CS 131 Compilers: Discussion 12: Inheritence and Simple Optimization

杨易为 季杨彪 尤存翰
{yangyw,jiyb,youch}@shanghaitech.edu.cn

2021 年 5 月 24 日

1 Object Oriented Programming

Recall the *stub* trick from the lecture, an implementation strategy for multiple inheritance is listed as following:

```
class Base {
  public:
    int foo;
    virtual int f() {
      std::cout << this << std::endl;
      return this->foo;
    }
};

class Base2 {
  public:
     int bar;
     virtual int g() {
      std::cout << this << std::endl;
      return this->bar;
     }
};

class Derived : public Base, public Base2 {
  public:
     int baz;
     virtual int h() {
       std::cout << this << std::endl;
       return this->baz;
     }
};
```

To overcome this issue, compilers will move the this pointer so that each method sees what it expects. For simplicity, they actually create stub methods that will move the this pointer before calling the original method. For example, the compiler may generate

```
  Base2::g' () { move_this_pointer(); g(); }
```

1. Roughly sketch the class layout of the above classes (using the same 4-byte offsetconvention from the inheritance lecture). ForDerived, assumeh()is storedinside its own version ofBase' s vtable.
2. What will the following print (assuming there is no padding)?

```
Derived a;
std::cout << &a << std::endl; // Suppose this prints 0x10
a.f();
a.g();
a.h();
```

1.1 Exceptions and Code Generation

In this question we' ll look at how the set_jmp/long_jmp exception mechanism can be im-plemented. Consider a 32-bit computer architecture with 5 registers (r0, ...,r4):r0-2are used for function parameters,r3is used for the return address, andr4is used to storethe return value. Implement set_jmp(jmp_buf*)and long_jmp(jmp_buf*, int)in assembly(AT&T syntax). Assume that the jmp_buf is large enough, and that the second argument to long_jmp is never 0.

Answer:

1. Is it possible to longjmp to the same jmp_buf multiple times?
2. Write a program which throws an exception, handles it, and returns to the exceptionsite from the exception handler using set_jmp/long_jmp.

Answer:

1.2 Basic Optimization on IR

In GCC or optimization of Java code, we do some genuine optimization like const propogation, in LLVM IR, we literally do them in the IR. Here are the process to do so.

1.2.1 Mem2Reg

This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses. An alloca is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. This is just the standard SSA construction algorithm to construct "pruned" SSA form.

```
; RUN: opt < %s -instcombine -mem2reg -S | grep "%A = alloca"
; RUN: opt < %s -instcombine -mem2reg -S | \
; RUN:    not grep "%B = alloca"
; END.

; Ensure that instcombine doesn't sink the loads in entry/cond_true into
; cond_next.  Doing so prevents mem2reg from promoting the B alloca.

define i32 @test2(i32 %C) {
entry:
  %A = alloca i32
  %B = alloca i32
  %tmp = call i32 (...) @bar( i32* %A )              ; <i32> [#uses=0]
  %T = load i32, i32* %A             ; <i32> [#uses=1]
  %tmp2 = icmp eq i32 %C, 0          ; <i1> [#uses=1]
  br i1 %tmp2, label %cond_next, label %cond_true

cond_true:            ; preds = %entry
  store i32 123, i32* %B
  call i32 @test2( i32 123 )         ; <i32>:0 [#uses=0]
  %T1 = load i32, i32* %B            ; <i32> [#uses=1]
  br label %cond_next

cond_next:            ; preds = %cond_true, %entry
  %tmp1.0 = phi i32 [ %T1, %cond_true ], [ %T, %entry ]              ; <i32> [#uses=1]
  %tmp7 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp8 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp9 = call i32 (...) @baq( )             ; <i32> [#uses=0]
```

```
  %tmp10 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp11 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp12 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp13 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp14 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp15 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp16 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp17 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp18 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp19 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  %tmp20 = call i32 (...) @baq( )             ; <i32> [#uses=0]
  ret i32 %tmp1.0
}

declare i32 @bar(...)

declare i32 @baq(...)
```

1.2.2 SimplifyCFG

Performs dead code elimination and basic block merging. Specifically:

