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
VLD2017-88,CPSY2017-132,RECONF2017-76(2018-01)

LLVM バックエンドの最適化性能テストのミュータント生成

田中　健司† 　石浦菜岐佐† 　西村　啓成†† 　福井　昭也††

† 関西学院大学 理工学部　〒669–1337 兵庫県三田市学園 2–1
†† ルネサス エレクトロニクス株式会社　〒135–0061 東京都江東区豊洲三丁目 2 番 24 号

あらまし　本稿では, 既存のテストプログラムから自動生成した機能等価なミュータントにより, コンパイラ基盤 LLVM のバックエンドの最適化性能をテストする手法を提案する. LLVM のバックエンドはコード生成のための変換に加えて, ターゲット依存の最適化や各種ピープホール最適化を行うため, 生成されているコードの正誤だけでなく, 意図通りの最適化が行われているかのテスト (性能テスト) が必要になる. 最適化の性能テストは, 主としてコンパイラ開発者が手動で作成したテストプログラムによって行われるため, テストプログラムのバリエーションが限られてしまう. 本稿の手法では, 開発者が特定の最適化機能を対象に作成したテストプログラムからミュータントを自動生成することによって, バックエンドの性能テストを強化する. 本手法のミューテーション操作は元プログラムの実行結果を変化させないものに限定するため, 意図通りの最適化ができているかどうかの判定はアセンブリコードの機械的な比較により行える. 提案手法に基づくツールを Perl 5 を用いて実装しテストを行った結果, LLVM 6.0.0 の x86_64 用バックエンドにおいて, バックエンドの性能向上の参考になると考えられるテストケースを 2 件検出した.
キーワード　LLVM IR, バックエンド, ミューテーション, 最適化

Mutant Generation of Performance Tests for LLVM Back-Ends

Kenji TANAKA†, Nagisa ISHIURA†, Masanari NISHIMURA††, and Akiya FUKUI††

† Kwansei Gakuin University,　Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan
†† Renesas Electronics Corporation,　Toyosu 3–2–24, Koto-ku, Tokyo, 135–0061 Japan

Abstract　This article presents a method of testing optimization capability of LLVM back-ends by generating functionally equivalent test mutants from existing test programs. Since the LLVM back-ends perform various target dependent and peephole optimization as well as transformations for code generation, it is necessary to test if optimization is properly done as designed to enhance performance, not to mention if generated codes are correct. Test programs for the performance test are usually developed manually by compiler designers, which do not always provide enough variation to cover corner cases. The method in this article attempts to augment test cases by generating mutants from existing test programs. The mutation in our method is designed not to change the functionality of the original test programs, so that insufficient optimization is detected by mechanical comparison of assembly codes. In a preliminary experiment on the LLVM 6.0.0 back-end for x86_64, a tool based on the proposed method has found two interesting cases which might contribute toward performance improvement of the back-end.
Key words　LLVM IR, Back-Ends, Mutation, Optimization

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, 高い信頼性とともに, 高い最適化性能が求められる. このため, 正しいコードが生成されているかどうかは無論のこと, 意図通りの最適化が行われているかについても徹底的なテストが必要になる.

オープンソースのコンパイラとしては GCC [1] が広く利用されているが, 近年では, LLVM/Clang [2] が GCC に代わるコンパイラとして利用される機会が増加している. LLVM はライセンスが BSD ライセンスに近く, GCC に比べ制約が緩いことから, Swift [3] のコンパイラや, 高位合成ツール LegUp [4] 等様々なプロジェクトで利用されるようになっている. LLVM/Clang は言語依存のフロントエンド, 言語/ターゲットに非依存のミドルエンド, ターゲット依存のバックエンドの 3 つが明確に独立した構造を持ち, その間を LLVM IR と呼ばれる中間表現を用いてやり取りする. このため, 新しい命令セットアーキテクチャに対応したコンパイラを作成するにはバックエンドのみを開発すれば良いが, その際には, バックエンドを対象としたテストが非常に重要になる.

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2018 by IEICE

バックエンドでは, 受け取った LLVM IR にターゲット依存のコード変換や最適化を行ってオブジェクトコードを生成する. この際, 最適化が意図通りに行われているかのテスト (性能テスト) は, 主としてコンパイラ開発者が作成したテストコードにより行われるため, テストのバリエーションを十分に増やせないことが課題となる.

コンパイラの性能テストを自動化する方法としては, ランダムに生成した C プログラムから生成されるアセンブリコードの比較に基づく方法が提案されている [5]〜[7]. しかしこの方法では, フロントエンドやミドルエンドによる変換や最適化により, バックエンドの特定の最適化パスを所望の条件でテストするテストケースが, ごく低確率でしか, あるいは全く生成されないことがあり, バックエンドのテストには必ずしも適していない.

そこで本研究では, 既存のバックエンド最適化のテストプログラムから多数のミュータントを生成することにより, 自動的にテストのバリエーションを増やす手法を提案する. ミューテーションは, 冗長コードの挿入等, 元プログラムの機能を変えないものに限定するため, エラー判定は生成されるアセンブリコードを機械的に比較することにより行える. 本手法に基づくシステムを Perl 5 を用いて実装した結果, LLVM 6.0.0 で最適化処理の改善の参考になると思われるテストケースを 2 件検出することができた.

