Fuchsia

Relative VTables in C++

Leonard Chan (*leonardchan@google.com*)

Agenda

01. What are VTables?
02. What are Relative VTables?
03. Benefits, Drawbacks, and Impact
04. Work put in
05. Future Improvements

01

What are VTables?

A crash course

What are VTables?

VTables (or virtual tables) are arrays of virtual functions.

Virtual functions are member functions of a C++ class that can be redefined in a child class.

These are used to implement runtime polymorphism in C++ through dynamic dispatching.

VTable Layout (under the Itanium C++ ABI)

```
// C++
class A {
 public:
  virtual void foo();
  virtual void bar();
};

class B : public A {
 public:
  void bar() override;
};

void func(A *a) {
  a->bar();
}
```

VTable for A		
Component	**Type**	**Value**
Offset to top	ptrdiff_t	0
Run-Time Type Information (RTTI)	64-bit pointer (to struct)	`nullptr` (with `-fno-rtti`)
Virtual function **foo**	64-bit pointer (to function)	**A::foo()**
Virtual function **bar**	64-bit pointer (to function)	**A::bar()**

VTable Layout (in ELF binary format)

```
// C++
class A {
 public:
  virtual void foo();
  virtual void bar();
};

class B : public A {
 public:
  void bar() override;
};

void func(A *a) {
  a->bar();
}
```

Located in a modifiable section (*.data.rel.ro*)

Because these symbols can be anywhere (and require **dynamic relocations**)

```
// Aarch64 (simplified)
.section      .data.rel.ro._ZTV1A,"aw"
_ZTV1A:   // vtable for A
        .xword 0                  // offset to top
        .xword 0                  // RTTI
        .xword _ZN1A3fooEv        // A::foo()
        .xword _ZN1A3barEv        // A::bar()

.section      .data.rel.ro._ZTV1B,"aw"
_ZTV1B:   // vtable for B
        .xword 0
        .xword 0
        .xword _ZN1A3fooEv        // A::foo()
        .xword _ZN1B3barEv        // B::bar()

_Z4funcP1A:   // func
        ldr    x8, [x0]           // Load the vtable
        ldr    x1, [x8, #8]       // Get the virtual function
        br     x1                 // Call the function
```

Dynamic Relocations and Position-Independent Code (PIC)

In ELF, symbols can be loaded anywhere in **PIC** binaries, so references to symbols are unknown until loaded.

A **relocation** is the process of resolving these references. **Dynamic relocations** are resolved by the dynamic linker after loading a binary.

binary.so	(???)
A::foo()	(???)
A::bar()	(???)
B::bar()	(???)
A_vtable - A::foo() - A::bar()	(???) (???) (???)
B_vtable - A::foo() - B::bar()	(???) (???) (???)

Dynamic Relocations and Position-Independent Code (PIC)

In ELF, symbols can be loaded anywhere in **PIC** binaries, so references to symbols are unknown until loaded.

A **relocation** is the process of resolving these references. **Dynamic relocations** are resolved by the dynamic linker after loading a binary.

