

DEVIRTUALIZATION IN LLVM

Piotr Padlewski

University of Warsaw

piotr.padlewski@gmail.com

IIIT

@PiotrPadlewski

CURRENT DEVIRTUALIZATION IN THE FRONTEND

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

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



DEVIRTUALIZATION BY THE FRONTEND

```
int test(A *a) {  
    // How to change it to a direct call here?  
    return a->foo();  
}
```

```
struct A {  
    virtual int foo () {return 42;}  
};
```

DEVIRTUALIZATION BY THE FRONTEND

```
int test(A *a) {  
    // How to change it to a direct call here?  
    return a->foo();  
}
```

```
struct A final {  
    virtual int foo () {return 42;}  
};
```



DEVIRTUALIZATION BY THE FRONTEND

```
int test(A *a) {  
    // How to change it to a direct call here?  
    return a->foo();  
}
```

```
struct A {  
    virtual int foo () final {return 42;}  
};
```



DEVIRTUALIZATION BY THE FRONTEND

```
int test(A *a) {  
    // How to change it to a direct call here?  
    return a->foo();  
}
```

```
namespace {  
    struct A {  
        virtual int foo () {return 42;}  
    };  
}
```

CURRENT DEVIRTUALIZATION IN THE MIDDLE END

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

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

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



CURRENT DEVIRTUALIZATION IN THE MIDDLE END

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

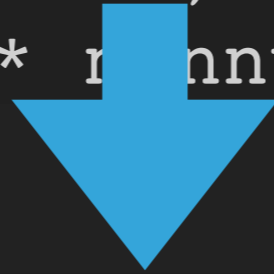
```
void g(A& a) {  
    a.foo();  
}
```

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



CURRENT DEVIRTUALIZATION IN THE MIDDLE END

```
call void @_ZN1AC1Ev(%struct.A* nonnull %1)
%3 = bitcast %struct.A* %1 to void (%struct.A*)***
%4 = load void (%struct.A**)**, void (%struct.A*)*** %3
%5 = load void (%struct.A*)*, void (%struct.A**)** %4
call void %5(%struct.A* nonnull %1)
```



```
%3 = getelementptr inbounds %struct.A, %struct.A*
    %1, i64 0, i32 0
store i32 (...)** {...} @_ZTV1A, {...} %3
call void @_ZN1A3fooEv(%struct.A* nonnull %1)
```

CURRENT DEVIRTUALIZATION IN THE MIDDLE END

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

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

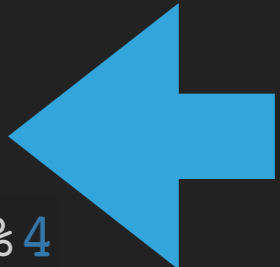
```
void f() {  
    A a;  
    g(a);  
}
```



CURRENT DEVIRTUALIZATION IN THE MIDDLE END

```
%1 = alloca %struct.A, align 8
%2 = bitcast %struct.A* %1 to i8*
call void @llvm.lifetime.start(i64 8, i8* %2)

%3 = getelementptr inbounds %struct.A, %struct.A* %1, i64 0, i32 0
store {...} @_ZTV1A {...}, %3
%4 = bitcast %struct.A* %1 to void (%struct.A*)***
call void @_ZN1A3fooEv(%struct.A* nonnull %1)
%5 = load void (%struct.A)**, void (%struct.A)*** %4
%6 = load void (%struct.A)*, void (%struct.A)** %5
call void %6(%struct.A* nonnull %1)
```



CURRENT DEVIRTUALIZATION IN THE MIDDLE END

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



CURRENT DEVIRTUALIZATION IN THE MIDDLE END

```
%3 = getelementptr inbounds %struct.A, %struct.A*  
    %1, i64 0, i32 0  
store {...} @_ZTV1A {...} %3  
%4 = load {...} @_ZTV1A {...}  
call void %4(%struct.A* nonnull %1)
```

CURRENT DEVIRTUALIZATION IN THE MIDDLE END

▶ Why isn't the vtable definition is visible?

▶ Why this **works**?