以下, 本稿では 2 章で LLVM の構造について述べ, 3 章で本手法に基づくシステム及び, ミューテーション操作について述べる. 4 章で実装及び実験について述べた後, 5 章でまとめと今後の課題を述べる.

2. コンパイラとそのテスト

2.1 コンパイラ基盤 LLVM

LLVM [8] は, コンパイラの開発に必要なライブラリ, プラグインなどを含む一連のソフトウェア群からなるコンパイラ基盤である. LLVM の構造を図 1 に示す. フロントエンドは C 言語や Swift 等の高級言語に対して字句解析と構文解析を行い, 中間表現である LLVM IR を生成する. ミドルエンドでは, 言語や CPU アーキテクチャとは独立した最適化を行う. バックエンドは, ミドルエンドから受け取った LLVM IR に対し, ターゲットのアーキテクチャに対応した命令選択やレジスタ割り当て, 命令スケジューリング等を行い, オブジェクトコードの生成を行う.

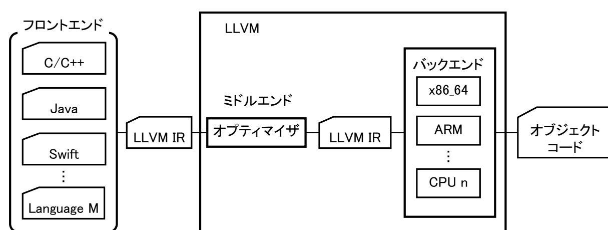

図 **1:** コンパイラ基盤 LLVM

LLVM は特定の CPU アーキテクチャに依存しない独立した仕様の中間表現 LLVM IR を規定し, フロントエンド, ミドルエンド, バックエンドの間のプログラム交換に使用している. LLVM IR の命令の多くはアセンブリに似た 3 番地コードであり, 変数はレジスタに保存される. LLVM IR は無限個の仮想レジスタを持つレジスタマシンを仮定した中間表現で, 基本的には SSA (静的単一代入, Static Single Assignment) 形式で表現される.

LLVM IR の例を図 2 に示す. このコードは変数 `a`, `b`, `c` を用いて `c = a + b` の演算を行うものである. 10–16 行目で変数の領域確保と値の格納を行い, 17–20 行目で値の呼び出しと計算, 計算結果の格納を行っている.

```
01: ; ModuleID = 'prog.c'
02: source_filename = "prog.c"
03: target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
04: target triple = "x86_64-apple-macosx10.11.0"
05:
06: ; Function Attrs: noinline nounwind ssp uwtable
07: define i32 @main() #0 {
08: entry:
09:   %retval = alloca i32, align 4
10:   %a = alloca i32, align 4
11:   %b = alloca i32, align 4
12:   %c = alloca i32, align 4
13:   store i32 0, i32* %retval, align 4
14:   store i32 -1, i32* %a, align 4
15:   store i32 110, i32* %b, align 4
16:   store i32 0, i32* %c, align 4
17:   %0 = load i32, i32* %a, align 4
18:   %1 = load i32, i32* %b, align 4
19:   %add = add i32 %0, %1
20:   store i32 %add, i32* %c, align 4
21:   ret i32 0
22: }
```

図 **2:** LLVM IR の例

2.2 バックエンドの最適化性能のテスト

新しい命令セットアーキテクチャ用のコンパイラを開発する場合には, 図 1 のバックエンドのみ開発すれば良いが, この場合には開発したバックエンドのテストを行うことが重要となる. バックエンドでは命令選択やレジスタ割り当て等, コード生成のために必要な変換の他に, ターゲット依存の最適化やこれらの変換の結果生じる冗長なコードを削除するための最適化が行われる. そのため, 正しいコードが生成されているかのみならず, 最適化パスが開発者の意図通りの動作をしているかの性能テストも重要になる.

バックエンドの最適化のテストは, バックエンドに直接入力可能なテストプログラム (主として LLVM IR) を用い, 生成されるアセンブリコードを確認することにより行われる. 図 3 は命令選択に関するテストであり, このプログラムから生成されたアセンブリコード (図 4) において, 29 行目の `select` 命令から生成される条件判定命令に, `testl` (32bit 演算) ではなく, `testb` (8bit 演算) が選択されるかどうかをテストしている.

このようなテストプログラムは, 最適化パスを開発するたびに手動で作成される. 当該の最適化が行われるべき条件の網羅や, 他の最適化との干渉により問題が生じないことを確認するためには膨大なテストケースが必要となるが, このようなテストケースの用意には多大な労力が必要となる.

3. LLVM バックエンドの最適化性能テストのミュータント生成

3.1 テスト手法の概要

本稿では, LLVM バックエンドの性能テストを行う既存のテストプログラムから自動的にミュータントを生成し, 生成されるアセンブリコードの比較によりエラー判定を行うテスト手法を提案する.