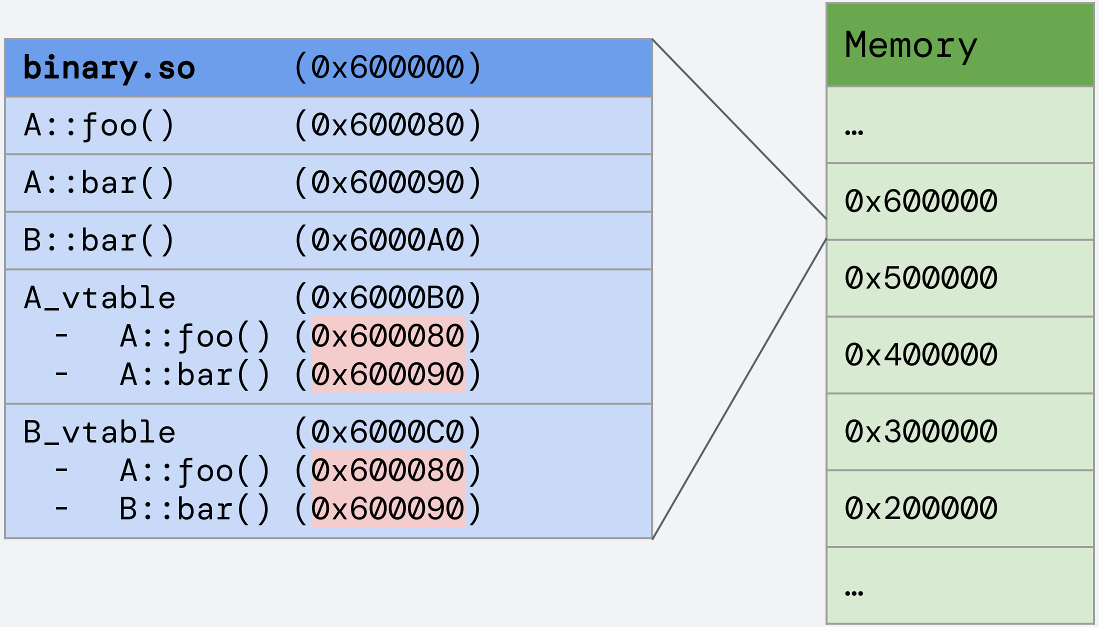
binary.so	(0x300000)
A::foo()	(0x300080)
A::bar()	(0x300090)
B::bar()	(0x3000A0)
A_vtable - A::foo() - A::bar()	(0x3000B0) (0x300080) (0x300090)
B_vtable - A::foo() - B::bar()	(0x3000C0) (0x300080) (0x300090)

Memory
…
0x600000
0x500000
0x400000
0x300000
0x200000
…

Dynamic Relocations and Position-Independent Code (PIC)

In ELF, symbols can be loaded anywhere in **PIC** binaries, so references to symbols are unknown until loaded.

A **relocation** is the process of resolving these references. **Dynamic relocations** are resolved by the dynamic linker after loading a binary.



VTables must be Writable (*at dynamic link time*)

So that the dynamic relocations can be patched.

Data in a writable sections are mapped to copy-on-write (COW) pages.

A COW page is shared between multiple processes until it is written to.
Then that page is cloned for that process.

If a binary is shared between N processes, then there could be up to N copies of a single COW page in memory.

Problem Statement

VTables are *not* PIC-friendly

For binaries that: are **PIC,** and use **Itanium C++ ABI**.

VTables contribute to the number of COW pages and **can use a lot of memory**.

In Fuchsia (at the time), ~30 MB of memory goes into modifiable data segments, a sizeable portion of which was from vtables.

How can we address this?

