成し，これをテスト対象のコンパイラでコンパイルして，実行した結果を式の期待値と比較することによりエラーを検出する．

Orange3 が生成するテストプログラムの例を図 1 に示す．21–23 行目に算術式があり，25–27 行目で期待値と比較を行っている．5–9 行目と 12–19 行目で宣言した変数に対して，ランダムな初期値を設定している．この例は，1 プログラム中の演算子の数 (式の数と 1 つの式に含まれる演算子の数の積) を 10 に設定して生成したものだが，コンパイラの完成度に応じてこの数を 1 〜 10,000 程度の間に設定してテストを行う．変数の数は，式数の 4 倍を上限に生成する．

```
01: #include <stdio.h>
02: #define OK() printf("@OK@\n")
03: #define NG(fmt,val) printf("@NG@ (t = "fmt")\n", val)
04:
05: volatile float x0 = -9.0F;
06: static signed long x2 = -1L;
07: signed long t0 = -2314269L;
08: unsigned long long t1 = 10LLU;
09: unsigned long long t2 = 22682838128721517LLU;
10:
11: int main (void)
12: {
13:   static signed short x5 = -451;
14:   volatile unsigned int x8 = 876U;
15:   unsigned long long x9 = 8614LLU;
16:   signed short x11 = 87;
17:   const volatile float x12 = -3732.0F;
18:   static volatile signed long k13 = -3717L;
19:   signed long k14 = 319664938L;
20:
21:   t0 = (x2%((x2+k14)>>((((signed int)x12)^x2)+k13)));
22:   t1 = ((x9*(x8%x5))%x11);
23:   t2 = (((signed int)(x12/x12))|(((signed int)x0)|t1));
24:
25:   if (t0 == -1L) { OK(); }else { NG("%ld", t0); }
26:   if (t1 == 6LLU) { OK(); } else { NG("%llu", t1); }
27:   if (t2 == 18446744073709551607LLU) { OK(); }
      else { NG("%llu", t2); }
28:
30:   return 0;
31: }
```

図 1　Orange3 が生成するテストプログラム例 (演算子数: 10)

テストプログラム中で使用する変数や演算子を表 1 に示す．変数の型は，符号有り/符号無し整数，浮動小数点数からランダムに選択し，修飾子と指定子をランダムに付加する．演算子は，算術演算子，論理演算子，ビット演算子，比較演算子，型変換演算子 (19 種類) からランダムに決定する．

表 1　Orange3 が生成するプログラム中の変数の型と演算

types	signed/unsigned (8, 16, 32, 64bit) floating (32, 64, 80bit)
type qualifiers	classes (static), modifiers (const, volatile)
operators	arithmetic (+, -, *, /, %) logical (&&, ||) comparison/relational (==, !=, <, >, <=, >=) bitwise (<<, >>, &, |, ^) type-conversion

テストプログラムの生成は，次の手順により行う．(1) 型や値などをランダムに決定した変数を複数生成する．(2) 式を表す解析木の形を決定し，ランダムに選択した演算子を節点に与え，ランダムに選択した変数を各葉に与える．(3) 解析木中の各節点の値をボトムアップに計算し，各式の期待値を求める．(4) 解析木からプログラムを生成する．

解析木の期待値計算の過程で未定義動作を検出すると，解析木を修正して未定義動作が起こらないようにする．例えば，符号付き整数の加算，減算，乗算でオーバーフローが起こった場合

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

には, 演算をそれぞれ減算, 加算, 除算に置換することによりこれを回避する. シフト演算の右オペランドの値が上下限値を超える場合には, 加算を挿入することにより右辺の値を調整する.

2.2.2 Orange3 によるテストプログラムの最小化

不具合を検出したプログラムをエラープログラムと呼ぶ. ランダムテストによるエラープログラムは数千行におよぶこともあり, コンパイラの不具合原因を特定するためには, プログラムの最小化 (エラーが起きる状態を残したまま, できる限りプログラムを縮約すること) が不可欠である.

