アセンブリコードおよび実行時間の比較に基づく C コンパイラの最適化のランダムテスト

北浦 幸太[†1]　石浦 菜岐佐[†1]

概要：本稿では，ランダムプログラムによる C コンパイラの最適化性能テストのためのアセンブリコード比較法を提案する．コンパイラには，信頼性とともに高い最適化の性能が要求されるため，意図通りの最適化が行われているかどうかはコンパイラの重要なテスト項目になる．本稿の手法では，ランダムに生成した C プログラムを 2 つの異なるコンパイラでコンパイルし，生成されるコードを比較することにより一方の最適化に不足があればこれを検出する．アセンブリコードの比較は，アセンブリコード中の不一致部分対の抽出と不一致命令の重み和に基づいて行う．さらに，アセンブリコードに差異があると判定したプログラムは，実行時間の計測を行って性能を比較する．本手法をランダムテストシステム Orange4 に実装した結果，GCC-8.0.0 (2017 年 5 月時点における最新バージョンの開発版) において，最適化の不具合を 1 件検出することができた．

キーワード：コンパイラ，最適化，ランダムテスト，アセンブリコード，実行時間計測

Random Testing of C Compiler Optimization Performance Based on Comparison of Assembly Codes and Execution Time

Kitaura Kota[†1]　Ishiura Nagisa[†1]

Abstract: This article proposes a method of comparing assembly codes for performance testing of compiler optimizers. We test compiler optimizers based on differential random testing, in which randomly generated programs are compiled by two different compilers and resulting pairs of assembly codes are compared to see if one of the compilers fails to perform expected optimization. In our method, discrepant pairs of code sections in the assembly codes are identified, and then the sums of the weights of discrepant instructions in the sections are computed. If siginificant differences are detected, the codes are further compared by measuring their execution time. A test system has been implemented on top of random test system Orange4, which has successfully detected a regression in the optimizer of a development version of GCC-8.0.0 (latest as of May, 2017).

Keywords: compiler, optimization, random test, assembly code, measurig execution time

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，高い信頼性が要求される．さらに，コンパイラには処理速度，メモリ使用量あるいは消費電力等に関して優れたコードを生成することも要求される．構文解析部や中間言語生成部は一旦開発されれば頻繁に更新されることはないが，最適化に関しては継続的に性能向上のための改良が行われるため，ここに不具合が発生する可能性が生じる．その際，コンパイラが正しいコードを生成しているかはもちろんのこと，コンパイラが意図通りの最適化を行っているかも重要なテスト項目となる [1].

コンパイラの最適化の性能をテストする一手法として，

[†1] 現在，関西学院大学
Presently with Kwansei Gauin Uniersity, 2–1 Gakuen, Sanda, Hyogo, 669–1337, Japan

ⓒ 2017 Information Processing Society of Japan

岩辻ら [2] は差分ランダムテストに基づく手法を提案している．この手法では，ランダムに生成したプログラムを 2 つのコンパイラでコンパイルし，生成されるアセンブリコードを比較することによって一方の最適化不足を検出する．しかし，その比較は命令数に基づいて行っているため，見逃しや誤判定が多く発生するという課題があった．また，この手法では分岐のないプログラムだけを扱っているが，プログラム中に条件文，ループ文，関数呼び出し等が存在すると，アセンブリコードの命令数のみで最適化不足を検出することは困難になる．

そこで本稿では，アセンブリコードの不一致部分に着目した比較，および実行時間の計測に基づく C コンパイラの最適化性能テスト手法を提案する．アセンブリコードの比較は，アセンブリコード中の不一致部分の抽出と不一致命令の重み和に基づいて行うことにより，精度の高いアセンブリコードの差分判定を実現する [3]．さらに，アセンブリコードの差分を検出した場合に実行時間の計測を行うことにより，誤判定を排除する．

本手法に基づくランダムテストシステムを Perl5 で実装した結果，GCC-8.0.0 (実験時における最新バージョン) で 1 件のコンパイラの最適化の不具合を検出し，開発者に報告した．

