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
VLD2020-66,CPSY2020-49,RECONF2020-85(2021-01)

ランダムプログラム生成による C コンパイラの VRP 最適化の性能テスト

村上　大喜†　　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 VRP 最適化が意図通りに行われているかどうかを検査するランダムテスト手法を提案する. VRP 最適化はコンパイル時に値が決定できる部分式を定数に置き換える最適化の一種であるが, 変換の条件が複雑であり不具合が生じやすい. 本手法は, VRP 最適化が適用できる算術式を含むプログラムと, そのプログラムに VRP 最適化を適用して得られるプログラムを自動生成し, それらをコンパイルして得られるアセンブリコードを比較することによって最適化の不足を検出する. 算術式の生成は VRP 最適化の解析とは逆方向に値の区間 (value range) を伝播させることにより行うが, この際に, 演算子と値の範囲を適切に選択することによって, 必ず VRP 最適化が適用できる式を生成する. 提案手法に基づくテストシステムを C コンパイラのランダムテストシステム Orange4 に追加実装して実験を行った結果, GCC の現時点での最新版 10.2.0 を含む複数のバージョンにおいて VRP 最適化の機会の見逃し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, 最適化

Performance Testing of VRP Optimization of C Compilers by Random Program Generation

Daiki MURAKAMI† and Nagisa ISHIURA†

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

Abstract　This paper proposes an automated method to test if C compilers properly perform VRP optimization. The VRP optimization is a sophisticated variant of constant folding that replaces subexpressions into constants at compile time, whose complex precondition may induce mis-optimization or under-optimization. The proposed method attempts to detect under-optimization by generating a pair of C programs, where VRP optimization is applicable to one program and VRP optimization is applied to the other program at source code level, and comparing the compiled assembly codes. The arithmetic expressions with VRP optimization opportunities are generated by propagating value ranges in the opposite direction to that in range analysis in the optimization and by appropriately choosing operators and operands. A test system has been implemented on top of a random test system Orange4, which detected under-optimization in the latest version 10.2.0 of GCC.

Key words　compiler, random testing, optimization

1. はじめに

ソフトウェア開発の基盤ツールであるコンパイラには, 非常に高い信頼性が求められるため, コンパイラが誤りのないコードを生成しているかに関して徹底的なテストが行われる. さらに近年のコンパイラには処理速度, メモリ使用量, 消費電力に関して優れたコードを生成することも期待されるため, 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 [1].

コンパイラの最適化性能のテスト手法としては, 専用のテストスイート [2] を用いる手法や, ベンチマーク実行のトレース比較 [3] を行う手法が存在するが, テストケースが有限である以上どうしても不具合の見逃しが生じてしまう. これを補う手法の一つとしてランダムテスト [1] が用いられる.

コンパイラの最適化性能をテストするランダムテスト手法には差分法 [4] [5] と等価プログラム法 [6] [7] が存在する. 定数畳み込みや演算強度の軽減等, 算術式のコンパイル時簡単化を行う tree-optimization は, 誤りのないコードを生成できるかどうかのテストでも不具合が生じ易い部分だが [8], 最適化の性能テストでも多くの不具合が検出されている [9]. VRP 最適化は複雑な tree-optimization の一種であり, 式中の各演算で取り得

る値の範囲を伝播させていくことにより, 値の定まらない変数が含まれる式に対しても簡単化を行うものである. しかし, 既存のプログラム生成手法では VRP 最適化の性能を直接テストするプログラムを生成できないため, VRP 最適化の性能テストは十分に行えない.

本稿では等価プログラム法に基づいて VRP 最適化を対象とした最適化性能のランダムテストを行うためのプログラムの生成手法を提案する. プログラムの生成時に VRP 最適化の解析とは逆方向に値の区間を伝播することによって, 必ず VRP 最適化が適用可能なテストプログラムを生成する.

本手法に基づくテストシステムを C コンパイラのランダムテストシステム Orange4 [10] に追加実装して実験を行った結果, 最新版の GCC-10.2.0 を含む複数の GCC のバージョンにおいて従来の手法では困難な VRP 最適化の機会の見逃しを検出することができた.

2. 等価プログラム生成によるコンパイラの最適化性能のランダムテスト

2.1 定数畳み込みと VRP 最適化

定数畳み込みとは, 定数 (および値が確定している変数) を含む式の値をコンパイル時に計算することによって, 式を簡単化する最適化手法である. 定数畳み込みの例を図 1 に示す. (a) のプログラムに定数畳み込みを行った結果, 5 行目は (b) のように簡単化できる. 定数畳み込みは単純な最適化だが, 他の最適化との組み合わせ (特に後続の条件分岐やループの簡単化との組み合わせ) で効果を発揮したり, 他の最適化の適用を可能にする.