最小化は, 一般的に, プログラムを縮約する変換をエラーが消えない限り適用する, という操作を繰り返すことにより行われる. Orange3 における自動最小化処理の流れを図 2 に示す. まず, プログラム中に含まれる算術式を削減する (Eliminate expressions). 一回目には二分探索 (Binary search) を, 二回目以降には線型探索法 (Linear search) を使用する. 次に, 算術式中の変数や演算を定数で置換する操作を, Top-down 方式または 3 種類の Bottom-up 方式 (In-oreder, Pre-oreder, Post-order) で行う (Expression minimization). 最後に, 変数の型や修飾子, および定数の絶対値の最小化を行う (Type and value minimization). いずれかの工程において効果があった場合には, 全体の工程を最初から繰り返す. 適用の順序によっては必ずしも最小のプログラムが生成されるわけではないが, それ以上は縮約操作が適用できないエラープログラムが得られる.

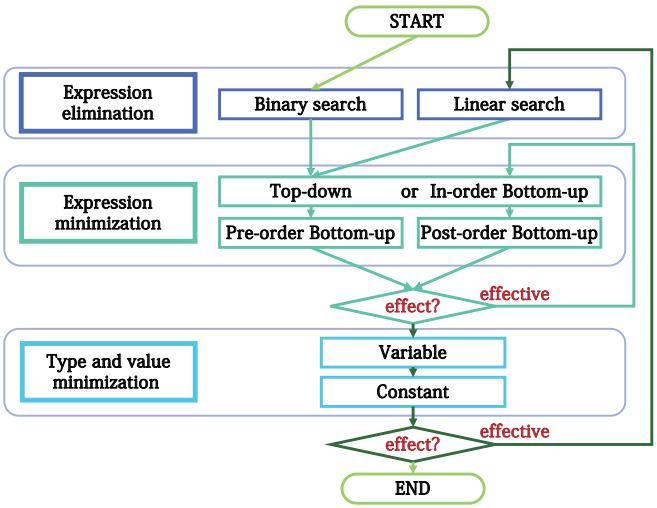


図 2 Orange3 におけるエラープログラムの最小化

3. 算術最適化の機会検出のランダムテスト

3.1 概 要

本手法は, ランダムテスト生成システムで生成する C プログラムと, C 言語レベルで可能な算術最適化を行った C プログラムから生成されるアセンブリコードを比較することにより, コンパイラが最適化を行っているかどうかのテストを行う.

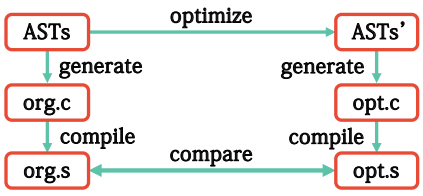


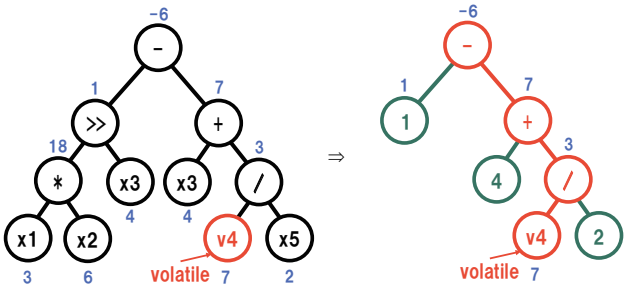
図 3 算術最適化の機会検出のランダムテストの流れ

テストの流れを図 3 に示す. まず算術式を表わす複数の解析木 (ASTs) をランダムに生成する. その解析木から C プログラム (`org.c`) を出力する一方で, 解析木に対して定数伝播/定数畳み込みの最適化処理を行い (ASTs') , そこから別のプログラム (`opt.c`) を生成する. この 2 つのプログラムをコンパイルし, 生成されたアセンブリコードの比較を行う. 一定以上の差異を検出すれば, これをエラーと判定する.

3.2 テストプログラムの生成

算術式の生成手法は Orange3 と同様である. まず解析木の形を決定し, 各節点に演算子を, 各葉に変数を, 各変数に型と初期値をランダムに与える. 期待値の計算と同時に, 未定義動作の回避処理を行う.

定数伝播/定数畳み込みは, volatile 変数の影響が及ばない範囲の部分木を縮約することにより行う. 例えば, 図 4 (a) が生成した解析木であり, `v4` が volatile 変数とする. まず変数 `x1`, `x2`, `x3`, `x5` を定数伝播によって定数に置き換える. 次に, volatile 変数の値に影響を受けない演算 `*`, `>>` に定数畳み込みを適用し, (b) を得る. (a) の解析木から (A) の C プログラムを, (b) の解析木から (B) を生成する.