本手法の流れを図 5 に示す. テストプログラムを本システムへ入力し, これにミューテーション操作を施すことによって複数の等価なミュータントを生成する. 生成したミュータントを LLVM バックエンドに入力してアセンブリコードを生成し, これをテストプログラムから生成されたアセンブリコードと比較する. 本手法のミューテーション操作は, 元プログラムの意味を変えないように行うため, 生成されるアセンブリコードに大

```
01: ; RUN: llc < %s -mtriple=x86_64-apple-macosx | FileCheck %s
02:
03: ; Cmp lowering should not look past the truncate unless the
      high bits are known
04: ; zero.
05: ; rdar://12027825
06:
07: define void @foo(i8 %arg4, i32 %arg5, i32* %arg14) nounwind {
08: bb:
09: ; CHECK-LABEL: foo:
10: ; CHECK-NOT: testl
11: ; CHECK: testb
12:   %tmp48 = zext i8 %arg4 to i32
13:   %tmp49 = and i32 %tmp48, 32
14:   %tmp50 = add i32 %tmp49, 1593371643
15:   %tmp55 = sub i32 %tmp50, 0
16:   %tmp56 = add i32 %tmp55, 7787538
17:   %tmp57 = xor i32 %tmp56, 1601159181
18:   %tmp58 = xor i32 %arg5, 1601159181
19:   %tmp59 = and i32 %tmp57, %tmp58
20:   %tmp60 = add i32 %tmp59, -1263900958
21:   %tmp67 = sub i32 %tmp60, 0
22:   %tmp103 = xor i32 %tmp56, 13
23:   %tmp104 = trunc i32 %tmp103 to i8
24:   %tmp105 = sub i8 0, %tmp104
25:   %tmp106 = add i8 %tmp105, -103
26:   %tmp113 = sub i8 %tmp106, 0
27:   %tmp114 = add i8 %tmp113, -72
28:   %tmp141 = icmp ne i32 %tmp67, -1263900958
29:   %tmp142 = select i1 %tmp141, i8 %tmp114, i8 undef
30:   %tmp143 = xor i8 %tmp142, 81
31:   %tmp144 = zext i8 %tmp143 to i32
32:   %tmp145 = add i32 %tmp144, 2062143348
33:   %tmp152 = sub i32 %tmp145, 0
34:   store i32 %tmp152, i32* %arg14
35:   ret void
36: }
```

図 **3:** 2012-08-07-CmpISelBug.ll

```
01:   .section  __TEXT,__text,regular,pure_instructions
02:   .macosx_version_min 10, 11
03:   .globl  _foo
04:   .p2align  4, 0x90
05: _foo:                                    ## @foo
06: ## BB#0:                                 ## %bb
07:   andl  $32, %edi
08:   orl  $1601159181, %edi        ## imm = 0x5F6FC00D
09:   andl  %edi, %esi
10:   shrl  $5, %esi
11:   testb  %sil, %sil
12:   jne  LBB0_1
13: ## BB#2:                        ## %bb
14:                                 ## implicit-def: %AL
15:   jmp  LBB0_3
16: LBB0_1:
17:   xorl  $13, %edi
18:   movb  $81, %al
19:   subb  %dil, %al
20: LBB0_3:                         ## %bb
21:   xorb  $81, %al
22:   movzbl  %al, %eax
23:   addl  $2062143348, %eax       ## imm = 0x7AE9CF74
24:   movl  %eax, (%rdx)
25:   retq
26:
27: .subsections_via_symbols
```

図 **4:** 2012-08-07-CmpISelBug.s

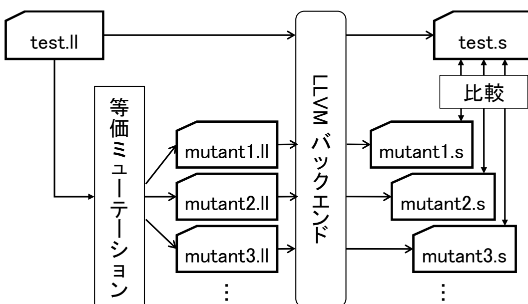

図 **5:** バックエンドの最適化のテストの流れ

きな違いが生じる場合には, バックエンドの最適化が意図通り機能していない可能性があると判定できる.

3.2 ミューテーション操作

本手法では, 元プログラムに不要な演算命令の挿入や, 演算結果を変えないような命令の書き換えを行って, ミュータントを生成する. 本手法で適用するミューテーション操作を以下に示す.

3.2.1 不要命令の挿入

この操作では, 演算結果が他の命令で参照されることがない命令を挿入する. 例えば, 図 6 において下線の命令が挿入した不要命令である. `%m1`, `%m2`, `%s1`, `%s2` はプログラム中のどこからも参照されないため, プログラムの動作は変わらない, 即ちこのプログラムの動作は元プログラムと等価である. これらの命令はバックエンドで削除されるため, ミュータントから生成されるアセンブリは基本的に元プログラムのものと変わらないはずである.

本手法では, `add` や `mul` 等の算術演算命令と, `load` や `store` のようなメモリアクセス命令を挿入する. 算術演算命令のオペランドは, 定数値かそれまでに定義されている仮想レジスタからランダムに選択する. メモリアクセス命令の場合は, 図 6 の 20–22 行目のように, スタック領域を確保する `alloca` 命令も併せて挿入する.