しかし, 式中に volatile 修飾された変数 (以下 volatile 変数) やグローバル変数が含まれる場合には, 定数畳み込みの適用が制限される. volatile 修飾子とは, その変数の値がプログラムの外部から書き換えられる可能性があることを表すもので, 複数プロセス間の共有変数や外部機器のレジスタにマッピングされた変数等に指定する. volatile 変数の値は実行時のどの時点でどの値に変わるかわからないため, その変数に関する最適化が抑止される. 図 2 (a) は 図 1 (a) の x1 に volatile 修飾子を付与したものであるが, x1 を含む部分式には定数畳み込みを適用することができなくなる.

VRP 最適化は, 定数値ではなく値の範囲を伝播させることにより, 定数畳み込みよりも強力な式の簡単化を実現するものである. 図 2 の (a) と同じプログラムに VRP 最適化を適用したものを図 3 に示す. MIN と MAX はそれぞれその変数の型の値の最小値と最大値であり, 図中の $[\mathrm{MIN}, \mathrm{MAX}]$ は, `x1` の値は定まっていないがこの範囲に入っていることを表している. `x2` の値の範囲は $[6,6]$ なので, `x % 2` の範囲は $[-5,5]$ となる. 同様に部分式の区間を求めていくことにより, 5 行目の式は `x1` の値によらず 0 に簡単化できる(注1).

2.2 コンパイラ最適化性能のランダムテスト

コンパイラの最適化性能のランダムテストの手法としては, 差分法 [4] [5] と等価プログラム法が提案されている. 差分法は, 図 4 (a) のようにランダムに生成したプログラム org.c を異なるコンパイラ compilerA, compilerB (あるいは同じコンパイラの異なるバージョン) でコンパイルし, 生成したアセンブリ

(注1) : コンマ演算子で `x1` を付加しているのは, `x1` の値を使用するか否かに関わらず `x1` の読み出しは行わなければならないためである

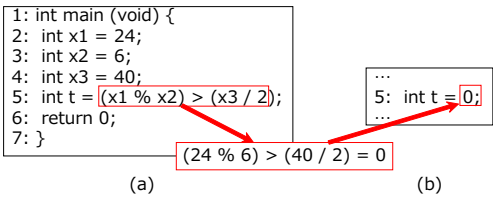


図 1 定数畳み込み

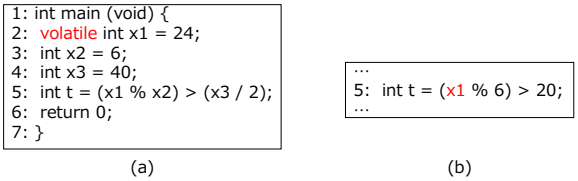


図 2 volatile 変数を含む式の定数畳み込み

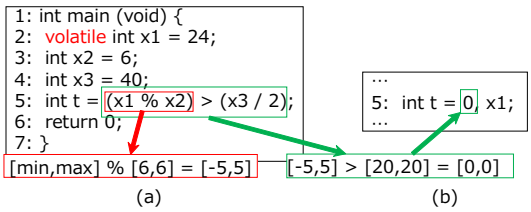
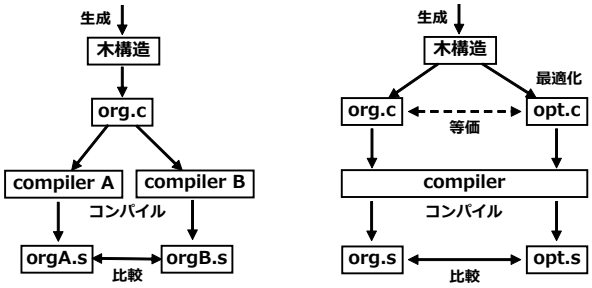


図 3 VRP 最適化



(a) 差分法 (b) 等価プログラム法

図 4 コンパイラの最適化性能のランダムテスト手法

コード orgA.s, orgB.s を比較することによってコンパイラの最適化機会の見逃しを検出するものである. 等価プログラム法は, 図 4 (b) のようにランダムに生成したプログラム org.c と, それに期待される最適化をソースコードレベルで行ったプログラム opt.c から生成されるアセンブリコード org.s, opt.s を比較することによって意図通りの最適化が行われているかをテストする手法である. 差分法はリグレッションのテストを容易に行うことができ, 等価プログラム法は差分法では検出できない最適化の見逃しを検出できる.