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

And this **doesn't**

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

PROBLEMS WITH ITANIUM ABI

- ▶ Surprisingly vtable will be external in both cases
- ▶ wheredidmyvtablego.com ? **Key function** optimization
 - ▶ „the first non-pure virtual function that is not inline at the point of class definition“
- ▶ But our key function is **foo**,
why does an inline virtual dtor break it?
- ▶ Why did it even work in the first example if it is external?

AVAILABLE_EXTERNALLY VTABLES

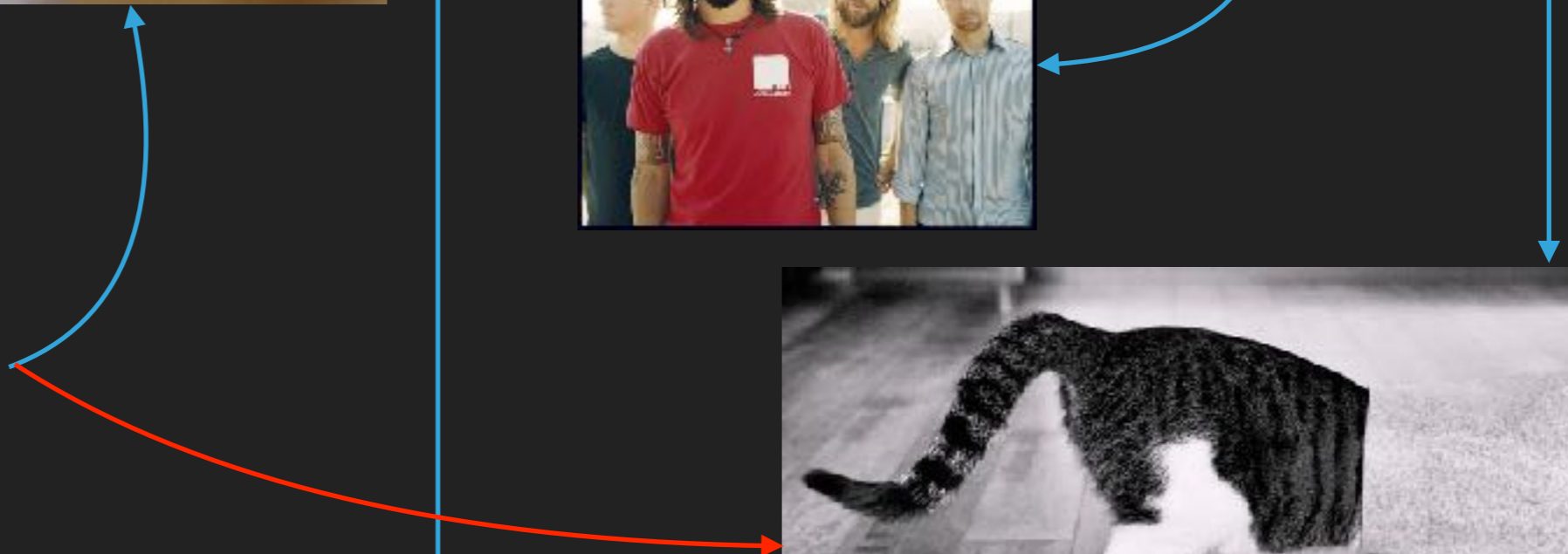
- ▶ It worked because it was possible to emit `available_externally vtable`
- ▶ It was not safe to do it in the second case because:
 - ▶ Itanium ABI doesn't guarantee that inline virtual function will be exported
 - ▶ We don't know if inline function was emitted and Clang is lazy about emitting inline functions

AVAILABLE_EXTERNALLY VTABLES



```
void f() {  
  A a;  
  g(a);  
  // a.~A();  
}
```

offset	rtti	A::foo	A::~~A()
--------	------	--------	----------



AVAILABLE_EXTERNALLY VTABLES

- ▶ `available_externally` vtables are not emitted if vtable contains one of:
 - ▶ inline method
 - ▶ hidden visibility method
- ▶ We could emit all inline virtual functions...
- ▶ If we would know that clang will emit all inline virtual functions, then it would be safe to emit vtable definition

THE HOLY GRAIL OF DEVIRTUALIZATION

vptr value



vtable definition

no vptr clobbering

C++ VIRTUAL FUNCTIONS SEMANTICS

▶ Can vptr really change?