1. Removes basic blocks with no predecessors.
2. Merges a basic block into its predecessor if there is only one and the predecessor only has one successor.
3. Eliminates PHI nodes for basic blocks with a single predecessor.
4. Eliminates a basic block that only contains an unconditional branch.

```
; NOTE: Assertions have been autogenerated by utils/update_test_checks.py
; RUN: opt -instcombine -simplifycfg -S < %s

target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; #include <limits>
; #include <cstdint>
;
; using size_type = std::size_t;
; bool will_not_overflow(size_type size, size_type nmemb) {
;   return (size != 0 && (nmemb > std::numeric_limits<size_type>::max() / size));
; }

define i1 @will_not_overflow(i64 %arg, i64 %arg1) {
; SIMPLIFYCFG-LABEL: @will_not_overflow(
; SIMPLIFYCFG-NEXT:  bb:
; SIMPLIFYCFG-NEXT:    [[T0:%.*]] = icmp eq i64 [[ARG:%.*]], 0
; SIMPLIFYCFG-NEXT:    br i1 [[T0]], label [[BB5:%.*]], label [[BB2:%.*]]
; SIMPLIFYCFG:       bb2:
; SIMPLIFYCFG-NEXT:    [[T3:%.*]] = udiv i64 -1, [[ARG]]
; SIMPLIFYCFG-NEXT:    [[T4:%.*]] = icmp ult i64 [[T3]], [[ARG1:%.*]]
; SIMPLIFYCFG-NEXT:    br label [[BB5]]
; SIMPLIFYCFG:       bb5:
; SIMPLIFYCFG-NEXT:    [[T6:%.*]] = phi i1 [ false, [[BB:%.*]] ], [ [[T4]], [[BB2]] ]
; SIMPLIFYCFG-NEXT:    ret i1 [[T6]]
;
; INSTCOMBINEONLY-LABEL: @will_not_overflow(
; INSTCOMBINEONLY-NEXT:  bb:
; INSTCOMBINEONLY-NEXT:    [[T0:%.*]] = icmp eq i64 [[ARG:%.*]], 0
; INSTCOMBINEONLY-NEXT:    br i1 [[T0]], label [[BB5:%.*]], label [[BB2:%.*]]
; INSTCOMBINEONLY:       bb2:
; INSTCOMBINEONLY-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG]], i64 [[ARG1:%.*]])
; INSTCOMBINEONLY-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINEONLY-NEXT:    br label [[BB5]]
; INSTCOMBINEONLY:       bb5:
; INSTCOMBINEONLY-NEXT:    [[T6:%.*]] = phi i1 [ false, [[BB:%.*]] ], [ [[UMUL_OV]], [[BB2]] ]
; INSTCOMBINEONLY-NEXT:    ret i1 [[T6]]
;
; INSTCOMBINESIMPLIFYCFGONLY-LABEL: @will_not_overflow(
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:  bb:
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[T0:%.*]] = icmp eq i64 [[ARG:%.*]], 0
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG]], i64 [[ARG1:%.*]])
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[T6:%.*]] = select i1 [[T0]], i1 false, i1 [[UMUL_OV]]
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    ret i1 [[T6]]
;
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-LABEL: @will_not_overflow(
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:  bb:
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG:%.*]], i64 [[ARG1:%.*]])
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:    ret i1 [[UMUL_OV]]
;
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-LABEL: @will_not_overflow(
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:  bb:
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[T0:%.*]] = icmp eq i64 [[ARG:%.*]], 0
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG]], i64 [[ARG1:%.*]])
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[T6:%.*]] = select i1 [[T0]], i1 false, i1 [[UMUL_OV]]
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    ret i1 [[T6]]
;
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-LABEL: @will_not_overflow(
```

```
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:  bb:
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG:%.*]], i64 [[ARG1:%.*]])
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:    ret i1 [[UMUL_OV]]
;
bb:
  %t0 = icmp eq i64 %arg, 0
  br i1 %t0, label %bb5, label %bb2

bb2:                                              ; preds = %bb
  %t3 = udiv i64 -1, %arg
  %t4 = icmp ult i64 %t3, %arg1
  br label %bb5

bb5:                                              ; preds = %bb2, %bb
  %t6 = phi i1 [ false, %bb ], [ %t4, %bb2 ]
  ret i1 %t6