コンパイル結果の比較法には, 2 つのコードの差分から実行サイクル数の大きな命令を抽出する方法 [9], ループに着目したスピルコード量の差を求める方法 [4], 実際の実行時間比較を併用する方法 [5] 等がある.

文献 [6] [7] では等価プログラム法で多数の最適化機会の見逃しを検出しており, その中で VRP 最適化の不足も検出している. しかし, opt.c の生成時に VRP 最適化は適用していないので, VRP 最適化の不足が検出できるのは org.c と opt.c のいずれかのコンパイル時に VRP 最適化が適用されなかった偶然のケースに限られている.

2.3 ランダムテストシステムを用いたプログラム生成

C コンパイラのテストのためにランダムなプログラムを生成する手法においては, 未定義動作を含むプログラムの生成をいかに回避するかが重要な課題になる. Csmith [8] では, 式や文の構文に制限を設けることによって未定義動作を回避しつつ, 関数呼び出し, 配列や構造体, ポインタ等を含む C 言語の幅広い構文要素を含むフログラムを生成している. Orange4 [10] ではプログラム中の全ての変数と式の値を保持するデータ構造を用いてプログラムを生成することにより, 未定義動作を含まない複雑な式の生成を可能にしている.

Orange4 の式生成の流れを図 5 に示す. まず最初に式の値 (12) を決定する. 次に, 演算子をランダムに選択し (この場合は +), 親の値が生成されるように子の値 (2 と 10) を選ぶ. この操作を再帰的に繰り返すことによって未定義動作を引き起さない式を生成し, かつ全ての変数と部分式の値をプログラム生成時に把握することを可能にしている.

Orange4 が最適化性能のテスト [5] のために生成するプログラム対の例を図 6 に示す. org.c の 4–18, 22–27 行目が変数宣言, 29–32 行目が算術式や制御文を含む文, 34–37 行目が計算結果を照合する文である. opt.c は 29–32 行目の式に定数畳み込みが行われている以外は org.c と同じである. しかし, opt.c の式に対して VRP 最適化は施していないため, VRP 最適化の不足は偶然にしか検出できない.

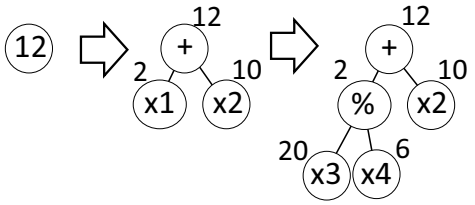


図 5　Orange4 の式生成

org.c

```
01:#define OK()
02:#define NG(test,fmt,val)  __builtin_abort()
03:
04:volatile unsigned long x1 = 268048037LU;
05:const volatile unsigned long x4 = 2LU;
06:volatile unsigned long x5 = 1072432214LU;
07:volatile signed int x6 = 1;
08:volatile signed long t0 = -3725799357L;
09:volatile unsigned long x10 = 13066417559506371878LU;
10:volatile unsigned long long x11 = 18446744073709551615LLU;
11:unsigned short t1 = 230U;
12:unsigned int x14 = 1U;
13:signed long x16 = 19259L;
14:signed int x17 = 1;
15:volatile signed short t2 = -1;
16:volatile unsigned char x8 = 212U;
17:static signed int x2 = 1;
18:unsigned char t3 = 0U;
19:
20:int main (void)
21:{
22:    volatile unsigned long x7 = 13066417559506391136LU;
23:    volatile unsigned long long x12 = 18446744073709551615LLU;
24:    unsigned int x13 = 1615882373U;
25:    volatile signed short x15 = -19258;
26:    volatile unsigned char x9 = 1U;
27:    static const signed long long x3 = 1LL;
28:
29:    t0 = ((signed long)(((signed int)(x1<<x4))-
           ((signed int)(x5/x6))));
30:    t1 = ((unsigned short)(((signed short)(x7%x10))*(x11>=x12)));
31:    t2 = ((signed short)(((signed long long)(x13%x14))||
           ((unsigned long long)((x15*((signed short)x14))+
           ((signed short)((unsigned short)(x16-x17)))))));
32:    t3 = ((x8/x9)>>((unsigned long long)((unsigned int)
           (((((signed short)t1)>=((unsigned long)t1)))*
           ((unsigned long)(x2*x3))))));
33:
34:    if (t0 == -240066L) { OK(); } else { NG("t0", "%ld", t0); }
35:    if (t1 == 19258U) { OK(); } else { NG("t1", "%hhu", t1); }
36:    if (t2 == 0) { OK(); } else { NG("t2", "%hhd", t2); }
37:    if (t3 == 106U) { OK(); } else { NG("t3", "%u", t3); }
38:
39:    return 0;
40:}
```