Ø2

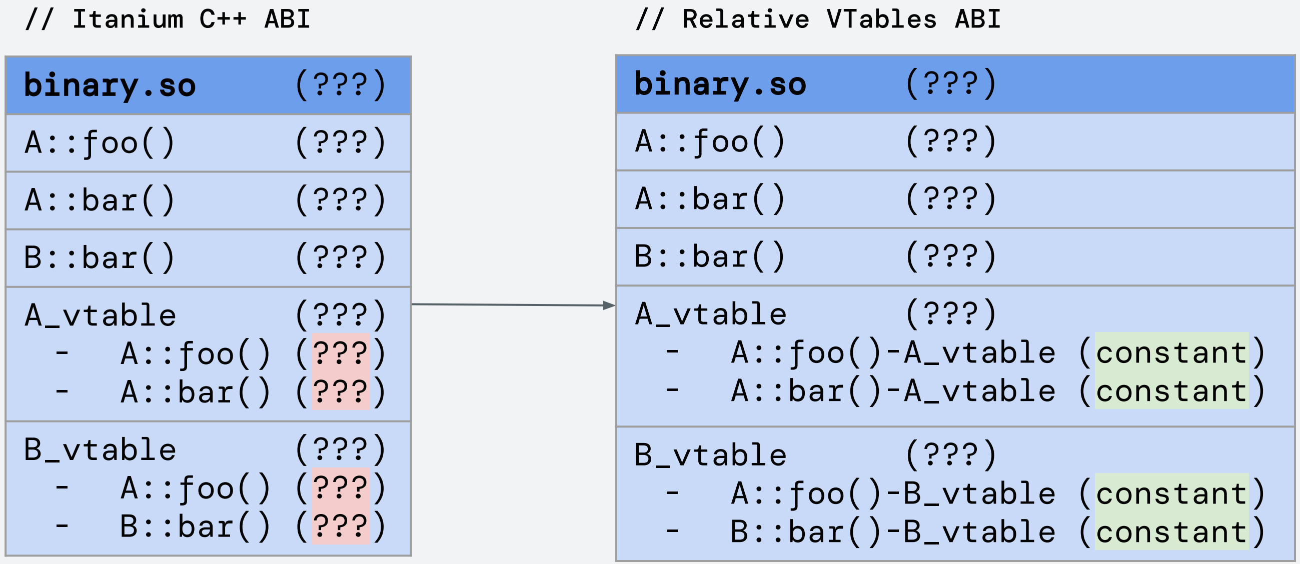
Relative VTables

Making vtables PIC-friendly

The Relative VTables C++ ABI

A space efficient ABI proposed by Peter Collingbourne that uses a **PIC-friendly encoding** of vtables.

Virtual function pointers are replaced with PC-relative offsets, which changes the dynamic relocations to static relocations.



Dynamic -> Static Relocations

Symbols within the same binary are a **constant offset** from each other.

// Itanium C++ ABI

binary.so	(???)
A::foo()	(???)
A::bar()	(???)
B::bar()	(???)
A_vtable - A::foo() - A::bar()	(???) (???) (???)
B_vtable - A::foo() - B::bar()	(???) (???) (???)

these are the same

// Itanium C++ ABI

binary.so	(addr)
A::foo()	(addr + a)
A::bar()	(addr + b)
B::bar()	(addr + c)
A_vtable - A::foo() - A::bar()	(addr + d) (addr + a) (addr + b)
B_vtable - A::foo() - B::bar()	(addr + e) (addr + a) (addr + c)

Dynamic -> Static Relocations

Symbols within the same binary are a **constant offset** from each other.

These change the dynamic relocations to **static relocations**, which are resolved at link time when building.

// Itanium C++ ABI

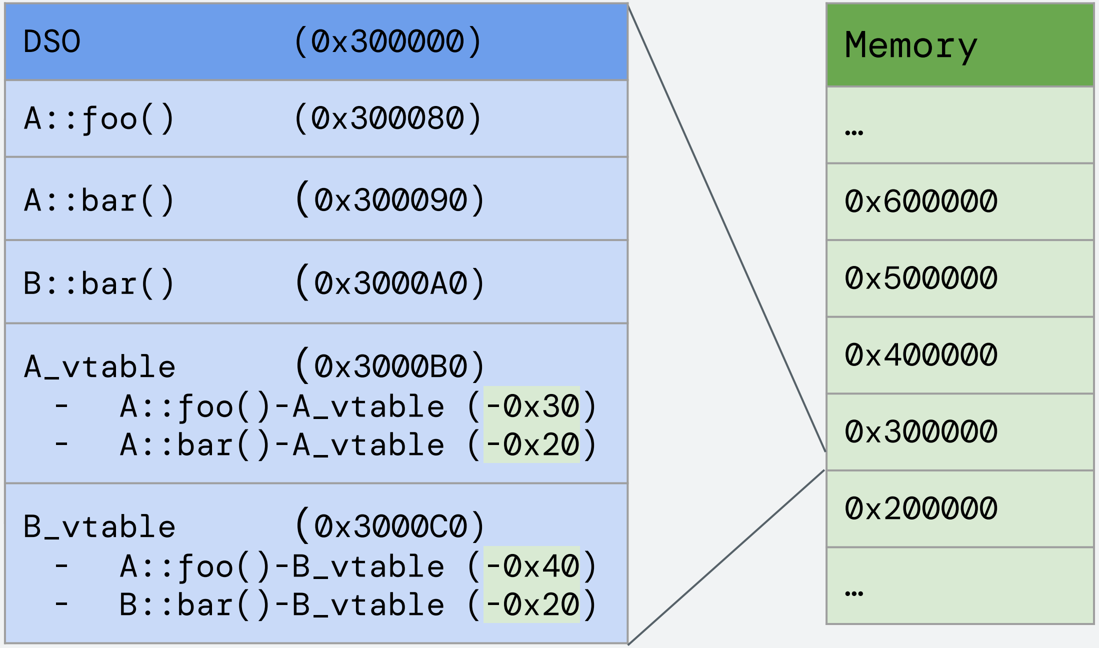
binary.so	(addr)
A::foo()	(addr + a)
A::bar()	(addr + b)
B::bar()	(addr + c)
A_vtable - A::foo() - A::bar()	(addr + d) (addr + a) (addr + b)
B_vtable - A::foo() - B::bar()	(addr + e) (addr + a) (addr + c)