2. コンパイラの最適化のランダムテスト

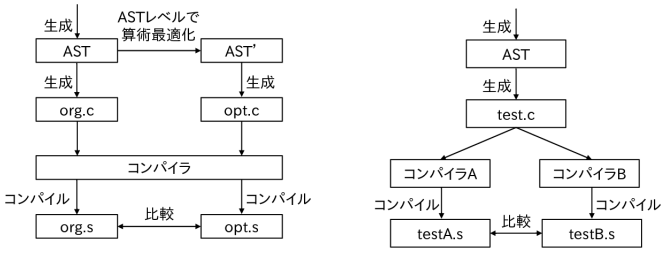
2.1 コンパイラの最適化のテスト

C コンパイラの最適化の効果を測定するテストスイートとしては NULLSTONE [4] がある．NULLSTONE は，40 以上の最適化処理を対象とした約 6,500 本のテストより成るが，テストプログラム数が限られるため，その検出能力には限界があると考えられる．

コンパイラの最適化のランダムテストとしては randprog [5] がある．randprog では volatile 変数に関するコードがコンパイラの最適化によって不正に除去されないかをランダムに生成したプログラムにより検査する．しかし，この手法では最適化による性能向上が意図通りに行われているかのテストは行っていない．

橋本ら [6] はランダムテストによる C コンパイラの算術最適化の不具合を検出する手法を提案している．この手法は，図 1(a) に示すように，ランダムに生成した C プログラム (org.c) とそれに算術最適化処理を行ったプログラム (opt.c) をコンパイルして得られるアセンブリコード (org.s, opt.s) を比較することにより，期待通りの最適化が行われているかどうかをテストするというものである．また，岩辻ら [2] は，差分テスト [7] に基づく C コンパイラの最適化性能のランダムテスト手法を提案している．図 1(b) に示すように，ランダムなテストプログラム (test.c) を 2 つの異なるコンパイラでコンパイルして得られるアセンブリコード (testA.s, testB.s) を比較することにより，一方の最適化の不足を検出する．同じコンパイラの異なるバージョンを用いれば，リグレッションテスト (バージョンアップによる不具合の混入のテスト) を行うことができる．

ランダムテストに用いる C プログラムは数百行から数千行に及ぶため，コンパイラの不具合が検出されても，原因の特定は困難である．そこで，最適化の不具合を残したまま，可能な限りテストプログラムを小さくする「最小化」が行われる．これは，図 2 のように，プログラムの一部を縮約する操作 (文の削除や式の簡単化等) を不具合が検出される限り繰り返すというものであり，最終的に，⑦のようにそれ以上縮約すると不具合が検出されないというプログラムが得られる．



(a) 等価プログラム法 [6]　　(b) 差分法 [2]

図 1: コンパイラの最適化性能のランダムテスト

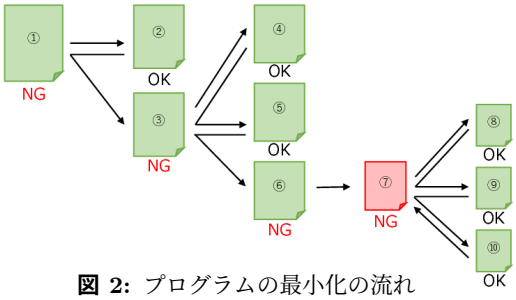


図 2: プログラムの最小化の流れ

2.2 ランダムプログラムの生成

文献 [2][6] の手法では，ランダムプログラムの生成に Orange3 [8] を利用している．Orange3 は， C コンパイラの算術最適化を対象としたランダムテストシステムであり，生成されるテストプログラムは算術式を右辺に持つ複数の代入文から成る．Orange3 では，ランダムな算術式を生成するのと同時に各部分式の正確な期待値を計算する．式生成の過程で未定義動作 (ゼロ除算や符号付き整数のオーバーフロー等) を検出すると，これを回避するように式を修正できるため，未定義動作を全く含まないランダムプログラムを生成できる．Orange3 は算術式の最適化のテストに特化しているため，条件文，ループ，関数呼び出し等