opt.c

```
            ...

29:    t0 = (signed long)(((signed int)(x1<<x4))-((signed int)(x5/x6)));
30:    t1 = ((unsigned short)(((signed short)(x7%x10))*(x11>=x12)));
31:    t2 = ((signed short)((signed long long)0LL||((unsigned long long)
           ((x15*(unsigned int)1U)+(unsigned short)19258U))));
32:    t3 = ((x8/x9)>>(unsigned int)1U);

            ...
```

図 6　Orange4 が生成する最適化性能のテストのプログラム対

3. VRP 最適化をテストするプログラムの生成

3.1 概　　要

本稿では, 文献 [6] [7] の手法を拡張することにより, 等価プログラム法に基づいて VRP 最適化の性能テストを行う手法を提案する.

本手法でも, Orange4 の手法と同様, 目標の値に評価される式をトップダウンに生成するが, この際に, VRP 最適化が適用できるパスに沿って VRP 最適化の解析とは逆方向に値の区間を伝播させていく. 演算子の選択を適切に制御することにより, 必ず VRP 最適化が適用できる式を生成する.

3.2 VRP 最適化が適用可能なプログラムの生成

本稿では, 1 つの式中に volatile 変数は唯一存在する場合を考える. また式の値は 0 になるものだけを考える.

本手法では, 式生成を行う際の節点に n 節点と v 節点の 2 種類を設ける. v 節点はそれを根とする部分木から生成される式に volatile 変数が含まれる節点であり, n 節点は含まれない節点である. n 節点には値を 1 つ, v 節点には値の区間 (value range) を定義する. この区間はその節点の取り得る値が必ずその区間内に収まっていることを表す.

式の生成は, 図 7 に示すように, 区間が [0,0] の v 節点から開始して, 節点を木に変換する操作を再帰的に適用することにより行う. 区間が [MIN, MAX] の v 節点が生成されれば, その節点に対応する変数を volatile 変数とする.

n 節点の木への変換は Orange4 の場合 (図 5) と同様である. v 節点の変換は, 次の (1)〜(4) により行う.

（1） 演算子と子節点の型を決定する
（2） 子の節点の一方を v, もう一方を n 節点に決定する
（3） n 節点の値を決定する
（4） v 節点の区間の決定する

このうち, (4) の区間は, 子の v 節点がその区間内のどの値を取っても親節点の値が親節点の区間の範囲内に収まり, かつ区間の幅ができるだけ大きくなるように決定する. 例えば, 図 8 (a) のように, 演算に加算を選択した場合には, 子の v 節点の区間は親の区間から子の n 節点に設定した値を減じたものになる. 図 8 (b) のように, 剰余算を選択した場合には, 演算の結果を [−2,10] の範囲に収められるできるだけ広い区間を求める. 上限はいくら大きくても結果は 4 を超えないが, 下限は −2 以上とする必要がある. この結果, 親節点の取りうる値の範囲は [−2,4] となり, [−2,10] の範囲に収まる. 親節点の取りうる値を [−2,10] に収める子節点の区間は他にも [−7,−5], [−12,−10], … があるが, 本手法ではその中からできるだけ区間が大きくなるものを選ぶ.

v 節点の演算には, 加算や剰余算の他に, 減算, 除算, 比較演算, 論理演算, シフト演算, bit 演算が選択できる.

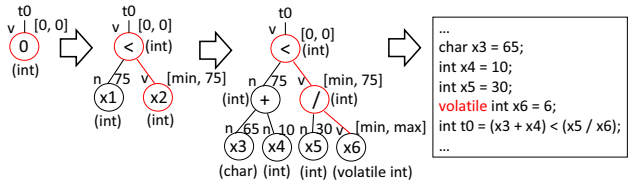
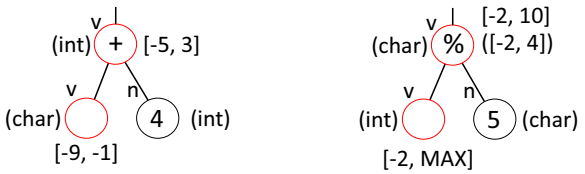


図 7　本手法の式生成の例



(a) 加算　　(b) 剰余算

図 8　子の v 節点の区間の決定

3.3　演算子の選択

v 節点の演算をランダムに選択すると区間が [MIN, MAX] の v 節点は非常に低い確率でしか生成できないため, 本手法では, 木があらかじめ定めた深さになった時点で, 子の v 節点に区間 [MIN, MAX] を生成できるように特定の演算と子の n 節点の値を選択する. ただし, v 節点の区間によっては必ずしも子の区間を [MIN, MAX] にできるとは限らないので, その場合には区間を調整する演算を挿入する.