ただし, オペランドをランダムに選んでいるため, 未定義動作が発生することがある. このような命令がエラーの原因となった場合には, エラー検出後の解析時に目視で排除する.

```
09:  ...
10:  %retval = alloca i32, align 4
11:  %F = alloca %struct.anon, align 16
12:  %K = alloca %0, align 4
13:  store i32 0, i32* %retval
14:  %0 = bitcast %0* %K to i32*
15:  %1 = load i32, i32* %0, align 4
16:  %m1 = load i32, i32* %retval align 4
17:  %2 = and i32 %1, -121
18:  %m2 = add i32 -120, -1
19:  %3 = or i32 %2, 32
20:  %s1 = alloca i32, align 4
21:  store i32 10, i32* %s1, align 4
22:  %s2 = load i32, i32* %s1, align 4
23:  store i32 %3, i32* %0, align 4
24:  %4 = bitcast %0* %K to i32*
25:  %5 = load i32, i32* %4, align 4
26:  %6 = lshr i32 %5, 3
27:  ...
```

図 **6:** 計算結果を使わない演算の挿入

3.2.2 死亡コードの挿入

この操作では, 実行されない基本ブロックの挿入を行う. 図 7 の例において, 下線部が挿入した死亡コードである. 14 行目の比較演算の結果は恒真であり, 15 行目の分岐命令は 23 行目の `true` ブロックに制御を移すため, 17–21 行目の `dead` ブロックのコードは実行されない. バックエンドにはこのようなコードを削除する最適化パスが実装されている.

死亡コードにはランダムな命令列を生成する. 分岐命令には, 無条件分岐か, 分岐条件が恒真となる条件分岐を用いる.

```
13:  ...
14:  %16 = load i32, i32* %retval
15:  %cmp = icmp eq i32 1, 1
16:  br i1 %cmp, label %true, label %trash
17:  dead: ; preds = %entry
18:  %m0 = add i32 %2, %3
19:  %m1 = sub i32 %12, %11
20:  %m2 = mul i32 %14, %15
21:  ret i32 %m2
22:  true: ; preds = %entry
23:  ret i32 %16
```

図 **7:** 死亡コードの挿入

3.2.3 演算強度の増強

この操作では, 計算結果を変えないように演算命令の強度を増強する. 図 8 の例では, シフト演算を等価な乗算に変換して

いる. プログラム内に存在する演算の内, 即値と即値, または変数と即値の演算のみ書き換えを行う.

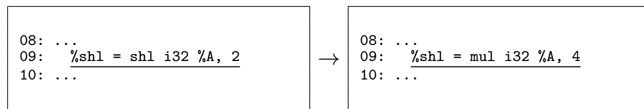

図 **8:** 演算強度の増強

3.2.4 型の増強

元プログラム内の整数型の演算の型を拡張し, より大きな型で計算を行うように変換する. 図 9 の例においては, 元プログラムの `i8` 型 (8bit の整数型) の演算を `i16` 型 (16bit の整数型) の演算へ変換し, オペランドと計算結果の型を合わせる型変換命令の挿入を行っている. バックエンドにはこのようなコードに対し, 型の軽減を行う最適化パスが実装されている.

本稿のミューテーション操作では, プログラム内の 128bit 未満の整数型の演算を, 元よりも大きな整数型または浮動小数型 (float 及び double) での計算に変換する.

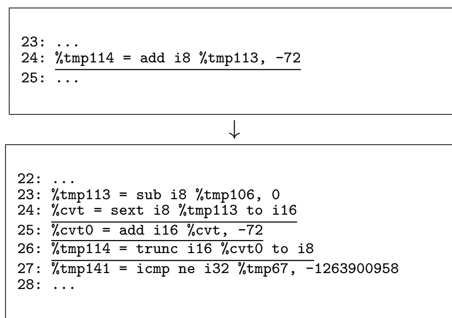

図 **9:** 型の増強

3.3 アセンブリの比較

本手法でのアセンブリコードの比較には, 文献 [7] と同様の手法を用いる. この手法では, 図 10 に示すように, 与えられた 2 つのアセンブリコードから, k 命令以上連続して命令が一致する部分を分離し, 不一致部分のコードの差を調べる. 不一致部分の対 (S_1, S_1'), (S_2, S_2'), $\cdots$ に対し, このうち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があると判定する. 不一致部分の比較は, 対応が取れない命令を抽出し, その重み和 (重みは命令実行サイクル数を想定して経験的に定める) を比較することにより行う. 図 10 の S_1, S_1' において, S_1 の `mov`, S_1' の `div`, `cmp`, `j` が対応の取れない命令であり, S_1 と S_1' の重み和はそれぞれ 1, 31 となる. この比が閾値未満であればアセンブリコードに差があると判定する. ただし, 本手法では, 重み和の比率ではなく, 差に基づいて判定を行う. これは, ミュータントと元プログラムのそれぞれから生成されたアセンブリコードが同じ, または一方が他方を内包する場合に比率を計算できなくなるためである.

4. 実装と実験

提案手法に基づくシステムを Perl 5 を用いて実装した. 本システムは, Mac OS X, Windows, Ubuntu Linux で動作する.

本稿では, GitHub の LLVM プロジェクト内にある 5 つのテストプログラム, 2011-03-02-DAGCombiner.ll, 2012-05-17-TwoAddressBug.ll, 2012-08-07-CmpISelBug.ll, 2012-10-03-DAGCycle.ll, imul.ll に対し, それぞれ 1,000 個のミュータントを生成して LLVM 6.0.0 の x86_64 ターゲットのバックエンドをテストした. 1 つのミュータントの生成には, 3.2 章で述べたミューテーション操作を合計 30 回行った. 2012-08-07-CmpISelBug.ll (図 3) を元に生成したミュータントの一例を図 11 に示す. 28–69 行目では死亡コードの挿入を, 23–25 行目と 74–76 行目では演算型の浮動小数型への変換を行っている.