▶ yes, it even changes multiple times during dynamic object construction

▶ But can it change after calling virtual function?

```
void Base::foo() { // virtual
    static_assert(sizeof(Base) == sizeof(Derived));
    new(this) Derived;
}
```

```
Base *a = new Base;
```

```
a->foo();
```

```
a->foo(); // Undefined behavior
```

C++ VIRTUAL FUNCTIONS SEMANTICS

```
Base* Base::foo() {  
    return new(this) Derived;  
}
```

```
Base *a = new Base;  
Base *a2 = a->foo();  
a2->foo(); // this is fine
```

- ▶ It is safe to assume that 2 virtual calls, using the same pointer, invoke the same function
- ▶ Be aware of other corner cases like constructors, placement new, unions...

TEACHING LLVM TO NOT CLOBBER VPTRS

- ▶ We want to say that vptr is invariant
- ▶ `llvm.invariant.{start|end}` doesn't fit - we don't know the start and the end
- ▶ `!invariant.load` metadata also doesn't work because vptr is not constant for the whole program
- ▶ We decided to add another metadata that would fit our needs - **`!invariant.group`** metadata

TEACHING LLVM TO NOT CLOBBER VPTRS

Semantics of !invariant.group !<MD>

- ▶ The invariant.group metadata may be attached to load/store instructions
- ▶ The existence of the invariant.group metadata on the instruction tells the optimizer that every **load** and **store** to the **same pointer operand** within the **same invariant group** can be assumed to load or store the same value
- ▶ Bitcasts don't affect when 2 pointers are considered the same

TEACHING LLVM TO NOT CLOBBER VPTRS

- ▶ What about cases like placement new, unions, ctors and dtors? Optimizer is smart enough to figure out these pointers are aliasing.
- ▶ declare `i8* @llvm.invariant.group.barrier(i8* <ptr>)`
- ▶ `invariant.group` returns the argument pointer, but the optimizer doesn't know about it
- ▶ It is added in every place where dynamic type can change

INVARIANT.GROUP STATUS

- ▶ Optimization works for single BB
- ▶ clang decorates vtable loads with `!invariant.group` with **-fstrict-vtable-pointers**, with pointer type mangled name as metadata
- ▶ it also add barriers in ctors/dtors and placement new

INVARIANT.GROUP PROBLEMS

- ▶ Adding barriers will add some misoptimizations
 - ▶ we could teach important passes how to look through barrier without breaking semantics
 - ▶ we could strip all invariant.group with barriers after devirtualization is done

INVARIANT.GROUP CORNER CASE

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

- ▶ we could add barrier like to compared pointers
barrier(a) == barrier(b)
- ▶ or we could add barrier for **a** and **b** after comparison

TEACHING LLVM ABOUT CONSTRUCTORS

```
call void @_ZN1AC1Ev(%struct.A* nonnull %a)
%1 = bitcast %struct.A* %a to i8***

%vtable = load {...} %1, !invariant.group !8
%cmp.vtables = icmp eq i8** %vtable,
               getelementptr {...} @_ZTV1A
call void @llvm.assume(i1 %cmp.vtables)

%vtable.i.cast = bitcast i8** %vtable to
                 i32 (%struct.A*)**
%2 = load {...} %vtable.i.cast
%call.i = call i32 @2(%struct.A* nonnull %a)
```

ASSUME LOADS STATUS

- ▶ There were big compile time regressions after emitting assume loads, mostly in InstCombine (AssumeCache)
- ▶ assume loads were „temporarily“ disabled by default requiring `-fstrict-vtable-pointers`
- ▶ Assume load can be the first reference to vtable in that module, causing similar problems as `available_externally` vtables. Not sure if still accurate.