本手法では [MIN, MAX] の v 節点を生成するために, 除算, 剰余算, 比較演算 (==, !=, <, >, <=, >=), および論理演算 (&&, ||) を用いる. 詳細は次の通りである, ただし, 親節点の区間を $[A, B]$, 子の n 節点の値を c と表記する.

(1) 除算

除数 (右の子) を n 節点にした場合 (図 9 (a)) は, c の絶対値を $\max(|\mathrm{MIN}|, |\mathrm{MAX}|)/\min(|A|, |B|)$ 以上にすれば子の v 節点の区間を [MIN, MAX] にできる. ただし, このためには $[A, B]$ が 0 を含んでいることが必要になる.

被除数 (左の子) を n 節点にした場合 (図 9 (b)) は, c の絶対値を $\min(|A|, |B|)$ 以下にすれば子の v 節点の区間を [MIN, MAX] にできる. ただし, このためには $[A, B]$ が 0 を含んでいることが必要になる.

(2) 剰余算

除数 を n 節点にした場合 (図 9 (c)) は, c の絶対値を $\min(|A|, |B|) + 1$ 以下にすれば子の v 節点の区間を [MIN, MAX] にできる. ただし, このためには $[A, B]$ が 0 を含んでいることが必要になる.

被除数 を n 節点にした場合 (図 9 (d)) は, c の範囲を $[A, B]$ にすれば子の v 節点の区間を [MIN, MAX] にできる. ただし, このためには $[A, B]$ が 0 を含んでいることが必要になる.

(3) 比較演算

小なり (<) 演算の場合 (図 9 (e)) は, c を MAX 以上にすれば子の v 節点の区間を [MIN, MAX] にできる. ただし, このためには $[A, B]$ が 0 を含んでいることが必要になる. また c を MIN 未満にした場合も [MIN, MAX] にできる, この場合には $[A, B]$ が 1 を含んでいることが必要になる.

他の比較演算についても同様の方法で区間を [MIN, MAX] にできる.

(4) 論理演算

論理和 (||) 演算の場合 (図 9 (f)) は, c を 0 以外にすれば子の v 節点の区間を [MIN, MAX] にできる. ただし, このためには $[A, B]$ が 1 を含んでいることが必要になる.

以上をまとめると, 親節点の区間が下記条件を満たせば, 必ず子の v 節点の区間を [MIN, MAX] にすることができる.

- 除算, 剰余算, &&演算: 親節点の区間が 0 を含む
- ==演算, !=演算: 親節点の区間が 0 と 1 を含む
- <演算, <=演算, >演算, >=演算: 親節点の区間が 0 または 1 を含む
- ||演算: 親節点の区間が 1 を含む

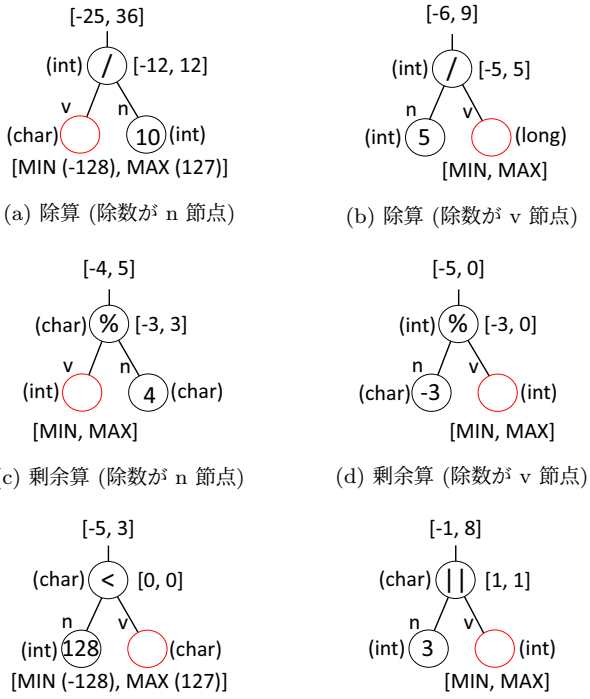
そこで本手法では, 木の深さの上限の手前で区間が 0 と 1 を含むように調整節点を挿入し, その次の段階で [MIN, MAX] の v 節点が生成できるようにする. 具体的な調整節点の挿入法は次の通りである.