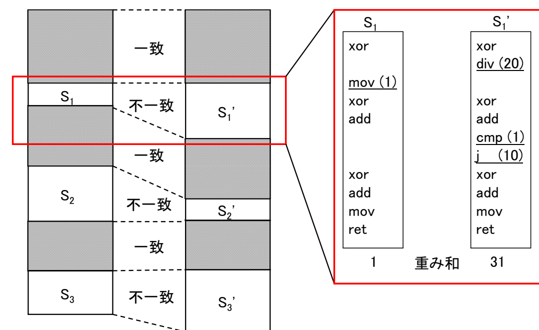

図 **10:** アセンブリコードの比較

表 1 (a) は, 今回実装した全てのミューテーション操作を用いて, テストを行った結果である. 表中の「#test」は生成したミュータントの総数である. 「#diff (>= +5)」, 「#diff (<= −2)」は, 生成したミュータントうちアセンブリコードの命令の重み和が 5 以上増加/ 2 以上減少した数を示す.

多くのミュータントでアセンブリコードに差が出たが, これらの主な原因は, バックエンドが冗長なコードを削除できないことであった. 例えば, LLVM 6.0.0 では条件分岐を用いた死亡コードは死亡コードと認識されず, また, `alloca` 命令を挿入すると, スタック領域を確保のためのアドレス計算命令が多く生成された. 浮動小数型から整数型への型緩和は行われていないため, アセンブリコードに差が現れた.

そこで, 死亡コードの挿入は無条件分岐によるものに限定し, `alloca` 命令の生成や, 浮動小数型への型変換を行わないように設定してテストを行った. 結果を表 1 (b) に示す. この実験では, 命令の重み和が 5 以上増えたミュータントが合計 4 件, 2 以上減少したミュータントが合計 58 件見つかった.

2012-08-07-CmpISelBug.ll (図 3) を元に生成したミュータントのうち, アセンブリコードの重み和が 5 以上増加していたミュータントを図 12 (a) に示す. このミュータントでは, 9–12 行目に死亡コードの挿入を行い, 7, 8, 13 行目, 24–26 行目では, 算術演算の型の拡張を行っている. 図 12 (b) は元プログラムとミュータントのアセンブリコードを比較した結果であり, 左側 (original.s) が元プログラムから生成されたアセンブリコード, 右側 (mutant.s) がミュータントから生成されたアセンブリコードである. この例では, ミュータントにおいて整数型の軽減に関する最適化が行われていない.

これは, 死亡コードの挿入により基本ブロックが細分されたことが原因であり, 多くのピープホール最適化が基本ブロック内に限定されていること, および死亡コードの削除はバックエンドの後半で (制御フロー最適化 (BranchFolding) の一環として) 行われることを考えれば, やむを得ない, 即ち不具合とは言えないと考えられる. しかし, LLVM では基本ブロックをまたがる命令選択 (GlobalISel) [9] の実装も進められており, このテストケースはその処理の参考になると考える. 死亡コードの削除はミドルエンドの最適化でも行われるため, 実際にこのようなテストケースがバックエンドの最適化パスに渡るかどうかは

```
01: ; RUN: llc < %s -mtriple=x86_64-apple-macosx | FileCheck %s
02: ; Cmp lowering should not look past the truncate unless the
      high bits are known
03: ; zero.
04: ; rdar://12027825
05: define void @foo(i8 %arg4, i32 %arg5, i32* %arg14) nounwind {
06: bb:
07:   ; CHECK-LABEL: foo:
08:   ; CHECK-NOT: testl
09:   ; CHECK: testb
10:   %tmp48 = zext i8 %arg4 to i32
11:   %tmp49 = and i32 %tmp48, 32
12:   %tmp50 = add i32 %tmp49, 1593371643
13:   %tmp55 = sub i32 %tmp50, 0
14:   %tmp56 = add i32 %tmp55, 7787538
15:   %tmp57 = xor i32 %tmp56, 1601159181
16:   %tmp58 = xor i32 %arg5, 1601159181
17:   %tmp59 = and i32 %tmp57, %tmp58
18:   %tmp60 = add i32 %tmp59, -1263900958
19:   %tmp67 = sub i32 %tmp60, 0
20:   %tmp103 = xor i32 %tmp56, 13
21:   %tmp104 = trunc i32 %tmp103 to i8
22:   %tmp105 = sub i8 0, %tmp104
23:   %cvt = sitofp i8 %tmp105 to float
24:   %cvt0 = fadd float %cvt, -103.000000
25:   %tmp106 = fptosi float %cvt0 to i8
26:   %tmp113 = sub i8 %tmp106, 0
27:   %tmp114 = add i8 %tmp113, -72
28:   br label %true
29: dead:
30:   %m_al = alloca i64, align 8
31:   %m = srem i128 -1263900958, -1917640
32:   %m_al0 = alloca i8, align 1
33:   %m0 = or i16 -72, 32
34:   %m_l = load i32, i32* %arg14
35:   %m_al1 = alloca i32, align 4
36:   store i32 0, i32* %m_al1, align 4
37:   %m_l0 = load i32, i32* %arg14
38:   %m1 = shl i8 %tmp106, %tmp104
39:   %m_l1 = load i32, i32* %arg14
40:   %m2 = ashr i64 1601159181, 32
41:   %m3 = add i8 %arg4, %tmp106
42:   %m4 = mul i32 %tmp67, %tmp50
43:   %m5 = lshr i16 0, 0
44:   %m6 = shl i16 -72, 32
45:   %m7 = add i8 %tmp113, %tmp114
46:   %m8 = srem i8 %tmp142, %tmp104
47:   %m_al2 = alloca i8, align 1
48:   store i8 -1263900958, i8* %m_al2, align 1
49:   %m9 = mul i128 0, -7818917
50:   %m_al3 = alloca i32, align 4
51:   store i32 81, i32* %m_al3, align 4
52:   %m_l2 = load i32, i32* %m_al3
53:   %m_al4 = alloca i64, align 8
54:   store i64 0, i64* %m_al4, align 8
55:   %m_l3 = load i64, i64* %m_al4
56:   %m10 = ashr i32 %arg5, %tmp49
57:   %m_al5 = alloca i8, align 1
58:   store i8 -1263900958, i8* %m_al5, align 1
59:   %m_l4 = load i8, i8* %m_al5
60:   %m11 = sdiv i16 -103, 13
61:   %m12 = udiv i16 0, 0
62:   %m_al6 = alloca i16, align 2
63:   %m13 = and i128 2062143348, -7818917
64:   %m14 = add i128 81, -1263900958
65:   %m_al7 = alloca i8, align 1
66:   store i8 81, i8* %m_al7, align 1
67:   %m15 = ashr i16 0, 0
68:   ret void
69: true:
70:   %tmp141 = icmp ne i32 %tmp67, -1263900958
71:   %tmp142 = select i1 %tmp141, i8 %tmp114, i8 undef
72:   %tmp143 = xor i8 %tmp142, 81
73:   %tmp144 = zext i8 %tmp143 to i32
74:   %cvt1 = uitofp i32 %tmp144 to double
75:   %cvt2 = fadd double %cvt1, 2062143348.000000
76:   %tmp145 = fptosi double %cvt2 to i32
77:   %tmp152 = sub i32 %tmp145, 0
78:   store i32 %tmp152, i32* %arg14
79:   ret void
80: }
```