ⓒ 2017 Information Processing Society of Japan

の制御構文を含むプログラムを生成しない．これに対し，Orange4 [9] は，プログラムの等価変換に基づいてランダムなプログラムを自動生成するシステムであり，Orange3 よりも広範な文法を扱える．最新バージョンでは，if 文および for 文を含むプログラムを生成できる．Orange4 が生成するテストプログラムの例を図 3 に示す．5 ～ 8 行目に global 変数の宣言, 11 ～ 24 行目に local 変数の宣言がある．それらの変数を用いた式と文が 26 ～ 34 行目にあり，36 行目で計算結果が正しいかどうかの判定を行っている．テストに使用するプログラムの規模はコンパイラの完成度に合わせて調節するが，数百行から数千行である．

```
 1: #include <stdio.h>
 2: #define OK()
 3: #define NG(fmt,val) __builtin_abort()
 4:
 5: const volatile signed int x9 = -59;
 6: signed long long x10 = 8198LL;
 7: signed short x11 = 18332;
 8:
 9: int main (void)
10: {
11:   static unsigned long long x0 = 7LLU;
12:   static const volatile signed long x1 = 0L;
13:   signed char x2 = 22;
14:   unsigned long x3 = 0LU;
15:   static unsigned char x4 = 29U;
16:   static const signed char x5 = -1;
17:   static signed int x6 = 123621;
18:   signed int x7 = -7293637;
19:   unsigned long long t0 = 46LLU;
20:   signed long t1 = 297271L;
21:   unsigned long x8 = 102005473280LU;
22:   signed long long t2 = 0LL;
23:   signed char t3 = 3;
24:   signed int i;
25:
26:   for( i = x9*x6; i < x5+x5; i -= x7+x3 ) {
27:     t0 = x3|x0;
28:     t1 = x10||x11;
29:     if( x1<<x1 ) {
30:       t2 = x8>>x4;
31:     }
32:     else {
33:       t3 = x2|x2;
34:     }
35:
36:   if (t3 == 22) { OK(); }
37:   else { NG("%d", t6); }
38:   return 0;
39: }
```

図 3: Orange4 が生成するテストプログラムの例

2.3 アセンブリコードの比較

文献 [2][6] の手法では，同一の，あるいは等価な C プログラムから生成される 2 つのアセンブリコードに対し，その性能に差があるかどうかを判定する．例えば, 図 4 の例では，明確に最適化の性能に差があると判定できるが，必ずしもコードに差があると判定できない場合も生じる．コードを実行すれば速度性能は測定できるが，実行時間にどの程度差があれば最適化の不具合であるかという明確な基準は存在しないため，最終的にはコンパイラの開発者の判断となる．本稿では，明確に性能差があると考えられるものだけを見逃しなく判定することを目標とする．

文献 [2] では，アセンブリコード中の命令数の割合で最適化の不足を判定している．また，文献 [6] では命令の種類まで考慮して命令数の比較を行っているが，最適化不足と判定されたプログラムを目視確認すると, 図 5 のように最適化不足とは断定できない場合が多くある．本稿では，このようなものを「誤判定」と呼ぶ．また，判定の閾値の設定によっては最適化の不足に見逃しが生じることも課題として挙げられる．

さらに，算術式のみのプログラムであればアセンブリコードの長さだけで最適化不足を判定できるが，条件文やループ等を含むプログラムは最適化によってコードのサイズが増大することもあるため，命令数のみに基づいて最適化の不具合を判定することは難しいと考えられる．

(1)GCC-5.2.1.s	(2)GCC-6.0.0.s

main:
  movq $1, -8(%rsp)
  movq -8(%rsp), %rax
 	
  xorl %eax, %eax
 	
  ret
	main:
	  subq $24, %rsp
	  xorl %edx, %edx
	  movl $425, %eax
	  movq $1, 8(%rsp)
	  movq 8(%rsp), %tri
	  shrq $63, %rsi
	  subq $1, %rsi
	  divl %rsi
	  xorl %edx, %edx
	  leaq 3(%rax), %rcx
	  movl $1, %eax
	  divq %rcx
	  testq %rax, %rax
	  jne .L5
	  xorl %eax, %eax
	  addq $24, %rsp
	  ret
	.L5:
	  call abort