- v 節点の区間 $[A, B]$ が $A \neq B$ を満たす場合:

$[A, B]$ が 0 と 1 の両方を含んでいれば調整節点は挿入しない. そうでなければ加減算節点を追加して, 区間が 0 と 1 の両方を含む v 節点を生成する.

- 節点の区間が $[A, A]$ である場合:

$A = 0$ または $A = 1$ であれば調整節点は挿入しない. そうでなければ, 加減算節点を追加して, 区間が [0,0] または [1,1] の v 節点を生成する.



(a) 除算 (除数が n 節点)　　(b) 除算 (除数が v 節点)

(c) 剰余算 (除数が n 節点)　　(d) 剰余算 (除数が v 節点)

(e) 比較演算　　(f) 論理演算

図 9　子 v 節点の区間を [MIN,MAX] にする演算

3.4　volatile 変数を畳み込む等価なプログラムの生成

本手法で生成した式の値は必ず 0 になるので, 図 10 の例のように式を 0 に置き換えることにより比較に用いる等価プログラムを得ることができる. ただし, volatile 変数は, プログラム中でその値を用いていなくても必ず読み出さなければならないので, コンマ演算子を用いて変数へのアクセスを明記する.

本手法で生成されるテストプログラム対の例を図 11 に示す. プログラムの構成は図 6 と同様で, org.c の 07–34 行目で変数

の初期化を行い, 36–38 行目で算術演算を行い, 40–42 行目で期待値照合を行っている. opt.c の 36–38 行目は算術式に VRP 最適化を行った結果であり, opt.c のそれ以外の行は org.c と同じである.

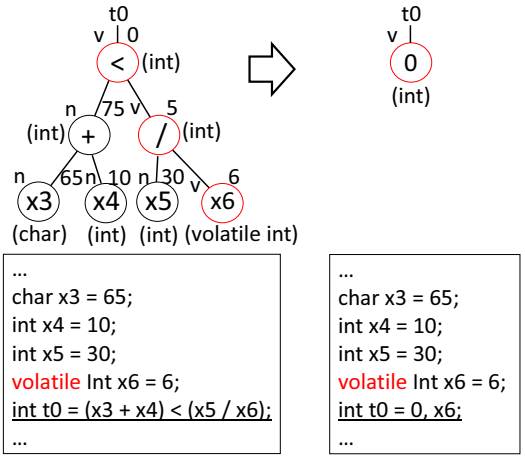


図 10　等価なプログラムの生成

org.c

```
01:#define OK()
02:#define NG(test,fmt,val)  __builtin_abort()
03:
04:int main (void)
05:{
06:
07:    static signed long x1 = 0L;
08:    unsigned int x3 = 0U;
09:    const unsigned int x5 = 3U;
10:    const signed short x6 = -15;
11:    const signed long long x8 = 1114769886683138829LL;
12:    const signed long x16 = -16559L;
13:    const signed long x17 = -1L;
14:    signed long x18 = 10582162649112L;
15:    const signed long x19 = 31527218475511L;
16:    const signed long x20 = 14L;
17:    volatile signed long x21 = 29L;
18:    signed long x22 = 15L;
19:    signed long t0 = 1403538L;
20:    static const signed int x24 = 302210371;
21:    static signed long long x25 = 302210371LL;
22:    static const signed long x27 = -8216910278661L;
23:    volatile signed long x28 = -3150890944190355132L;
24:    static signed long t1 = -13542L;
25:    const signed long x29 = -1114738359464663318L;
26:    signed long x30 = -1114717414408836919L;
27:    const signed long x31 = -1114769886683138829L;
28:    signed long x32 = 20945055826398L;
29:    const signed long x33 = 1L;
30:    signed long long t2 = 42958833LL;
31:    static unsigned int t3 = 12568U;
32:    static const signed long x35 = -6185L;
33:    const signed long x36 = -22744L;
34:    signed char t4 = 0;
35:
36:    t0 = ((signed long)(((signed long)(((((unsigned char)x1)+
           ((signed short)x16)))/((signed long)(((x21%x22))+
           ((signed long)((signed short)(((signed int)x6)-
           ((signed long long)x1)))))))))>=(x18^x19)));
37:    t1 = ((signed long)((((signed long)(((((signed long long)x1)+
           x18))+((signed long)(x27>=x28))))+((signed long)(x27-
           ((signed long long)x5))))<=((signed long)(x24^x25))));
38:    t2 = ((signed long long)((((signed long)(((signed long)(x21<=
           ((signed long)x24)))-(x32+x33)))+((signed long)(x31/
           ((signed char)x17)))))>=((signed long)((signed long long)(x8>>
           ((signed char)x3))))));
39:
40:    if (t0 == 0L) { OK(); } else { NG("t0", "%ld", t0); }
41:    if (t1 == 0L) { OK(); } else { NG("t1", "%ld", t1); }
42:    if (t2 == 0LL) { OK(); } else { NG("t2", "%lld", t2); }
43:
44:    return 0;
45:}
```