(a) (b)

(A)

```
01:          int x1 = 3;
02:          int x2 = 6;
03:          int x3 = 4;
04: volatile int v4 = 7;
05:          int x5 = 2;
06: int t = ((x1 * x2) >> x3) - (x3 + (v4 / x5));
07: if ( t == -6 ) { OK(); } else { NG(); }
```

⇓

(B)

```
01:          int x1 = 3;
02:          int x2 = 6;
03:          int x3 = 4;
04: volatile int v4 = 7;
05:          int x5 = 2;
06: int t = 1 - (4 + (v4 / 2));
07: if ( t == -6 ) { OK(); } else { NG(); }
```

図 4 C プロフラムレベルでの最適化

3.3 アセンブリコードの比較

コンパイラが所望の最適化を行ったとしても, 2 つのアセンブリコード (図 3 の `org.s` と `opt.s`) が一致するとは限らない. 異なる形のプログラムは, 異なる最適化パスを通って変換されることが行われることがあるため, 同水準の最適化が行われていても, 命令の順序, 命令のアドレッシングモードやデータのサイズ等が異なるコードになることがある.

そこで本手法では, アセンブリコード中の命令数, および命令の種類毎の数を比較することによってエラー判定を行う. 「命令の種類」は, 演算は同じだが, アドレッシングモードやデータのサイズが異なる命令をグループ化したものである. 例えば x86 の命令 `addb` (8bit), `addw` (16bit), `addl` (32bit), `addq` (64bit) は同一の種類に分類する.

`org.s` と `opt.s` の命令数が異なっている場合には, 期待す

る最適化が行われていない可能性がある．このため，org.s と opt.s の命令数をそれぞれ n, n' としたとき，その比率 r_1 がエラー検出の一つ目の指標となる．

$$r_1 = \frac{n'}{n} \tag{1}$$

また，org.s と opt.s の命令の種類毎の数が異なっている場合にも，最適化が行われていない可能性がある．それぞれの命令の種類数を m, m' とし，数が一致した命令の種類数を m_u としたとき，その比率 r_2 がエラー検出の二つ目の指標となる．

$$r_2 = \frac{m_u}{\left(\frac{m+m'}{2}\right)} \tag{2}$$

本手法では，この r_1 と r_2 の相乗平均 r を求め，これがしきい値を下回るとき，エラーと判定する．

$$r = \sqrt{r_1 \cdot r_2} \tag{3}$$

3.4 プログラムの最小化

本手法におけるエラープログラムの自動最小化は，Orange3 と同様の手順で行う．解析木レベルでテストプログラムを縮約する変換を行う度に，org.c と opt.c を生成し，コンパイラが出力するアセンブリコードの比較を行う．式 (3) の r がしきい値を下回れば変換を適用し，しきい値を上回れば変換を取り消す．この操作を，可能な全ての変換がそれ以上適用できなくなるまで繰り返し行う．

4. Volatile 修飾された変数に関する最適化のランダムテスト

2 つの C プログラムから生成されるアセンブリを比較することにより，volatile 修飾された変数に関する最適化が意図通りに行われているかどうかをテストすることもできる．

volatile 修飾された変数は，他のスレッドで実行されるプログラムや，外部のデバイス等，当該プログラム以外により読み書きされる可能性のある変数であり，当該プログラムに記述されている通りにロードやストアが行われなければならない．このためコンパイラは，

（1） volatile 修飾された変数のロード/ストアを削除する最適化を行ってはならない

が，その一方で，

（2） volatile 修飾された変数の影響を受けない部分の最適化は行わなければならない

ことにも留意しなければならない．

本手法では，volatile 変数の初期値が異なる 2 つのプログラムを生成し，そこから生成されるアセンブリコードを比較することにより，コンパイラが volatile 変数に関する最適化を正しく行っているかどうかのテストを行う．

テストの流れを図 5 に示す．まず，Orange3 と同様のランダムなプログラム vol1.c を生成し，次に volatile 変数の初期値があらかじめ設定した値だけ異なる vol2.c を生成する．値の変更によって未定義や，処理系動作を引き起こす計算式が生成される可能性があるが，期待値の再計算は行わない．生成されたアセンブリ vol1.s と vol2.s の比較は 3 章と同じ手法で行う．

