An investigation of missed opportunities in devirtualization

Yangguang Li, Szymon Sobieszek, Ehsan Amiri

www.huawei.com

HUAWEI TECHNOLOGIES CO., LTD.

Whole-program devirtualization in LLVM

typeinfo name

typeinfo

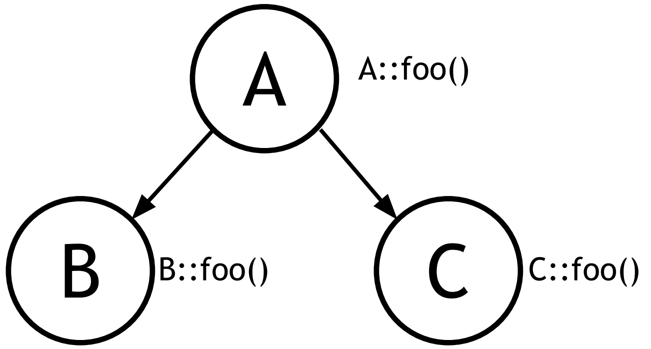
```
@_ZTI1A = hidden constant { ptr, ptr } { ................., ptr @_ZTS1A }, align 8

@_ZTS1A = hidden constant [3 x i8] c"1A\00", align 1

@_ZTV1A = hidden unnamed_addr constant { [13 x ptr] } { [13 x ptr] [..., ..., ..., ..., ..., ..., ..., ..., ..., ptr @_ZNK1A3fooEv, ..., ..., ...] }, align 8,
!type !5185, !type !5186, !type !5187, !type !5188, !type !5189, !type !5190, !type !5191, !vcall_visibility !72
```

Virtual table

Type metadata

Pointer to virtual function A::foo.
The offset is important

```
@_ZTV1A = [......, ......, ptr @_ZN1A3fooEv,  ......, ......]
@_ZTV1B = [......, ......, ptr @_ZN1B3fooEv,  ......, ......]
@_ZTV1C = [......, ......, ptr @_ZN1C3fooEv,  ......, ......]
```



```
class A {
public:
  virtual void foo() {}
};
class B : public A {
public:
  void foo() {}
};
class C : public A {
public:
  void foo() {}
};
```

WPD checks for every pair

(type info, vtable offset) whether all

function pointers in **compatible** vtables

are the same.

Whole-program devirtualization in LLVM

type info name

type info

```
@_ZTI1A = hidden constant { ptr, ptr } { ................., ptr @_ZTS1A }, align 8

@_ZTS1A = hidden constant [3 x i8] c"1A\00", align 1

@_ZTV1A = hidden unnamed_addr constant { [13 x ptr] } { [13 x ptr] [..., ..., ..., ..., ..., ..., ..., ..., ..., ptr @_ZNK1A3fooEv, ..., ..., ...] }, align 8,
!type !5185, !type !5186, !type !5187, !type !5188, !type !5189, !type !5190, !type !5191, !vcall_visibility !72
```

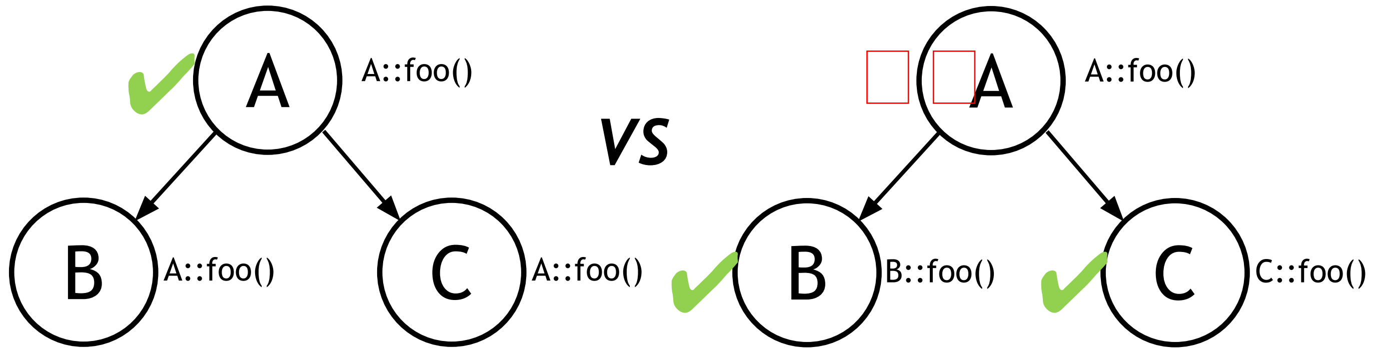
Virtual table

Type metadata

Typeinfo name of the class that defines the called function
C *c; c->foo() □ typeinfo name of A is used in the metadata

```
%vtable = load ptr, ptr %call, align 8, !tbaa !12271
%1 = tail call { ptr, i1 } @llvm.type.checked.load(ptr %vtable, i32 72, metadata !"_ZTSA"), !nosanitize !12273
%2 = extractvalue { ptr, i1 } %1, 0, !nosanitize !12273
invoke void %2(.........) .........
```