opt.c

```
           ...

36:    t0 = 0L, x21;
37:    t1 = 0L, x28;
38:    t2 = 0LL, x21;

           ...
```

図 11　本手法において生成されるテストプログラムの対

4. 実装と実験結果

4.1 実　　装

本稿の提案手法に基づくコンパイラの最適化性能のランダムテストシステムを Orange4 [10] のプログラム生成系を利用して実装した. 実装言語は Perl 5 であり, システムは Ubuntu Linux 等の UNIX 環境で動作する. 今回想定したアーキテクチャは x86_64 であり, アセンブリコードの構文は GNU アセンブラを前提としている. アセンブリコードの比較は文献 [5] [9] で提案されている手法で行った.

本実装では, 式の生成においてシフト演算とビット単位の論理演算は用いていない. シフト演算は Orange4 の式生成アルゴリズムとの整合性の問題で実装が難しかったためである. ビット単位の論理演算は, 式生成時の区間の伝播が難しかったためである.

4.2 実　　験

最新版を含む GCC の 6 つのバージョンと LLVM (Clang) の 2 つのバージョンを対象にテストを行った. 最適化オプションは -O3 とし, 1 つのプログラム中の演算子数は 100, テスト数は 2,000 とした. 実行環境はプロセッサが Intel Core i5–6200U CPU@2.30GHz, OS は Ubuntu 18.04.5 LTS である.

GCC のテスト結果を表 1 に示す. 表中の compiler はテスト対象のコンパイラ, #test はテストプログラム数, time [h] は実行時間 (単位は時間) を表している. #diff はアセンブリコードに有意な差があるプログラム数, つまりコンパイラの最適化不足を検出したプログラムの数を表している. GCC-7.5.0 で最適化不足検出した 3 つのプログラムはすべてのバージョンで最適化不足を検出した. GCC-8.0.1, GCC-9.3.0, GCC-10.2.0 で最適化不足を検出した 6 つのプログラムはすべて同じものであった.

最新版の GCC-10.2.0 で差分を検出したプログラムを最小化したものを図 12 に示す. アセンブリコード (org.s opt.s) の比較から, org.c に対して VRP 最適化が行えていないことがわかる. 旧バージョン GCC-7.5.0 では org.c と opt.c の両方に対して opt.s と同じコードが生成されていたため, これはリグレッション (バージョンアップに伴う不具合の導入) と考えられる. この不具合は GCC Bugzilla(注2) を通して開発チームに報告し, 開発版の GCC にて対応が行われた.

同じく GCC-10.2.0 で検出したプログラムを最小化したものを図 13 に示す. アセンブリコードの比較から, この org.c に対しても意図通りの最適化を行えていないことがわかる. この不具合についても GCC Bugzilla(注3) を通して開発チームに報告している. このプログラムは実験したすべてのバージョンで検出されたため, 差分法では検出が困難な最適化の見逃しである.

LLVM のテスト結果を表 2 に示す. 2,000 のテストプログラムは GCC の実験と同じテストプログラムである. この実験では最適化機会の見逃しは検出されなかった.

(注2)：https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97964
(注3)：https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97967

表 1　実験結果 (GCC)

compiler	#test	time [h]	#diff
GCC-5.5.0	2000	0.20	12
GCC-6.5.0	2000	0.20	11
GCC-7.5.0	2000	0.20	3
GCC-8.0.1	2000	0.21	6
GCC-9.3.0	2000	0.20	6
GCC-10.2.0	2000	0.20	6