Volatile 変数の値はいつ外部から書き換えられるか分からないため，その初期値に依存する最適化は行えないはずである．すなわち，vol1.c と vol2.c を比較すると，初期値を計算/設定するコード以外は同じになるはずであり，これを検査することにより，上記 (1) の不正最適化，および (2) の最適化機会の逸失を検出することができる．

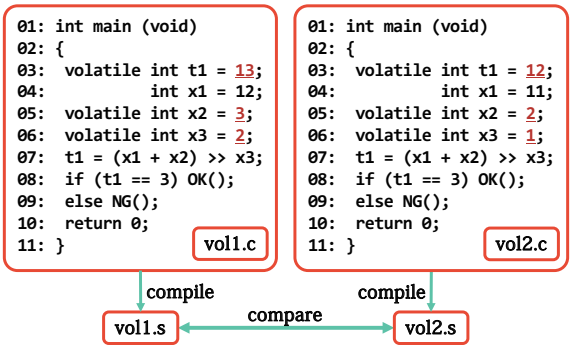


図 5 Volatile 変数に関する最適化のランダムテストの流れ

5. 実装と実験結果

5.1 実　　装

提案手法に基づくランダムテストシステムを，Orange3 ライブラリを利用し，Perl 5.20.1 で実装した．本システムは，Windows Cygwin, Mac OS X, Ubuntu Linux 等の環境で動作する．

5.2 最適化機会検出テストの実験結果

GCC, LLVM/Clang, SunCC, IntelCC の 10 バージョンのコンパイラについて，本手法によるテストを 12 時間ずつ行った．1 本のテストプログラム中に含まれる演算子の数は 500 とし，エラー判定の指標 r は 80% とした．プロセッサは Intel(R) Core(TM) i7–4930K 3.40GHz, RAM は 15.6GiB である．拡張命令を生成する浮動小数点演算や，MOV 命令を複数生成するケースある剰余算 (%) 演算は，アセンブリの比較が難しいため除外した．

表 2 最適化機会検出テストの実験結果

Compiler (Target)	Opt	#Test	#Missed
GCC-4.4.7 (A)	-O3	53,374	30,852
GCC-4.6.4 (A)	-O3	53,113	27,416
GCC-4.7.3 (A)	-O3	53,237	24,222
GCC-4.8.2 (A)	-O3	54,000	20,119
GCC-5.0.0(注2) (B)	-O3	49,574	17,925
LLVM/Clang-2.8 (B)	-O3	61,008	1,110
LLVM/Clang-3.3 (B)	-O3	60,495	1,105
LLVM/Clang-3.6(注3) (B)	-O3	55,804	434
SunCC-5.12 (C)	-O5	33,552	3,994
IntelCC-15.0.1 (D)	-O3	45,847	22,187

time: 12 (h), size: 500, r: 80 (%)
CPU:Intel(R) Core(TM) i7–4930K 3.40GHz, RAM:15.6GiB
A:x86_64-pc-linux-gnu, B:x86_64-unknown-linux-gnu,
C:linux-i386, D:Intel(R)-64

表 2 に示す「Opt」はテストした最適化オプション，「#Test」は実行したテストプログラムの数，「#Missed」は最適化の機会を検出した数である．全てのコンパイラで最適化の機会を検出することができた．

図 6 は，LLVM/Clang-3.6(注4) (-O3 オプション) で検出したエラープログラム (851 行) を最小化した結果である．(a) の org.c がオリジナルの C プログラム，opt.c が定数伝播/定数畳み込みの処理を行った C プログラムである．2 つのプログラムの違いは 06 行目の a が初期値 1 で置き換えられていること

(注2)：gcc version 5.0.0 20141010 (experimental)
(注3)：clang version 3.6.0 (trunk 217856)
(注4)：clang version 3.6.0 (trunk 217334) (x86-unknown-linux-gnu)

org.c

```
01: int main (void)
02: {
03:  static   int a = 1;
04:  volatile int b = 1;
05:
06:  int c = (a / 0+b) >= 2;
07:
08:  if (c == 0);
09:  else __builtin_abort();
10:
11:  return 0;
12: }
```

opt.c

```
01: int main (void)
02: {
03:  static   int a = 1;
04:  volatile int b = 1;
05:
06:  int c = (1 / 0+b) >= 2;
07:
08:  if (c == 0);
09:  else __builtin_abort();
10:
11:  return 0;
12: }
```