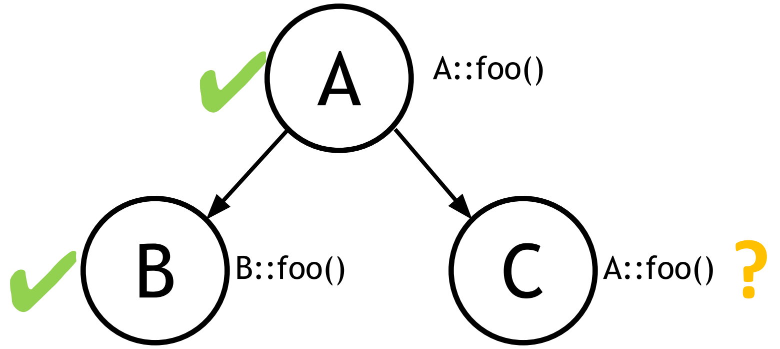
```
@_ZTV1A = [ptr null,
           ptr @_ZTI1A,
           ptr @_ZN1A3fooEv]
@_ZTV1B = [ptr null,
           ptr @_ZTI1B,
           ptr @_ZN1A3fooEv]
@_ZTV1C = [ptr null,
           ptr @_ZTI1C,
           ptr @_ZN1A3fooEv]
```



```
@_ZTV1A = [ptr null,
           ptr @_ZTI1A,
           ptr @_ZN1A3fooEv]
@_ZTV1B = [ptr null,
           ptr @_ZTI1B,
           ptr @_ZN1B3fooEv]
@_ZTV1C = [ptr null,
           ptr @_ZTI1C,
           ptr @_ZN1C3fooEv]
```

What kind of opportunities do we miss?

```
class A {
public:
  virtual void foo() {}
};
class B : public A {
public:
  void foo() {}
};
class C : public A {};
void bar(A *a, B *b, C *c)
{
  a->foo();
  b->foo();
  c->foo();
}
```



```
%vtable = load ptr, ptr %call, align 8, !tbaa !12271
%1 = tail call { ptr, i1 } @llvm.type.checked.load(ptr %vtable, i32 72, metadata !"_ZTSA")
%2 = extractvalue { ptr, i1 } %1, 0, !nosanitize !12273
invoke void %2(.........) .........
```

But can we rely on data type of C? What about non-strict aliasing and type punning.

Devirtualization already assumes data types of objects are reliable. It is easy to construct test cases that have different behavior with and without devirtualization.

Another idea: What if we keep track of type conversions in the source code?

What other opportunities do we miss?

- Example: We construct an object and pass it to a function as a pointer to a parent class.
- `Json_array`, `Json_object`, and `Json_string` are all leaf classes deriving from `Json_dom`.

```
void DB_restrictions::get_as_json(Json_array &restrictions_array) const {
    for (auto &revocations_itr : m_restrictions) {
      Json_array privileges;
      Json_object revocations_obj;
      Json_string db_name(revocations_itr.first.c_str());
      revocations_obj.add_clone(consts::Database, &db_name);
      ulong revokes_mask = revocations_itr.second;
      while (revokes_mask != 0){
        Json_string priv_str(get_one_priv(revokes_mask));
        privileges.append_clone(&priv_str);
      }
      revocations_obj.add_clone(consts::Privileges, &privileges);
      restrictions_array.append_clone(&revocations_obj);
    }
  }
```

```
bool add_clone(const std::string &key, const Json_dom *value) {
      return value == nullptr || add_alias(key, value->clone());
}
```

How often do these opportunities occur?

- Inlining exposes these devirtualization opportunities
  - Assuming the caller keeps some information about the constructor.
  - But devirtualization exposes inlining opportunity.
  - A simple and quick experiment: Do more devirtualization after LTO inlining and repeat inlining.
  - Number outside parenthesis: All extra devirtualization after keeping track of conversions.
  - Number inside the parentheses: extra devirtualizations when we keep track of constructor calls.

- Some of the simplest cases can be caught by CSE, if we add a CSE pass (very small fraction)
- There are multiple ways to generalize/tune this experiment.

	Number of devirtualized callsites		
Benchmark	Before LTO inlining	After LTO inlining	Perf gain
omnetpp_r	39 (14)	194 (92)	-
xalancbmk_r	30 (10)	120 (103)	1%
parest_r	24 (24)	20 (20)	-
MySQL	272 (207)	3312 (3118)	0.4%

Thank you

www.huawei.com