図 4: 正判定

(1) GCC-5.2.1.s	(2) GCC-6.0.0.s

main:
  subq $24, %rsp
  movl $1, 8(%eax)
  movl $1, 12(%rsp)
  movl 8(%rsp), %edx
  shrl $31, %eax
 	
  andl $1, %eax
  imulq %rdx, %rax
  testq %rax, %rax
  jbe .L5
  xorl %eax, %eax
  addq $24, %rsp
  ret
.L5:
  call abort
	main:
	  subq $24, %rsp
	  movl $1, 8(%eax)
	  movl $1, 12(%rsp)
	  movl 8(%rsp), %edx
	  shrl $31, %eax
	  movl %eax, %ecx
	  movl %edx, %eax
	  andl $1, %eax
	  imulq %rdx, %rax
	  testq %rax, %rax
	  jbe .L5
	  xorl %eax, %eax
	  addq $24, %rsp
	  ret
	.L5:
	  call abort

図 5: 誤判定

3. 不一致部分対の抽出に基づくアセンブリコードの比較

3.1 概要

命令数に基づくアセンブリコードの比較法では，一方のコードの一箇所で最適化不足に起因する命令数の増加が

ⓒ 2017 Information Processing Society of Japan

あっても，テストプログラムが大きくてアセンブリコードの命令数が多ければ，その影響の検出は難しく，見逃しが生じてしまう．逆に，コンパイラのデータ転送や加減算に対するコード生成法の違いで，性能差は無いにもかかわらず命令数が異なるコードが生成され，誤判定が生じることがある．本稿では，この課題を解決するために，

(1) アセンブリコード中の不一致部分のみを抽出して比較する
(2) 不一致部分からさらに不一致命令を抽出し，その重み和で比較を行う

ことにより，見逃しを減らし，精度の高いアセンブリコードの差分判定を図る．

3.2 アセンブリコードの比較

(1) アセンブリコード中の不一致部分対の抽出

図 6 に示すように，2 つのアセンブリコードに対し，一致部分と不一致部分を分離し，不一致部分に対してのみコードの性能差を調べる．一致部分は，命令とオペランドが k 命令以上 (k は 7 程度を想定している) 連続して一致しているかどうかにより決定する．不一致部分の対 (S_1, S'_1), (S_2, S'_2), $\cdots$ に対して性能差があるかどうかを調べ，このうちの一対にでも有意差があれば，2 つのアセンブリコードに差異があると判定する．

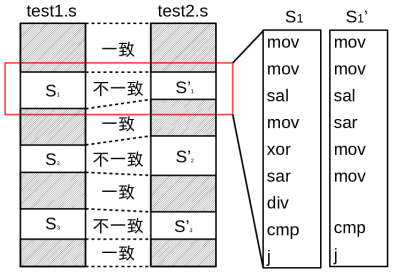


図 6: アセンブリコード中の不一致部分対の抽出

(2) 不一致命令の重み和での比較

(1) で抽出した不一致部分対の比較は，その中から不一致命令を抽出し，重み和を算出することにより行う．

図 7 の不一致部分対 S_1 と S'_1 において， S_1 の xor 命令と div 命令，および S'_1 の mov 命令 (下線を付したもの) が不一致である．不一致部分対 S_i と S'_i における不一致命令とは，S_i 中の命令と同じ命令 (一致命令) が S'_i 中にあれば両者を削除するという操作を可能な限り繰り返して得られる命令である．一致命令の検索の際に命令の順序は考慮しない．これは，コンパイラの最適化による命令の順序変更の影響を受けないようにするためである．

各命令には実行サイクル数に相当する重みを定義しておき，不一致命令に対してその重み和を算出する．図 7 の例では， ALU 命令や mov 命令の重みを 1，div 命令の重みを 45 としており，S_1 と S'_1 の重み和はそれぞれ 46，1 となる．この比が閾値未満であれば，S_1 と S'_1 には性能の差異があると判定する．