図 **11:** 2012-08-07-CmpISelBug.ll から生成したミュータント

わからないが, 今後の最適化の強化や最適化パスの構成の更新によっては, このようなテストケースが有用となることも考えられる.

図 13 は, 逆にミュータントのアセンブリの重み和の方が小さくなった例である. このミュータント (図 13 (a)) では, 19–22 行目に死亡コードの挿入を行っている. 元プログラムのアセンブリコード (original.s) とこのミュータントのアセンブリコード (mutant.s) を比較すると (図 13 (b)), ミュータントの方が分岐命令が少ない.

このコードの変化は, やはり死亡コードの挿入によって基本ブロック分割の変化によって生じたものであるが, ここからバックエンドの最適化に関する改善の余地を指摘できる.

まず, mutant.s において, 16 行目の条件分岐命令 (`jne`) は不要であり, その削除ができていないことが指摘できる. これを削除すると, mutant.s の分岐命令はなくなり, 元の LLVM コードから分岐命令を含まないアセンブリを生成可能なことがわかるが, original.s においては分岐命令が 2 つ生成されている (12–15 行目).

このうち, 15 行目の `jmp .LBB0_3` は, 12 行目の “`jne .LBB0_1`” を “`je .LBB0_3`” とすれば削除可能である.

これらの命令を削除したアセンブリコード (図 13 (c)) の差は, LLVM IR コードの 24 行目の `select` 命令の未定義値オペランド (`undef`) に対するコードの生成に起因するものである. どちらのコードが高速かについては議論の余地があるが, 元プログラムとミュータントのいずれからも高速と判断される方のアセンブリコードが生成されることが望ましいと考えられる.

表 **1:** テストの実行結果

(a) 全てのミューテーション操作を使用

元プログラム	#test	#diff	
		>= +5	<= −2
2011-03-02-DAGCombiner.ll	1,000	347	0
2012-05-17-TwoAddressBug.ll	1,000	348	0
2012-08-07-CmpISelBug.ll	1,000	641	7
2012-10-03-DAGCycle.ll	1,000	376	0
imul.ll	1,000	632	0

(b) 無条件分岐の死亡コード挿入及び型変換のみ

元プログラム	#test	#diff	
		>= +5	<= −2
2011-03-02-DAGCombiner.ll	1,000	0	0
2012-05-17-TwoAddressBug.ll	1,000	0	0
2012-08-07-CmpISelBug.ll	1,000	1	58
2012-10-03-DAGCycle.ll	1,000	0	0
imul.ll	1,000	3	0

この 2 件のテストケースが検出しているのは, 必ずしも元のテストケースが意図していたものではない. しかし, このようなバックエンド最適化の改善点を見出すためのテストケースは, 通常は多数のアセンブリコードを開発者が目視調査して検出しているため, この作業をある程度自動化できるという点で, 本手法は有効であると考える.

5. むすび

本稿では, LLVM バックエンドの最適化性能をテストするために, 既存のテストプログラムから機能等価なミュータントを自動生成し, アセンブリを比較する手法を提案した. 実験の結果, LLVM 6.0.0 において最適化処理の改善の参考になると思われるテストケースを 2 件検出することができた.