}

; Same as @will_not_overflow, but inverting return value.

define i1 @will_overflow(i64 %arg, i64 %arg1) {
; SIMPLIFYCFG-LABEL: @will_overflow(
; SIMPLIFYCFG-NEXT:  bb:
; SIMPLIFYCFG-NEXT:    [[T0:%.*]] = icmp eq i64 [[ARG:%.*]], 0
; SIMPLIFYCFG-NEXT:    br i1 [[T0]], label [[BB5:%.*]], label [[BB2:%.*]]
; SIMPLIFYCFG:       bb2:
; SIMPLIFYCFG-NEXT:    [[T3:%.*]] = udiv i64 -1, [[ARG]]
; SIMPLIFYCFG-NEXT:    [[T4:%.*]] = icmp ult i64 [[T3]], [[ARG1:%.*]]
; SIMPLIFYCFG-NEXT:    br label [[BB5]]
; SIMPLIFYCFG:       bb5:
; SIMPLIFYCFG-NEXT:    [[T6:%.*]] = phi i1 [ false, [[BB:%.*]] ], [ [[T4]], [[BB2]] ]
; SIMPLIFYCFG-NEXT:    [[T7:%.*]] = xor i1 [[T6]], true
; SIMPLIFYCFG-NEXT:    ret i1 [[T7]]
;
; INSTCOMBINEONLY-LABEL: @will_overflow(
; INSTCOMBINEONLY-NEXT:  bb:
; INSTCOMBINEONLY-NEXT:    [[T0:%.*]] = icmp eq i64 [[ARG:%.*]], 0
; INSTCOMBINEONLY-NEXT:    br i1 [[T0]], label [[BB5:%.*]], label [[BB2:%.*]]
; INSTCOMBINEONLY:       bb2:
; INSTCOMBINEONLY-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG]], i64 [[ARG1:%.*]])
; INSTCOMBINEONLY-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINEONLY-NEXT:    [[PHITMP:%.*]] = xor i1 [[UMUL_OV]], true
; INSTCOMBINEONLY-NEXT:    br label [[BB5]]
; INSTCOMBINEONLY:       bb5:
; INSTCOMBINEONLY-NEXT:    [[T6:%.*]] = phi i1 [ true, [[BB:%.*]] ], [ [[PHITMP]], [[BB2]] ]
; INSTCOMBINEONLY-NEXT:    ret i1 [[T6]]
;
; INSTCOMBINESIMPLIFYCFGONLY-LABEL: @will_overflow(
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:  bb:
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[T0:%.*]] = icmp eq i64 [[ARG:%.*]], 0
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG]], i64 [[ARG1:%.*]])
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[PHITMP:%.*]] = xor i1 [[UMUL_OV]], true
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    [[T6:%.*]] = select i1 [[T0]], i1 true, i1 [[PHITMP]]
; INSTCOMBINESIMPLIFYCFGONLY-NEXT:    ret i1 [[T6]]
;
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-LABEL: @will_overflow(
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:  bb:
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG:%.*]], i64 [[ARG1:%.*]])
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:    [[PHITMP:%.*]] = xor i1 [[UMUL_OV]], true
; INSTCOMBINESIMPLIFYCFGINSTCOMBINE-NEXT:    ret i1 [[PHITMP]]
;
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-LABEL: @will_overflow(
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:  bb:
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[T0:%.*]] = icmp eq i64 [[ARG:%.*]], 0
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG]], i64 [[ARG1:%.*]])
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[PHITMP:%.*]] = xor i1 [[UMUL_OV]], true
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    [[T6:%.*]] = select i1 [[T0]], i1 true, i1 [[PHITMP]]
; INSTCOMBINESIMPLIFYCFGCOSTLYONLY-NEXT:    ret i1 [[T6]]
;
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-LABEL: @will_overflow(
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:  bb:
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:    [[UMUL:%.*]] = call { i64, i1 } @llvm.umul.with.overflow.i64(i64 [[ARG:%.*]], i64 [[ARG1:%.*]])
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:    [[UMUL_OV:%.*]] = extractvalue { i64, i1 } [[UMUL]], 1
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:    [[PHITMP:%.*]] = xor i1 [[UMUL_OV]], true
; INSTCOMBINESIMPLIFYCFGCOSTLYINSTCOMBINE-NEXT:    ret i1 [[PHITMP]]
;
bb:
  %t0 = icmp eq i64 %arg, 0
  br i1 %t0, label %bb5, label %bb2

bb2:                                              ; preds = %bb
  %t3 = udiv i64 -1, %arg
  %t4 = icmp ult i64 %t3, %arg1
  br label %bb5

bb5:                                              ; preds = %bb2, %bb
  %t6 = phi i1 [ false, %bb ], [ %t4, %bb2 ]
  %t7 = xor i1 %t6, true
  ret i1 %t7
}
```