ミュータント生成に基づくLLVMバックエンドの最適化性能テストにおけるエラー判定の強化

Enhanced Error Detection in Performance Test of LLVM Backend Based on Mutant Generation

神田諭志 [1]　石浦菜岐佐 [1]　西村啓成 [2]　福井昭也 [2]
Satoshi Kanda　Nagisa Ishiura　Masanari Nishimura　Akiya Fukui

[1] 関西学院大学 理工学部　School of Science and Technology, Kwansei Gakuin University
[2] ルネサス エレクトロニクス株式会社　Renesas Electronics Corporation

1　はじめに

コンパイラ基盤 LLVM [1] では, 機械依存のバックエンド (LLC) を開発すれば新しいプロセッサ用のコンパイラを作成できるが, その際にはバックエンドのテストが重要となる. 文献 [2] では LLC に対するテストプログラムからミュータントを生成することによってテストを強化する手法を提案している. この手法では, 生成されるアセンブリのコスト比較によりエラー判定を行っているが, 生成される命令の型等の微妙な相違は検出できなかった. 本稿では, LLVM のテストスイートで用いられている命令パターンの検査を併用することによってエラー判定を強化する手法を提案する.

2　LLVM バックエンドの最適化性能テスト

文献 [2] のテスト手法では, LLVM アセンブリで書かれた LLC の既存テストプログラムに対し, 計算結果を変化させない (従って LLC の最適化処理で削除される) ような書き換え (変異操作) を行って新たなテストプログラム (変異体) を得, LLC が生成するコードに差異が生じないかどうかを検査する. 変異操作は, 不要命令の挿入, 死亡コードの挿入, 演算型の拡張 (8bit 加算の 32bit 加算への変換等), 演算の増強 (シフトの乗算への変換等) 等である. 生成されるコードの比較は, 命令に定義した重みの和に基づいて行われ, 変異操作がコード全体の性能を低下させればそれを検出できるが, 個々の命令のオペランド長等の細かい差異は検出できない. 一方, LLVM のテストスイートでは, テストプログラムから生成されるべき/生成されてはならない命令のパターンを記述し, それを検査している. 個々の命令のアドレス指定方式やデータ型まで検査できるが, 想定しない命令が生成されてコード品質が低下することは検出できない.

3　パターンマッチングによるエラー判定の併用

本稿の手法では, 文献 [2] のエラー判定において LLVM テストスイートのパターンマッチングによる検査を併用する. LLC が生成するコードは基本的には変異操作の影響を受けないので, 検査用のパターン記述は元のテストプログラムのものをそのまま利用できる. ただし, 一部の変異操作は覗き穴最適化に影響を与え, 多くの変異体がこの検査でエラーとなることがあるため, そのような変異操作は除外する必要がある. 具体的な例としては, 死亡コードの挿入は変異操作から除外する.

4　実験結果

提案手法に基づくテストシステムを, [2] を拡張することにより実装した. 対象とした LLVM のバージョンは 4.0.0 である. LLVM テストスイートの 2012-08-07-CmpISelBug から生成した 110,000 件の変異体のうち 34 件でエラーを検出した. 図 1 の test.ll は元のテストプログラムであり, "→" の右側が挿入した変異である. 4~5 行目では testl ではなく testb が生成されるべきことを記述している. test.s と mutant.s は, それぞれ test.ll とその変異体から生成されたアセンブリである. 変異によって 7 行目で testb が生成できなくなったことが検出されているが, これは従来法では検出できなかったものである. 一方で, 全体としては mutant.s のほうが簡潔であり, LLC の最適化処理の改善を考える上で参考になるテストケースと考えられる.

test.ll (→ mutant.ll)

```
 1  define void @foo(i8 %a1, i32 %a2, i32* %a4) nounwind {
 2  bb:
 3    ; CHECK-LABEL: foo:
 4    ; CHECK-NOT: testl
 5    ; CHECK: testb
 6    %t1 = zext i8 %a1 to i32
 7    %t2 = and i32 %t1, 2
 8    %t3 = add i32 %t2, 4096
 9                          → %cvt1 = zext i32 %t3 to i128
10    %t4 = add i32 %t3, 1 → %cvt2 = add i128 %cvt1, 1
11                          → %t4 = trunc i128 %cvt2 to i32
12    %c1 = xor i32 %t4, 4097
13    %c2 = and i32 %c1, %a2
14    %c3 = icmp ne i32 %c2, 0
15    %v1 = xor i32 %t4, 1
16    %v2 = trunc i32 %v1 to i8
17                          → %cvt3 = zext i8 %v2 to i64
18    %v3 = add i8 %v2, 2 → %cvt4 = add i64 %cvt3, 2
19                          → %cvt5 = trunc i64 %cvt4 to i32
20                          → %v3 = trunc i32 %cvt5 to i8
21    %t5 = select i1 %c3, i8 %v3, i8 undef
22    %t6 = zext i8 %t5 to i32
23    store i32 %t6, i32* %a4
```

test.s

```
 1 foo:
 2 # BB #0:
 3   andl    $2, %edi
 4   orl     $4097, %edi
 5   andl    %edi, %esi
 6   shrl    %esi
 7   testb   %sil, %sil
 8   jne     .LBB0_1
 9 # BB #2:
10    jmp     .LBB0_3
11  .LBB0_1:
12    xorl    $1, %edi
13    addb    $2, %dil
14  .LBB0_3:
15    movzbl  %dil, %eax
16    movl    %eax, (%rdx)
```

→

mutant.s

```
 1 foo:
 2 # BB #0:
 3   andl    $2, %edi
 4
 5
 6
 7   testl   %esi, %edi
 8   jne     .LBB0_1
 9 # BB #2:
10    jmp     .LBB0_3
11 .LBB0_1:
12
13    addl    $2,%edi
14 .LBB0 _3:
15    movzbl  %dil, %eax
16    movl    %eax, (%rdx)
```

図 1　生成されるコードの差分を検出したミュータント

5　むすび

本稿では, 二つの判定基準を併用することによって LLVM バックエンドのテストを強化する手法を提案した. ミューテーション操作のバリエーションを増加させることが今後の課題としてあげられる.

参考文献

[1] https://llvm.org/.

[2] 田中, 石浦, 西村, 福井: "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," 信学技報, VLD2017-88 (Jan. 2018).