// Relative VTables ABI

binary.so	(addr)
A::foo()	(addr + a)
A::bar()	(addr + b)
B::bar()	(addr + c)
A_vtable - A::foo()-A_vtable - A::bar()-A_vtable	(addr + d) (a - d) (b - d)
B_vtable - A::foo()-B_vtable - B::bar()-B_vtable	(addr + e) (a - e) (c - e)

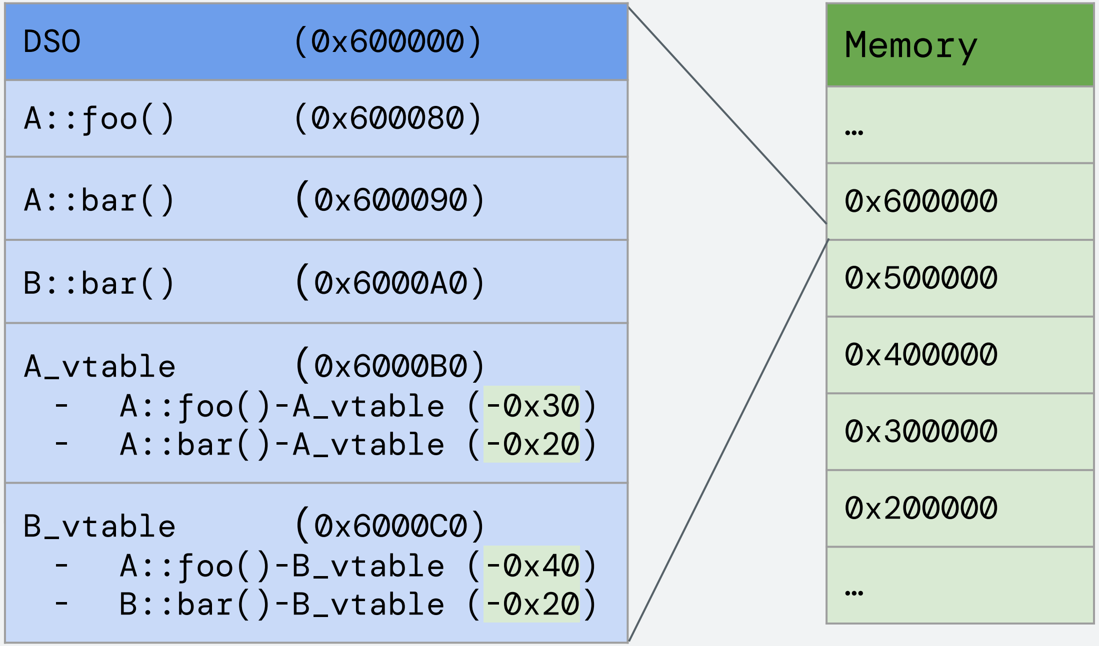
Static Relocations and PIC

Offsets within the same DSO can be computed statically, so they will stay the same value regardless of where the DSO is loaded.



Static Relocations and PIC

Offsets within the same DSO can be computed statically, so they will stay the same value regardless of where the DSO is loaded.



Updated VTable Layout

In the small memory model, all binaries are assumed to be at most 4GB in size.

For 64-bit targets using the small memory model, **offsets can also be 32 bits wide**.