(a) 2 つのテストプログラム

org.s (LLVM/Clang-3.6 -O3)

```
01:   .text
02:   .file   "org.c"
03:   .globl  main
04:   .type   main,@function
05: main:           # @main
06:   .cfi_startproc
07: # BB#0:         # %entry
08:   pushq   %rax
09: .Ltmp0:
10:   .cfi_def_cfa_offset 16
11:   movl    $1, 4(%rsp)
12:   movl    4(%rsp), %eax
13:   leal    1(%rax), %ecx
14:   cmpl    $2, %ecx
15:   ja      .LBB0_2
16: # BB#1:         # %entry
17:   cmpl    $2, %eax
18:   jge     .LBB0_3
19: .LBB0_2:        # %if.end
20:   xorl    %eax, %eax
21:   popq    %rdx
22:   retq
23: .LBB0_3:        # %if.else
24:   callq   abort
25: .Ltmp1:
26:   .size   main,.Ltmp1-main
27:   .cfi_endproc
```

opt.s (LLVM/Clang-3.6 -O3)

```
01:   .text
02:   .file   "opt.c"
03:   .globl  main
04:   .type   main,@function
05: main:           # @main
06:   .cfi_startproc
07: # BB#0:         # %entry
08:   movl    $1, -4(%rsp)
09:   movl    -4(%rsp), %eax

10:   xorl    %eax, %eax

12:   retq
13: .Ltmp0:
14:   .size   main,.Ltmp0-main


15:   .cfi_endproc
```

(b) 2 つのプログラムから生成したアセンブリコード

org-gcc.s (GCC-4.8.2 -O3)

```
01:   .file    "org.c"
02:   .section .text.startup, "ax",@progbits
03:   .p2align 4,,15
04:   .globl   main
05:   .type main, @function
06: main:
07: .LFB0:
08:   .cfi_startproc
09:   movl    $1, -4(%rsp)
10:   movl    -4(%rsp), %eax
11:   xorl    %eax, %eax
12:   ret
13:   .cfi_endproc
14: .LFE0:
15:   .size    main, .-main
16:   .ident   "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
17:   .section .note.GNU-stack, "",@progbits
```

(c) GCC にて生成したアセンブリコード

図 6　最適化機会検出テストで検出したエラー (LLVM/Clang-3.6)

だけなので, 同じコードが生成されると期待される. それぞれをコンパイルした結果が, (b) の `org.s`, `opt.s` である. `org.s` には `opt.s` に比べて冗長な命令が含まれている. 比較のために, `org.c` を GCC-4.8.2 (`-O3` オプション) でコンパイルして得られるコードを (c) の `org-gcc.s` に示すが, (b) の `opt.s` とほぼ一致する. これより, `org.c` のコードに対し, LLVM/Clang の最適

org.c

```
01: int main (void)
02: {
03:                 int a = -1;
04:  volatile unsigned b = 1U;
05:                 int c =  1;
06:
07:  c = a+972195718 >> 1LU<=b;
08:
09:  if (c == 486097858);
10:  else __builtin_abort();
11:
12:  return 0;
13: }
```

opt.c

```
01: int main (void)
02: {
03:                 int a = -1;
04:  volatile unsigned b = 1U;
05:                 int c =  1;
06:
07:  c = 972195717 >> 1LU<=b;
08:
09:  if (c == 486097858);
10:  else __builtin_abort();
11:
12:  return 0;
13: }
```

(a) 2 つのテストプログラム

org.s

```
        .
        .
        .
01: main:
02: .LFB0:
03:   .cfi_startproc
04:   subq    $24, %rsp
05:   .cfi_def_cfa_offset 32
06:   movl    $1, 12(%rsp)
07:   movl    12(%rsp), %eax
08:   testl   %eax, %eax
09:   movl    $972195717, %eax
10:   setne   %cl
11:   sarl    %cl, %eax
12:   cmpl    $486097858, %eax
13:   jne     .L5
14:   xorl    %eax, %eax
15:   addq    $24, %rsp
16:   .cfi_remember_state
17:   .cfi_def_cfa_offset 8
18:   ret
19: .L5:
20:   .cfi_restore_state
21:   call    abort
22:   .cfi_endproc
        .
        .
        .
```

opt.s

```
        .
        .
        .
01: main:
02: .LFB0:
03:   .cfi_startproc
04:   subq    $24, %rsp
05:   .cfi_def_cfa_offset 32
06:   movl    $1, 12(%rsp)
07:   movl    12(%rsp), %eax
08:   testl   %eax, %eax

09:   jne     .L2
10:   call    abort
11: .L2:
12:   xorl    %eax, %eax
13:   addq    $24, %rsp
14:   .cfi_def_cfa_offset 8
15:   ret


16:   .cfi_endproc
        .
        .
        .
```

