C++ devirtualization in clang (extra)

Piotr Padlewski

Wstęp do kodu LLVM

LLVM to niskopoziomowy kod pośredni - Intermediate form/representation - IR.

Charakteryzuje się formą SSA - static single assignment, czyli brak przypisywania wartości do tej samej zmiennej wiele razy.

Wstęp do kodu LLVM

Inną cechą charakteru jest to że jest on słabo typowalny - tz ma typy ale można się ich pozbyć castami (wskaźniki).

Charakteryzuje się także powtarzaniem typu wszędzie gdzie się da.

Wstęp do kodu LLVM

%name - zmienne

@name - zmienne globalne

!name - metadata

#liczba - atrybuty

; - początek komentarza

ret - return

call - call

alloca <Typ> - alokowanie na stosie

cast typu:

%ptr2 = bitcast <Typ>* %ptr, to <Typ2>*

wczytanie wartości z adresu:

load <Ty>, <Ty>* %ptr

zapisanie wartości pod adres

store <T> %val, <T>* %ptr

wyciągnięcie czegoś ze structa

getelementptr ...

funkcje specjalne - intrinsics

Funkcje bez definicji. Stworzone aby pomagać głównie w optymalizacjach:

@llvm.lifetime.start(...)

@llvm.lifetime.end(...)

Wstęp do kodu LLVM

```
void fun(int);
void fun2(const int&);

int main() {
    int p = 3;
    fun(p);
    fun2(p);
    return p;
}
```

kompilujemy:

```
clang++ plik.cc -S -emit-llvm -O2 -o plik.ll
```

Jeśli nie chcemy optymalizacji:

```
clang++ plik.cc -S -emit-llvm -O0
    -mllvm --disable-llvm-optzns -o plik.ll
```

Wstęp do kodu LLVM

```
define i32 @main() #0 {
  %p = alloca i32, align 4
  %1 = bitcast i32* %p to i8*
  call void @llvm.lifetime.start(i64 4, i8* %1) #3
  store i32 3, i32* %p, align 4, !tbaa !1
  tail call void @_Z3funi(i32 3)
  call void @_Z4fun2RKi(i32* nonnull dereferenceable(4) %p)
  %2 = load i32, i32* %p, align 4, !tbaa !1
  call void @llvm.lifetime.end(i64 4, i8* %1) #3
  ret i32 %2
}
```

Why devirtualization matters?

- Because we want to inline virtual functions!
- even if we will not inline, optimizer will take advantage of direct call
- we can make binaries smaller

Everything will lead to performance boost.

How virtual calls works

```
struct A {
  virtual void foo();
};
void g(A * a) {
  a->foo();
}
```

```
%vtable = load void (%struct.A*)**, void (%struct.A*)*** %a, align 8
%1 = load void (%struct.A*)*, void (%struct.A*)** %vtable, align 8
call void %1(%struct.A* %a)
```

Old devirtualization

```
struct A {
  virtual void foo();
};

void f() {
  A a;
  a.foo();
  a.foo();
}
```

Both will be direct call

Old devirtualization

```
struct A {
  virtual void foo();
};
void A::foo() { }
void g(A& a) {
  a.foo();
}

void f() {
  A a;
  g(a);
}
```

Works because g will be inlined

… and also because foo is defined in this TU.

Old devirtualization

```
struct A {
  A();
  virtual void foo();
};
void A::foo() { }
void g(A& a) {
  a.foo();
}
void f() {
  A a;
  g(a);
}
```

Constructor is now outline.

Won’t devirtualize because optimizer doesn’t know what is stored to vptr.

Old devirtualization

```
struct A {
  virtual void bar();
  virtual void foo();
};
void A::bar() { }
void g(A& a) {
  a.foo();
  a.foo();
}
void f() {
  A a;
  g(a);
}
```

Won’t devirtualize because optimizer doesn’t know that foo didn’t change vptr.

Old devirtualization

```
struct A {
  virtual void foo();
  virtual ~A() = default;
};
void g(A& a) {
  a.foo();
}
void f() {
  A a;
  g(a);
}
```

Won’t devirtualize because vtable is now external.