今後の課題としては, 現在のアセンブリコード比較の手法では, 演算の型の変化など, アセンブリコードのわずかな変化を検出できないため, 比較方法の改良を行うことがあげられる. また, 新たなミューテーション操作を実装することにより, 生成可能なテストパターンを増加させることがあげられる.

```
01: define void @foo(i8 %arg4, i32 %arg5, i32* %arg14) nounwind {
02: bb:
03:   %tmp48 = zext i8 %arg4 to i32
04:   %tmp49 = and i32 %tmp48, 32
05:   %tmp50 = add i32 %tmp49, 1593371643
06:   %tmp55 = sub i32 %tmp50, 0
07:   %cvt9 = sext i32 %tmp55 to i128
08:   %cvt10 = add i128 %cvt9, 7787538
09:   br label %true
10: dead:
11:   ret void
12: true:
13:   %tmp56 = trunc i128 %cvt10 to i32
14:   %tmp57 = xor i32 %tmp56, 1601159181
15:   %tmp58 = xor i32 %arg5, 1601159181
16:   %tmp59 = and i32 %tmp57, %tmp58
17:   %tmp60 = add i32 %tmp59, -1263900958
18:   %tmp67 = sub i32 %tmp60, 0
19:   %tmp103 = xor i32 %tmp56, 13
20:   %tmp104 = trunc i32 %tmp103 to i8
21:   %tmp105 = sub i8 0, %tmp104
22:   %tmp106 = add i8 %tmp105, -103
23:   %tmp113 = sub i8 %tmp106, 0
24:   %cvt = sext i8 %tmp113 to i16
25:   %cvt0 = add i16 %cvt, -72
26:   %tmp114 = trunc i16 %cvt0 to i8
27:   %tmp141 = icmp ne i32 %tmp67, -1263900958
28:   %tmp142 = select i1 %tmp141, i8 %tmp114, i8 undef
29:   %tmp143 = xor i8 %tmp142, 81
30:   %tmp144 = zext i8 %tmp143 to i32
31:   %tmp145 = add i32 %tmp144, 2062143348
32:   %tmp152 = sub i32 %tmp145, 0
33:   store i32 %tmp152, i32* %arg14
34:   ret void
35: }
```

(a) テストプログラム (下線部が挿入命令)

```
+---+--------------------------------+---+--------------------------------+
|   | original.s                     |   | mutant.s                       |
+---+--------------------------------+---+--------------------------------+
|  5|_foo:                           |  5|_foo:                           |
|  6|## BB#0:                        |  6|## BB#0:                        |
|  7|    andl    $32, %edi           |  7|    andl    $32, %edi           |
*  8|    orl     $1601159181, %edi   *  8|    addl    $1593371643, %edi   *
*  9|    andl    %edi, %esi          *  9|    movslq  %edi, %rax          *
* 10|    shrl    $5, %esi            * 10|    addq    $7787538, %rax      *
* 11|    testb   %sil, %sil          * 11|    movl    %eax, %ecx          *
|   |                                * 12|    xorl    $1601159181, %ecx   *
|   |                                * 13|    xorl    $1601159181, %esi   *
|   |                                * 14|    testl   %esi, %ecx          *
| 12|    jne     LBB0_1              | 15|    jne     LBB0_1              |
* 13|## BB#2:                        * 16|## BB#2:                        *
| 14|                                | 17|                                |
| 15|    jmp     LBB0_3              | 18|    jmp     LBB0_3              |
| 16|LBB0_1:                         | 19|LBB0_1:                         |
* 17|    xorl    $13, %edi           * 20|    xorl    $13, %eax           *
* 18|    movb    $81, %al            * 21|    movb    $-103, %cl          *
* 19|    subb    %dil, %al           * 22|    subb    %al, %cl            *
* 20|LBB0_3:                         * 23|    movzbl  %cl, %eax           *
|   |                                * 24|    addl    $-72, %eax          *
|   |                                * 25|LBB0_3:                         *
| 21|    xorb    $81, %al            | 26|    xorb    $81, %al            |
| 22|    movzbl  %al, %eax           | 27|    movzbl  %al, %eax           |
| 23|    addl    $2062143348, %eax   | 28|    addl    $2062143348, %eax   |
+---+--------------------------------+---+--------------------------------+
```

(b) アセンブリコードの比較結果

図 **12:** 最適化不足の検出例

謝　辞

本研究に関して御助言を頂いた元関西学院大学の藤原大輔氏 (現 NTT コミュニケーションズ株式会社) に感謝致します. また, 本研究を行うにあたり, 御助言や御協力を頂いた, 関西学院大学理工学部石浦研究室の諸氏に感謝致します.

文　献

[1] GCC, the GNU Compiler Collection (online), `https://gcc.gnu.org/` (accessed 2017-11-14).

[2] Clang: A C language family frontend for LLVM (online), `http://clang.llvm.org/` (accessed 2017-11-14).

[3] Swift (online), `https://developer.apple.com/swift/` (accessed 2017-11-14).

[4] LegUp (online), `http://legup.eecg.utoronto.ca` (accessed 2017-11-14).

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

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

[7] 北浦幸太, 石浦菜岐佐: "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," 情報処理学会 *DA シンポジウム 2017*, pp. 39–44 (Aug. 2017).

[8] The LLVM Compiler Infrastructure (online), `http://llvm.org/` (accessed 2017-11-14).