(b) 2 つのプログラムから生成したアセンブリコード

図 7　最適化機会検出テストで検出したエラー (GCC-4.8.4)

化には改良の余地があると考えられる. この件は, LLVM/Clang の Bugzilla に報告しており(注5), これによりコンパイラの修正が行われた.

図 7 は, GCC-4.8.4(注6) (`-Os` オプション) にて検出したエラープログラム (1,121 行) を最小化した結果である. (a) の `org.c` がオリジナルの C プログラム, `opt.c` が定数伝播/定数畳み込みの処理を行った C プログラムである. 2 つのプログラムの違いは 07 行目の `a+972195718` が計算されていることだけなので, 同じコードが生成されると期待される. それぞれをコンパイルした結果が, (b) の `org.s`, `opt.s` である. `org.s` では, `abort` を呼び出す分岐命令が存在し, 最適化に改良の余地があると考えられる. この件は, GCC の Bugzilla に報告しており(注7), `org.s` と `opt.s` とでシフト演算の最適化パスが異なっているためとの分析がなされた.

ただし, 図 6 や図 7 のような簡潔なプログラムが, 表 2 の「#Missed」の全てのプログラムに対して得られたわけではない. 大きなエラープログラムではアセンブリが異なるが, 少しでも縮約を試みるとアセンブリの相違がしきい値以下になるというケースが多く見られた. アセンブリの比較方法や最小化の

(注5) : `http://llvm.org/bugs/`; bug 20916
(注6) : gcc version 4.8.4 20140622 (prerelease) (x86-unknown-linux-gnu)
(注7) : `http://gcc.gnu.org/bugzilla/`; bug 61839

手順にはまだ改良の余地があると考えられる.

5.3 volatile 変数に関する最適化テストの実験結果

本手法によるテストの実験結果を表 3 に示す.「#Error」は, 最適化のミスコンパイルの検出数であり, この手法では volatile 変数に関する誤りコードを検出することはできなかった. しかし, 表中の「#Missed」に示す件数の最適化機会を検出することができた.

表 3 Volatile 変数に関する最適化テストの実験結果

Compiler (Target)	Opt	#Test	#Error	#Missed
GCC-4.4.7 (A)	-O3	45,795	0	5
GCC-4.6.4 (A)	-O3	45,204	0	15
GCC-4.7.3 (A)	-O3	44,157	0	13
GCC-4.8.2 (A)	-O3	44,986	0	12
GCC-5.0.0(注2) (B)	-O3	40,510	0	10
LLVM/Clang-2.8 (A)	-O3	47,582	0	0
LLVM/Clang-3.3 (A)	-O3	46,843	0	0
LLVM/Clang-3.6(注3) (B)	-O3	42,973	0	3
SunCC-5.12 (C)	-O5	32,260	0	253
IntelCC-15.0.1 (D)	-O3	36,363	0	16,952

表 2 と同様

図 8 は, GCC-5.0.0(注8) (`-Os` オプション) で検出したエラープログラム (691 行) を最小化した結果である. (a) の `vol1.c` がオリジナルの C プログラム, `vol2.c` が volatile 変数の初期値を変更した C プログラムである. 2 つのプログラムの違いは 05 行目の volatile 変数 `c` の初期値が異なるだけで, 以降に `c` は出現しないため同じコードが生成されると期待される. それぞれをコンパイルした結果が, (b) の `vol1.s`, `vol2.s` である. `vol1.s` には, `vol2.s` に比べて冗長な命令が含まれており, 最適化に改良の余地があると考えられる. この件は, GCC の Bugzilla に報告しており(注9), これによりコンパイラの修正が行われた.