DEMO

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

void call_foo(A *a) {
    a->foo();
    a->foo();
    a->foo();
}

void test() {
    A a;
    call_foo(&a);
}
```

```

define void @_Z8call_fooP1A(%struct.A* %a) {
    %0 = bitcast %struct.A* %a to i32 (%struct.A*)***
    %vtable = load {...} %0
    %1 = load {...} %vtable
    %call = tail call i32 @%1(%struct.A* %a)
    %vtable1 = load {...} %0
    %2 = load {...} %vtable1
    %call3 = tail call i32 @%2(%struct.A* %a)
    %vtable4 = load {...} %0
    %3 = load {...} %vtable4
    %call6 = tail call i32 @%3(%struct.A* %a)
    ret void
}

```

AFTER ALL OPTIMIZATIONS (WITHOUT -FSTRICT-VTABLE-POINTERS)


```
define void @_Z8call_fooP1A(%struct.A* %a) {
    %0 = bitcast %struct.A* %a to i32 (%struct.A*)***
    %vtable = load {...} %0, !invariant.group !8
    %1 = load {...} %vtable
    %call = tail call i32 @1(%struct.A* %a)
    %vtable1 = load {...} %0, !invariant.group !8
    %2 = load {...} %vtable1
    %call3 = tail call i32 @2(%struct.A* %a)
    %vtable4 = load {...} %0, !invariant.group !8
    %3 = load {...} %vtable4
    %call6 = tail call i32 @3(%struct.A* %a)
    ret void
}
```

```
define void @_Z8call_fooP1A(%struct.A* %a) {
    %0 = bitcast %struct.A* %a to i32 (%struct.A*)***
    %vtable = load {...} %0, !invariant.group !8
    %1 = load {...} %vtable
    %call = tail call i32 @1(%struct.A* %a)
    %2 = load {...} %vtable
    %call3 = tail call i32 @2(%struct.A* %a)
    %3 = load {...} %vtable
    %call6 = tail call i32 @3(%struct.A* %a)
    ret void
}
```

AFTER GVN (USING MEM DEP ANALYSIS)

```
define void @_Z8call_fooP1A(%struct.A* %a) {
    %0 = bitcast %struct.A* %a to i32 (%struct.A*)***
    %vtable = load {...} %0, !invariant.group !8
    %1 = load {...} %vtable, !invariant.load !9
    %call = tail call i32 @1(%struct.A* %a)
    %2 = load {...} %vtable, !invariant.load !9
    %call3 = tail call i32 @2(%struct.A* %a)
    %3 = load {...} %vtable, !invariant.load !9
    %call6 = tail call i32 @3(%struct.A* %a)
    ret void
}
```

```
define void @_Z8call_fooP1A(%struct.A* %a) {
    %0 = bitcast %struct.A* %a to i32 (%struct.A*)***
    %vtable = load {...} %0, !invariant.group !8
    %1 = load {...} %vtable, !invariant.load !9
    %call = tail call i32 @1(%struct.A* %a)
    %call3 = tail call i32 @1(%struct.A* %a)
    %call6 = tail call i32 @1(%struct.A* %a)
    ret void
}
```

```
define void @_Z4testv() {
    %a = alloca %struct.A, align 8
    %0 = bitcast %struct.A* %a to i8*
    call void @llvm.lifetime.start(i64 8, i8* %0)
    call void @_ZN1AC1Ev(%struct.A* nonnull %a)
    %1 = bitcast %struct.A* %a to i8***
    %vtable = load {...} %1, !invariant.group !8
    %cmp.vtables = icmp eq i8** %vtable,
                    getelementptr {...} @_ZTV1A {...}
    call void @llvm.assume(i1 %cmp.vtables)
    call void @_Z8call_fooP1A(%struct.A* nonnull %a)
    call void @llvm.lifetime.end(i64 8, i8* %0)
    ret void
}
```

```

define void @_Z4testv() {
  %a = alloca %struct.A, align 8
  %0 = bitcast %struct.A* %a to i8*
  call void @llvm.lifetime.start(i64 8, i8* %0)
  call void @_ZN1AC1Ev(%struct.A* nonnull %a)
  %1 = bitcast %struct.A* %a to i8***
  %vtable = load {...} %1, !invariant.group !8
  %cmp.vtables = icmp eq i8** %vtable,
    getelementptr {...} @_ZTV1A {...}
  call void @llvm.assume(i1 %cmp.vtables)
  %2 = bitcast %struct.A* %a to i32 (%struct.A*)***
  %vtable.i = load {...} %2, !invariant.group !8
  %3 = load {...} %vtable.i !invariant.load !9
  %call.i = call i32 %3(%struct.A* %a)
  %call3.i = call i32 %3(%struct.A* %a)
  %call6.i = call i32 %3(%struct.A* %a)
  call void @llvm.lifetime.end(i64 8, i8* %0)
  ret void
}

```

AFTER INLINING