(CPU Intel Core i5–6200U CPU@2.30GHz, Ubuntu 18.04.5 LTS)

org.c (GCC-10.2.0)

```
 1:  int main()
 2:  {
 3:    volatile int a = -1;
 4:    long b = -2;
 5:    int c = a>0;
 6:    int d = b*c;
 7:    int e = 1-d;
 8:    int t = (-1/(int)e)==1;
 9:    return 0;
10:  }
```

opt.c (GCC-10.2.0)

```
 1:  int main()
 2:  {
 3:    volatile int a = -1;
 4:
 5:    a;
 6:
 7:
 8:    int t = 0;
 9:    return 0;
10:  }
```

org.s (GCC-10.2.0)

```
 1:  movl $1,12(%ecx)
 2:  movl $-1,12(%rsp)
 3:  movl 12(%rsp),%eax
 4:  testl %eax,%eax
 5:  setle %al
 6:  movzbl %al,%eax
 7:  leal -2(%rax,%rax),%eax
 8:  subl %eax,%ecx
 9:  movl $-1
10:  cltd
11:  idiv %ecx
12:  cmpl $1,%eax
13:  jg .L3
14:  xorl %eax,%eax
```

opt.s (GCC-10.2.0)

```
 1:  movl $1,-4(%rsp)
 2:  movl -4(%rsp),%eax
 3:
 4:
 5:
 6:
 7:
 8:
 9:
10:
11:
12:
13:
14:  xorl %eax,%eax
```

図 12　GCC-10.2.0 で検出した最適化の不具合

org.c (GCC-10.2.0)

```
 1:  int main()
 2:  {
 3:    volatile int a = 1;
 4:    int b = a%2;
 5:    int t = 200<(short)(-50*b);
 6:    return 0;
 7:  }
```

opt.c (GCC-10.2.0)

```
 1:  int main()
 2:  {
 3:    volatile int a = 1;
 4:    a;
 5:    int t = 0;
 6:    return 0;
 7:  }
```

org.s (GCC-10.2.0)

```
 1:  movl $1,12(%rsp)
 2:  movl 12(%rsp),%eax
 3:  movl %eax,%edx
 4:  shrl $31,%edx
 5:  addl %edx,%eax
 6:  andl $1,%eax
 7:  subl %edx,%eax
 8:  imul $-50,%eax,%eax
 9:  cmpw $200,$ax
10:  cmpl $1,%eax
11:  jg .L3
12:  xorl %eax,%eax
```

opt.s (GCC-10.2.0)

```
 1:  movl $1,-4(%rsp)
 2:  movl -4(%rsp),%eax
 3:
 4:
 5:
 6:
 7:
 8:
 9:
10:
11:
12:  xorl %eax,%eax
```

図 13　GCC-10.2.0 で検出した最適化機会の見逃し

表 2　実験結果 (LLVM)

compiler	#test	time [h]	#diff
LLVM-10.0.0	2000	0.14	0
LLVM-11.0.1	2000	0.14	0

(CPU Intel Core i5–6200U CPU@2.30GHz, Ubuntu 18.04.5 LTS)

5. む す び

本稿では C コンパイラの VRP 最適化の性能テストを目的としたプログラムのランダム生成手法を提案した. 実験の結果, 最新版の GCC-10.2.0 を含むコンパイラで従来法では検出が困難な VRP 最適化不足を検出できた.

本稿では, 式の値を 0 に, 式中の volatile 変数の数を 1 に限定しているが, この制限は緩和可能と考えられる. また, シフト演算やビット単位の論理演算に関係する VRP 最適化不足も実際に検出されているので, これらの演算に対応した拡張を行なっていくことも今後の課題として挙げられる.

謝　　辞

本研究に関してご協力, ご討議頂いた関西学院大学理工学部石浦研究室の諸氏に感謝致します.

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] Nullstone Corporation: NULLSTONE for C (online), `http://www.nullstone.com/` (accessed 2019-01-22).

[3] T. Moseley, D. Grunwald, and R. Peri: “OptiScope: Performance Accountability for Optimizing Compilers,” in *Proc. IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2009)*, pp. 254–264 (Mar. 2009).

[4] G. Barany: “Finding Missed Compiler Optimizations by Differential Testing,” in *Proc. International Conference on Compiler Construction (CC 2018)*, pp. 92–91 (Feb. 2018).

[5] K. Kitaura and N. Ishiura: “Random Testing of Compilers’ Performance Based on Mixed Static and Dynamic Code Comparison,” in *Proc. ACM International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST 2018)*, pp. 38–44 (Nov. 2018).

[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] 樋口, 石浦：“コンパイラの最適化性能テストにおける畳み込み可能な部分式のトップダウン生成,” 信学ソ大, A–6–6 (Sept. 2017).

[8] X. Yang, Y. Chen, E. Eide and J. Regehr: “Finding and Understanding Bugs in C Compilers,” in *Proc. ACM Conference on Programming Language Design and Implementation (PLDI ’11)*, pp. 283–294 (Oct. 2011).

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

[10] 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).