[9] GlobalISel (online), `https://llvm.org/docs/GlobalISel.html` (accessed 2017-11-14).


```
01: define void @foo(i8 %arg4, i32 %arg5, i32* %arg14) nounwind {
02: bb:
03:   %tmp48 = zext i8 %arg4 to i32
04:   %tmp49 = and i32 %tmp48, 32
05:   %tmp50 = add i32 %tmp49, 1593371643
06:   %tmp55 = sub i32 %tmp50, 0
07:   %tmp56 = add i32 %tmp55, 7787538
08:   %tmp57 = xor i32 %tmp56, 1601159181
09:   %tmp58 = xor i32 %arg5, 1601159181
10:   %tmp59 = and i32 %tmp57, %tmp58
11:   %tmp60 = add i32 %tmp59, -1263900958
12:   %tmp67 = sub i32 %tmp60, 0
13:   %tmp103 = xor i32 %tmp56, 13
14:   %tmp104 = trunc i32 %tmp103 to i8
15:   %tmp105 = sub i8 0, %tmp104
16:   %tmp106 = add i8 %tmp105, -103
17:   %tmp113 = sub i8 %tmp106, 0
18:   %tmp114 = add i8 %tmp113, -72
19:   br label %true
20: dead:
21:   ret void
22: true:
23:   %tmp141 = icmp ne i32 %tmp67, -1263900958
24:   %tmp142 = select i1 %tmp141, i8 %tmp114, i8 undef
25:   %tmp143 = xor i8 %tmp142, 81
26:   %tmp144 = zext i8 %tmp143 to i32
27:   %tmp145 = add i32 %tmp144, 2062143348
28:   %tmp152 = sub i32 %tmp145, 0
29:   store i32 %tmp152, i32* %arg14
30:   ret void
31: }
```

(a) アセンブリの重み和が減少したミュータント

```
+---+--------------------------------+---+--------------------------------+
|   | original.s                     |   | mutant.s                       |
+---+--------------------------------+---+--------------------------------+
|  5|_foo:                           |  5|_foo:                           |
|  6|## BB#0:                        |  6|## BB#0:                        |
|  7|    andl    $32, %edi           |  7|    andl    $32, %edi           |
|  8|    orl     $1601159181, %edi   |  8|    orl     $1601159181, %edi   |
|  9|    andl    %edi, %esi          |  9|    andl    %edi, %esi          |
* 10|    shrl    $5, %esi            * 10|    andl    $32, %esi           *
* 11|    testb   %sil, %sil          * 11|    addl    $-1263900958, %esi  *
* 12|    jne     .LBB0_1             *   |                                |
* 13|# BB#2:                         *   |                                |
* 14|                                *   |                                |
* 15|    jmp     .LBB0_3             *   |                                |
* 16|.LBB0_1:                        *   |                                |
| 17|    xorl    $13, %edi           | 12|    xorl    $13, %edi           |
| 18|    movb    $81, %al            | 13|    movb    $81, %al            |
| 19|    subb    %dil, %al           | 14|    subb    %dil, %al           |
* 20|.LBB0_3:                        * 15|    cmpl    $-1263900958, %esi  *
|   |                                * 16|    jne     .LBB0_2             *
|   |                                * 17|# BB#1:                         *
|   |                                * 18|                                *
|   |                                * 19|.LBB0_2:                        *
| 21|    xorb    $81, %al            | 20|    xorb    $81, %al            |
| 22|    movzbl  %al, %eax           | 21|    movzbl  %al, %eax           |
| 23|    addl    $2062143348, %eax   | 22|    addl    $2062143348, %eax   |
+---+--------------------------------+---+--------------------------------+
```

(b) アセンブリコードの比較結果

```
+---+--------------------------------+---+--------------------------------+
|   | original.s                     |   | mutant.s                       |
+---+--------------------------------+---+--------------------------------+
|  5|_foo:                           |  5|_foo:                           |
|  6|## BB#0:                        |  6|## BB#0:                        |
|  7|    andl    $32, %edi           |  7|    andl    $32, %edi           |
|  8|    orl     $1601159181, %edi   |  8|    orl     $1601159181, %edi   |
|  9|    andl    %edi, %esi          |  9|    andl    %edi, %esi          |
* 10|    shrl    $5, %esi            * 10|    andl    $32, %esi           *
* 11|    testb   %sil, %sil          * 11|    addl    $-1263900958, %esi  *
* 12|    je     .LBB0_3              *   |                                |
* 13|# BB#2:                         *   |                                |
* 14|                                *   |                                |
* 15|                                *   |                                |
* 16|.LBB0_1:                        *   |                                |
| 17|    xorl    $13, %edi           | 12|    xorl    $13, %edi           |
| 18|    movb    $81, %al            | 13|    movb    $81, %al            |
| 19|    subb    %dil, %al           | 14|    subb    %dil, %al           |
* 20|.LBB0_3:                        * 15|    cmpl    $-1263900958, %esi  *
|   |                                * 16|                                *
|   |                                * 17|# BB#1:                         *
|   |                                * 18|                                *
|   |                                * 19|.LBB0_2:                        *
| 21|    xorb    $81, %al            | 20|    xorb    $81, %al            |
| 22|    movzbl  %al, %eax           | 21|    movzbl  %al, %eax           |
| 23|    addl    $2062143348, %eax   | 22|    addl    $2062143348, %eax   |
+---+--------------------------------+---+--------------------------------+
```

(c) 命令を変換/削除後のアセンブリコード

図 **13:** 最適化促進の検出例