@vtable for A = external unnamed_addr constant [5 x i8*]

Calling the same virtual function

```
struct A {
  virtual void foo();
};
void g(A * a) {
  a->foo();
  a->foo();
}
```

```
%vtable = load  (…) %a
 %1 = load void (...) %vtable
 call void %1(%struct.A* %a)

%vtable2 = load  (…) %a
 %2 = load void (...) %vtable2
 call void %2(%struct.A* %a)
```

Placement new is evil

```
struct A {
  virtual void foo();
};
struct B : A {
  virtual void foo();
};
void A::foo() { new(this) B; } //in .cc
void g() {
  A *a = new A;
  a->foo();
  a->foo(); // Undefined behaviour
}
```

```
struct A {
  virtual A* foo();
};
struct B : A {
  virtual A* foo();
};
A* A::foo() { return new(this) B; }
void g() {
  A *a = new A;
  A *a2 = a->foo();
  a2->foo(); // this is fine
}
```

assuming type after ctor call

After calling the constructor, it would be very nice to know what is the vptr value.

```
call void @_ZN1AC1Ev(%struct.A* %a)

 %vtable = load i64*, i64** %a, align 8
 %cmp = icmp eq i64* %vtable, @vtable
 tail call void @llvm.assume(i1 %cmp)

 %vtable1 = load void (%struct.A*)**, void (%struct.A*)*** %a, align 8
```

available_externally vtables

```
@vtable for A = external unnamed_addr constant [6 x i8*]

@vtable for a = available_externally unnamed_addr constant [6 x i8*] (

    definition)
```

Unfortunately it is not safe to generate available_externally vtables in cases like:

- class has inline virtual functions
- class has hidden virtual functions - __attribute__((visibility("hidden")))

propagating “constness” of vptr

```
struct A {
  virtual void foo();
};
void g(A * a) {
  a->foo();
  a->foo();
}
```

```
%vtable = load  (…) %a
 %1 = load void (...) %vtable
 call void %1(%struct.A* %a)

%vtable2 = load  (…) %a
 %2 = load void (...) %vtable2
 call void %2(%struct.A* %a)
```

propagating “constness” of vptr

```
%vtable = load  (…) %a
 %1 = load void (...) %vtable
 call void %1(%struct.A* %a)

%vtable2 = load  (…) %a
 %2 = load void (...) %vtable2
 call void %2(%struct.A* %a)
```

```
%vtable = load  (…) %a
 %1 = load void (...) %vtable
 call void %1(%struct.A* %a)

 %2 = load void (...) %vtable
 call void %2(%struct.A* %a)
```

introducing !invariant.group

```
%vtable = load  (…) %a, !invariant.group !0
 %1 = load void (...) %vtable
 call void %1(%struct.A* %a)

%vtable2 = load  (…) %a, !invariant.group !0
 %2 = load void (...) %vtable2
 call void %2(%struct.A* %a)
```

Dealing with placement new

```
int main() {
  MyClass c;
  c.foo();
  auto c2 = new (&c) MyOtherClass();
  c2->foo();
  c2->~MyOtherClass();
  new (&c) MyClass();
}
```

introducing invariant.group.barrier

```
auto c2 = new (&c) MyOtherClass();
```

```
call void @_ZN7MyClassD1Ev(%struct.MyClass* nonnull %c) #1
%2 = bitcast %struct.MyClass* %c to %struct.MyOtherClass*
%3 = call @llvm.invariant.group.barrier(%2)
```

introducing -fstrict-vtable-pointers

For reasons like:

- LTO between code with **!invariant.group** and without is unsafe
- Optimizers don’t know how to deal with **invariant.group.barrier**
- **UBSan** doesn’t check for UBs from changing dynamic type
- Whole thing is still experimental.

Corner cases

```
void g() {
  A *a = new A;
  a->foo();
  A *b = new(a) B;
  assert(a == b);
  b->foo();
}
```

After assert llvm knows that a == b and can replace one with another.

Results

Unknown...

We know that devirtualization happen.

But the project is not yet done, there are so many things that we can still do, and the largest thing will come with !invariant. group.

Fun stats

LLVM:

- added lines: 1275 removed lines: 66 total lines: 1209

CLANG:

- added lines: 4109 removed lines: 1934 total lines: 2175

- Two 1.5h talks (SFBay C++, Google)
- Two 15min talks at cppcon
- 2kg gain

Sources

Devirtualization paper:

https://goo.gl/r5dOxR

How devirtualization works in gcc (*Jan Hubička*)

http://hubicka.blogspot.com/2014/01/devirtualization-in-c-part-1.html

https://www.youtube.com/watch?v=oN15LjHX9xY

Thank you

Richard Smith (host)

Nick Lewycky

David Majnemer
Reid Kleckner
Daniel Berlin

All folks from clang team