```

define void @_Z4testv() {
  %a = alloca %struct.A, align 8
  %0 = bitcast %struct.A* %a to i8*
  call void @llvm.lifetime.start(i64 8, i8* %0)
  call void @_ZN1AC1Ev(%struct.A* nonnull %a)
  %1 = bitcast %struct.A* %a to i8***
  %vtable = load {...} %1, !invariant.group !8
  %cmp.vtables = icmp eq i8** %vtable,
    getelementptr {...} @_ZTV1A {...}
  call void @llvm.assume(i1 %cmp.vtables)
  %vtable.i.cast = bitcast i8** %vtable to i32 (%struct.A*)**
  %2 = load {...} %vtable.i.cast !invariant.load !9
  %call.i = call i32 %2(%struct.A* nonnull %a)
  %call3.i = call i32 %2(%struct.A* nonnull %a)
  %call6.i = call i32 %2(%struct.A* nonnull %a)
  call void @llvm.lifetime.end(i64 8, i8* nonnull %0) #3
  ret void
}

```

AFTER INST COMBINE

```

define void @_Z4testv() {
    %a = alloca %struct.A, align 8
    %0 = bitcast %struct.A* %a to i8*
    call void @llvm.lifetime.start(i64 8, i8* %0)
    call void @_ZN1AC1Ev(%struct.A* nonnull %a)
    %1 = bitcast %struct.A* %a to i8***
    %vtable = load {...} %1, !invariant.group !8
    %cmp.vtables = icmp eq i8** %vtable,
                    getelementptr {...} @_ZTV1A {...}
    call void @llvm.assume(i1 %cmp.vtables)
    %call.i = call i32 @_ZN1A3fooEv(%struct.A* nonnull %a)
    %call3.i = call i32 @_ZN1A3fooEv(%struct.A* nonnull %a)
    %call6.i = call i32 @_ZN1A3fooEv(%struct.A* nonnull %a)
    call void @llvm.lifetime.end(i64 8, i8* nonnull %0)
    ret void
}

```

AFTER GVN - GLOBAL VALUE NUMBERING

SPECULATIVE DEVIRTUALIZATION

- ▶ We speculate that one function will be called often, so we specialize virtual function to have a branch with direct call
- ▶ By doing it we can then inline speculated call
- ▶ We can either compare value of vptr, or the value of the function
- ▶ Interesting problem: if the speculated function was not tasty enough for the inliner, then we should do ~ ~virtualization by removing the branch

INDIRECT CALL PROMOTION

- ▶ Using PGO data we can make something similar to speculative devirtualization, but much more accurate
- ▶ It works on any indirect call
- ▶ It is implemented on LLVM level
- ▶ It will have to work well with speculative devirtualization

WHOLE PROGRAM OPTIMIZATION WITH LTO

- ▶ Knowledge about **full*** inheritance graph is the key
 - ▶ dynamic library can inherit from our types, which makes it impossible to know the full inheritance graph
- ▶ Other analysis and code specialization
- ▶ Some virtual functions can be optimized to loading bit mask/integer value from extended vtable

```
struct Base {  
    enum Type { kBase, kDerived };  
    virtual bool isDerived() const { return false; } // or:  
    virtual bool isOfType(Type t) const { return t == kBase; }  
};
```

THINLTO

- ▶ ThinLTO could solve many of the problems faced with `-fstrict-vtable-pointers`
- ▶ It is capable of doing fancy LTO tricks
- ▶ Right now ThinLTO doesn't import global objects, which limits devirtualization

DEVIRTUALIZATION CREDITS

- ▶ Anders Carlsson, David Majnemer, John McCall, Mehdi Amini, Nick Lewycky, Pete Cooper, Peter Collingbourne, Reid Kleckner, Richard Smith, Rong Xu, Sunil Srivastava, ...

QUESTIONS?

REFS

- ▶ <http://lists.llvm.org/pipermail/llvm-dev/2016-January/094600.html>
- ▶ <http://hubicka.blogspot.com/2014/01/devirtualization-in-c-part-1.html>