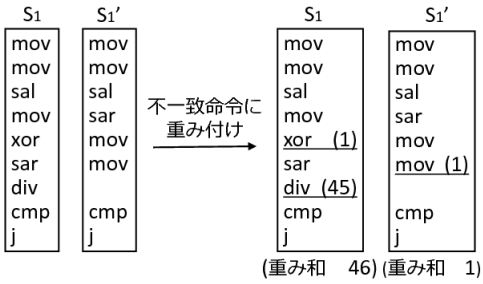


図 7: 不一致命令とその重み和

4. 実行時間計測に基づくコードの比較

テストプログラムに条件文，ループ，関数等が含まれると，アセンブリコードの差がそのまま実行時間の差に反映されるとは限らない．条件によってはあるコード部分が実行されなかったり，ループ展開や関数のインライン展開が行われるためである．また，分岐命令を一切含まないアセンブリコードであっても，コードから推測される性能差が実機で再現されるとは限らない．

そこで，本稿の手法では，前節の手法によってアセンブリコードレベルで性能差があると判定したプログラムに対し，これを最小化したプログラムの実行時間を測定することによって，最終的に性能差の有無を判定する．ランダムに生成したプログラムに対して直接実行時間の測定を行わないのは，テストプログラムが長いと実行に時間がかかるためと，前節でも述べた通り，プログラムが長いと最適化の差異の影響が観測できなくなる可能性が高いためである．

一方，最小化されたプログラムの実行時間はごく僅かになるため，測定誤差が問題となる．そこで，本稿の手法では，テストプログラムの内部で必要回数だけコードの実行をループさせる．テストプログラムの変換例を図 8 に示す．(1) が元の (最小化された) テストプログラムであり，その中にある main は， (2) の実行時間測定用のプログラムの中では main0 と改名されて指定回数繰り返し実行される．この際に，main0 のインライン展開 (とそれに伴う最適化) を防ぐため， 2 行目にインライン展開を強制的に抑止するディレクティブを挿入している．

5. 実験結果

本稿で提案するランダムテストを Orange4 に実装した．本システムは Perl 5.20.2 で実装し，Ubuntu16.04LTS の環境で動作する．

GCC-6.1.0 を比較対象に GCC-7.0.1 の最適化性能テス

ⓒ 2017 Information Processing Society of Japan

(1) 最小化後

```
#include <stdio.h>

int main (void){
  ...
  return 0;
}
```

(2) 実行時間計測用

```
1: #include <stdio.h>
2: int main0()
     __attribute__((noinline));
3: int main (void)
4: {
5:   long i;
6:   for(i=0;i<10000000;i++)
7:   {
8:     int rc = main0();
9:   }
10:  return 0;
11: }
12: int main0(){
13:   ...
14:   return 0;
15: }
```

図 8: 実行時間計測のための変換

トを行った．コンパイラの最適化オプションは -O3 である．1 つのテストプログラム中の演算数は 800 に設定した．また，アセンブリコードに差があると判定する際の命令数，あるいは不一致命令の重み付き和の比の閾値は 80 % に，実行時間を比較する際の比の閾値は 40 % に設定した．

結果を表 1，表 2 に示す．表 1 はテストプログラムが条件文とループを含まない場合，表 2 は含む場合である．表中の「テスト数」はテストプログラムの総数，「NG 数」はそのうちアセンブリコードの比較により差があると判定されたものの数，「目視確認」は最小化されたテストプログラムを目視して最適化性能に差があると判定できたものの数，「実行時間差」は実行時間に差が生じたものの数である．表 1 において，命令数の比に基づく従来法では，アセンブリコードの比較によって差分を 1 件も検出できなかったのに対し，本手法では 90 件検出できた．このうち 81 件が目視で有意差ありと判定され，実行時間を計測した結果 14 件に差があると判定された．この 14 件は全て 81 件に含まれる．目視で誤判定と判断された 9 件は，ショートカットオペレータ (&& 演算や || 演算) をコンパイルした結果生じる条件分岐命令の影響で，不一致部分対の抽出に失敗したことによるものであった．表 2 では，従来法でアセンブリコードの差分が 1 件検出されたが，実行時間に差はなかった．これに対し本手法では，実行時間差が生じるプログラムを 9 件検出することができた．なお，実行時間の測定は，Intel Core i5–6200U 2.30GHz×4, RAM 7.6GiB の環境で行ったが Intel Xeon E3–1276 v3 2.30GHz×4, RAM 15.6GiB の環境でも同じ結果を得ることができた．