6. 結　　論

本稿では, ランダムテストにより C コンパイラの算術最適化機会を検出する手法を提案した. 実験の結果, GCC-4.10.0, LLVM/Clang-3.6, SunCC, IntelCC の最適化の機会を検出することができ, コンパイラの性能向上に有用なランダムテストを考案できた.

しかし, 検出した全てのプログラムを自動で十分に最小化することはできなかった. アセンブリコードの比較方法や最小化手法の改良が重要な課題として挙げられる. また, 本手法を定数伝播, 定数畳み込み以外の最適化に拡張することも今後の課題である.

謝　　辞

本研究を行うにあたり, 多くの御助言や御協力を頂きました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. 本研究は一部科学研究費補助金 (25330073) による.

vol1.c

```
01: int main (void)
02: {
03:           long a = -1L;
04:  volatile long b =  0L;
05:  volatile long c =
      0x100000000L;
06:
07:  a = (1+b >> 63 << 1) /
      0x100000000L;
08:
09:  if (a == 0L);
10:  else __builtin_abort();
11:
12:  return 0;
13: }
```

vol2.c

```
01: int main (void)
02: {
03:           long a = -1L;
04:  volatile long b =  0L;
05:  volatile long c = 0L;
06:
07:  a = (1+b >> 63 << 1) /
       0x100000000L;
08:
09:  if (a == 0L);
10:  else __builtin_abort();
11:
12:  return 0;
13: }
```

(a) 2 つのテストプログラム

vol1.s

```
        .
        .
        .
01: main:
02: .LFB0:
03:   .cfi_startproc
04:   subq   $24, %rsp
05:   .cfi_def_cfa_offset 32
06:   movabsq $4294967296, %rcx
07:   movq   $0, (%rsp)
08:   movq   %rcx, 8(%rsp)
09:   movq   (%rsp), %rax
10:   incq   %rax
11:   sarq   $63, %rax
12:   addq   %rax, %rax
13:   cqto
14:   idivq  %rcx
15:   testq  %rax, %rax
16:   je  .L2
17:   call   abort
18: .L2:
19:   xorl   %eax, %eax
20:   addq   $24, %rsp
21:   .cfi_def_cfa_offset 8
22:   ret
23:   .cfi_endproc
24: .LHOTE0:
        .
        .
        .
```

vol2.s

```
        .
        .
        .
01: main:
02: .LFB0:
03:   .cfi_startproc
04:   movq   $0, -24(%rsp)
05:   movq   $0, -16(%rsp)
06:   movq   -24(%rsp), %rax
07:   xorl   %eax, %eax
08:   ret
09:   .cfi_endproc
10: .LHOTE0:
        .
        .
        .
```

(b) 2 つのプログラムから生成したアセンブリコード

図 8 Volatile 変数に関する最適化テストで検出したエラー (GCC-5.0.0)

文　　献

[1] Plum Hall, Inc.: The Plum Hall Validation Suite for C (online), `http://www.plumhall.com/stec.html` (accessed 2014-12-10).

[2] ACE Associated Computer Experts: SuperTest Compiler Test and Validation Suite (online), `http://www.ace.nl/compiler/supertest.html` (accessed 2014-12-10).

[3] Free Software Foundation, Inc.: Installing GCC: Testing (online), `http://gcc.gnu.org/install/test.html` (accessed 2014-12-10).

[4] T. Fukumoto, K. Morimoto, and N. Ishiura: "Accelerating regression test of compilers by test program merging," in Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies, pp. 42–47 (2012).

[5] C. Lindig: "Find a Compiler Bug in 5 Minutes," ACM Int. Symp. on Automated Analysis-Driven Debugging, pp. 3–12 (Sept. 2005).

[6] 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).

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

[8] E. Eide and J. Regehr: "Volatiles Are Miscompiled, and What to Do about It," in Proc. ACM Int. Conf. on Embedded Software, pp. 255–264 (Oct. 2008).

[9] Nullstone Corporation: NULLSTONE for C (online), `http://www.nullstone.com/` (accessed 2014-12-10).

[10] 日本規格協会: "プログラム言語 C JIS X 3010:1993 (ISO/IEC9899:1990)," (Oct. 1993).

(注8): gcc version 5.0.0 20141215 (experimental) (x86-unknown-linux-gnu)

(注9): `http://gcc.gnu.org/bugzilla/`; bug 64322