本手法により，GCC-7.0.1 を比較対象に GCC-8.0.0 (2017 年 5 月時点での最新開発版) の -O3 オプションのテストを行った結果，5,000 件のテストプログラム中 3 件で実行時間差を検出した．最小化したプログラムとそのアセンブリコードを図 9 に示す．この不具合は，GCC の Bugzilla を通して開発チームに報告した [*1]．

表 3 は，GCC-8.0.0 と LLVM/Clang-4.0 の比較を行った結果である．600 の演算子を含む 500 件のテストプログラムについてテストを行った結果，GCC-8.0.0 の方が高速だったものが 11 件，LLVM/Clang の方が高速だったものが 18 件検出された．

[*1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=81165

表 1: 実験結果 (算術式のみのテストプログラム)

method	テスト数	NG 数	目視確認	実行時間差
本手法	5,000	90	81	14
従来法	5,000	0	0	0

compiler: GCC-7.0.1 -O3, GCC-6.1.0 -O3

表 2: 実験結果 (条件文とループを含むテストプログラム)

method	テスト数	NG 数	目視確認	実行時間差
本手法	5,000	118	31	9
従来法	5,000	1	1	0

compiler: GCC-7.0.1 -O3, GCC-6.1.0 -O3

表 3: GCC-8.0.0 と LLVM/Clang-4.0 の比較

コンパイラ (ターゲット)	テスト数	実行時間差
GCC-8.0.0	500	18
LLVM/Clang-4.0.0	500	11

compiler option: -O3

6. むすび

本稿では，アセンブリコードの不一致部分に着目した比較，および実行時間の計測に基づく C コンパイラの最適化性能テストを提案した．アセンブリコードの比較は，アセンブリコード中の不一致部分対の性能比較と不一致命令を抽出し，重み和を算出に基づいて行った．さらに，アセンブリコードの差分を検出した場合に実行時間の計測を行うことで誤判定を排除した．性能評価の結果から従来の比較法より最適化不足のあるプログラムの見逃しや誤判定を減らすことができ，精度の高い比較法を実現した．

提案した比較法を等価変換に基づくランダムテストシステムに実装した結果，GCC-8.0.0 で 1 件の最適化の不足を検出，開発者に報告した．

今後の課題としては，関数呼び出し，制御文等が含まれるプログラムを用いた最適化のテストが挙げられる．

謝辞 本稿の研究にあたり，御指導，御助言を頂きました関西学院大学石浦研究室の諸氏に感謝いたします．

ⓒ 2017 Information Processing Society of Japan

参考文献

[1] 石浦菜岐佐: “コンパイラのファジング,” 電子情報通信学会 *Fundamentals Review*, vol. 9, no. 3, pp. 188–196 (Jan. 2016).

[2] M. Iwatsuji, A. Hashimoto, and N. Ishiura: “Detecting Missed Arithmetic Optimization in C Compilers by Differential Random Testing (short paper),” in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2016)*, pp. 2–3 (Oct. 2016).

[3] 北浦, 石浦: “C コンパイラの最適化のリグレッションテストのためのアセンブリコード比較法, ” 信学ソ大, A–6–12 (Sept. 2016).

[4] Nullstone Corporation: NULLSTONE for C (online), `http://www.nullstone.com/` (accessed 2017-01-07).

[5] E. Eide and J. Regehr: “Volatiles Are Miscompiled, and What to Do about It,” in *Proc. ACM International Conference on Embedded Software (EMSOFT 2008)*, pp. 255–264 (Oct. 2008).

[6] A. Hashimoto and N. Ishiura: “Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs,” *IPSJ Trans. System LSI Design Methodology*, vol. 9, pp. 21–29 (Feb. 2016).

[7] R. B. Evans and A. Savoia: “Differential Testing: A New Approach to Change Detection,” in *Proc. ACM ESEC-FSE ’07*, pp. 549–552 (Sept. 2007).

[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. System LSI Design Methodology*, vol. 7, pp. 91–100 (Aug. 2014).

[9] K. Nakamura and N. Ishiura: “Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation,” in *Proc. Asia and Pacific Conference on Circuits and Systems (APCCAS 2016)*, pp. 676–679 (Oct. 2016).

```
 1: void func() __attribute__((noinline));
 2:
 3: short x0 = 15;
 4:
 5: int main (void)
 6: {
 7:   long i;
 8:   for( i = 0; i < 100000000; i++){
 9:     func();
10:   }
11:   return 0;
12: }
13: void func(){
14:   volatile int x1 = 1U;
15:   volatile char x2 = 0;
16:   char t0 = 0;
17:   unsigned long t1 = 2LU;
18:   int i = 0;
19:   if(1>>x2){
20:     t0 = -1;
21:     t1 = (1&(short)(x1^8U))-1;
22:   }
23:   while(i>(int)((1U>>t1)+
         (char)(128%(10*(25LU&(29%x0))))))){
24:     i += (int)(12L/(1 != (int)t1));
25:   }
26:   if(t0 != -1)__builtin_abort();
27:   if(t0 != 0L)__builtin_abort();
28: }
```

(1) C プログラム

gcc-8.0.0.s	gcc-7.0.1.s
`main:`	`main:`
`  subq $24, %rsp`	`  subq $24, %rsp`
	`  movl $1, %eax`
`  movl $1, %edi`	`  movl $1, 12(%rsp)`
`  movl $1, 12(%rsp)`	`  movb $0, 11(%rsp)`
`  movb $0, 11(%rsp)`	
`  movl %edi, %eax`	
`  movzbl 11(%rsp), %ecx`	`  movzbl 11(%rsp), %ecx`
`  sarl %cl, %eax`	`  sarl %cl, %eax`
`  testl %eax, %eax`	`  testl %eax, %eax`
`  je .L7`	`  jne .L12`
`  movl 12(%rsp), %ecx`	
`  movl $-1, %r9d`	
`  andl $1, %ecx`	
`  subl $1, %ecx`	
`  movslq %ecx, %r8`	
`  shrl %cl, %edi`	
`.L2`	`.L2`
`  movswl x0(%rip), %esi`	`  call abort`
`  movl $29, %eax`	`.L12`
`  cltd`	`  movl 12(%rsp), %eax`
`  divl %esi`	`  andl $1, %eax`
`  movl $128, %eax`	`  subl $1, %eax`
`  andl $25, %edx`	`  testl %eax, %eax`
`  leaq (%rdx,%rdx,4), %rsi`	`  jne .L2`
`  xorl %edx, %edx`	
`  addq %rsi, %rsi`	
`  divq %rsi`	
`  movsbl %dl, %edx`	
`  addl %edi, %edx`	
`  jns .L3`	
`  cmpl $1, %ecx`	
`  je .L10`	
`.L3`	
`  cmpb $-1, %r9b`	
`  jne .L6`	
`  testq %r8, %r8`	
`  jne .L6`	
`  addq $24, %rsp`	`  addq $24, %rsp`
`  ret`	`  ret`
`.L10`	
`  ud2`	
`.L7`	
`  movl $2, %ecx`	
`  xorl %edi, %edi`	
`  movl $2, %r8d`	
`  xorl %r9d, %r9d`	
`  jmp .L2`	
`.L6`	
`  call abort`	
⋮	⋮

(2) アセンブリコード

図 9: 最適化の不足として報告したプログラム

ⓒ 2017 Information Processing Society of Japan