	Itanium C++ ABI		**Relative VTables ABI**	
Component	**Type**	**Value**	**Type**	**Value**
Offset to top	ptrdiff_t	0	int32_t	0
Run-Time Type Information (RTTI)	64-bit pointer (to struct)	`nullptr` (with `-fno-rtti`)	int32_t	0 (with `-fno-rtti`)
Virtual function **foo**	64-bit pointer (to function)	**A::foo()**	int32_t	**A::foo() - A_vtable**
Virtual function **bar**	64-bit pointer (to function)	**A::bar()**	int32_t	**A::bar() - A_vtable**

```
// Aarch64 (Itanium C++ ABI)
        .section        .data.rel.ro._ZTV1A,"aw"
_ZTV1A:   // vtable for A
        .xword 0                    // offset to top
        .xword 0                    // RTTI
        .xword _ZN1A3fooEv          // A::foo()
        .xword _ZN1A3barEv          // A::bar()

        .section        .data.rel.ro._ZTV1B,"aw"
_ZTV1B:   // vtable for B
        .xword 0
        .xword 0
        .xword _ZN1A3fooEv          // A::foo()
        .xword _ZN1B3barEv          // B::bar()

_Z4funcP1A:   // func
        ldr     x8, [x0]            // Load the vtable
        ldr     x1, [x8, #8]        // Get the virtual function
        br      x1                  // Call the function
```

```
// Aarch64 (Relative VTables C++ ABI)
        .section        .rodata._ZTV1A,"a"
_ZTV1A:   // vtable for A
        .word  0                                  // Offset to top
        .word  0                                  // RTTI
        .word  _ZN1A3fooEv@PLT-(_ZTV1A+8)         // A::foo()-A_vtable
        .word  _ZN1A3barEv@PLT-(_ZTV1A+8)         // A::bar()-A_vtable

        .section        .rodata._ZTV1B,"a"
_ZTV1B:   // vtable for B
        .word  0
        .word  0
        .word  _ZN1A3fooEv@PLT-(_ZTV1B+8)         // A::foo()-B_vtable
        .word  _ZN1B3barEv@PLT-(_ZTV1B+8)         // B::bar()-B_vtable

_Z4funcP1A:   // func
        ldr     x8, [x0]             // Load vtable
        ldrsw   x9, [x8, #4]         // Get relative offset
        add     x1, x8, x9           // Add the offset
        br      x1                   // Call
```

PIC-friendly Encodings

Avoid referencing **addresses** and use constant **integers** wherever possible (*dynamic vs static relocations*).

Take advantage of PC-relative offsets.

Clang already uses this for unwind info (`.eh_frame`), table lookup optimizations [1], and profile formatting.

Swift already uses this [2].

[1] Gulfem Savrun Yeniceri
[2] J. Groff & D. Gregor

Ø3

Benefits, Drawbacks, and Impact

Benefits: NO dynamic relocations in vtables

```
        .section     .rodata._ZTV1A
_ZTV1A:  // vtable for A
        .word  0                        // Offset to top
        .word  0                        // RTTI
        .word  _ZN1A3fooEv@PLT-(_ZTV1A+8)
        .word  _ZN1A3barEv@PLT-(_ZTV1A+8)

        .section        .rodata._ZTV1B
_ZTV1B:  // vtable for B
        .word  0
        .word  0
        .word  _ZN1A3fooEv@PLT-(_ZTV1B+8)
        .word  _ZN1B3barEv@PLT-(_ZTV1B+8)

_Z4funcP1A:  // func
        ldr     x8, [x0]           // Load vtable
        ldrsw   x9, [x8, #4]       // Get relative offset
        add     x1, x8, x9         // Add the offset
        br      x1                 // Call
```

VTables can be pure readonly and shared between processes.

VTables have no dynamic relocations.

Faster startup time.

Lower memory impact (fewer COW pages).

Benefits: VTables sizes are halved (*for 64-bit platforms*)

```
        .section      .rodata._ZTV1A
_ZTV1A:   // vtable for A
        .word  0                          // Offset to top
        .word  0                          // RTTI
        .word  _ZN1A3fooEv@PLT-(_ZTV1A+8)
        .word  _ZN1A3barEv@PLT-(_ZTV1A+8)

        .section         .rodata._ZTV1B
_ZTV1B:   // vtable for B
        .word   0
        .word   0
        .word   _ZN1A3fooEv@PLT-(_ZTV1B+8)
        .word   _ZN1B3barEv@PLT-(_ZTV1B+8)

_Z4funcP1A:   // func
        ldr      x8, [x0]          // Load vtable
        ldrsw    x9, [x8, #4]      // Get relative offset
        add      x1, x8, x9        // Add the offset
        br       x1                // Call
```

Binary size decrease.*

Lower memory impact (smaller data objects)

Drawbacks: More instructions

Extra instructions at each call site for adding the offset (+1 on AArch64, +3 on x86_64)

.text increase can counter data decrease

Itanium C++ ABI (AArch64)	Relative VTables C++ ABI (AArch64)

`ldr x8, [x0]      (1) Load vtable`
`ldr x1, [x8, #8]  (2) Load vfunc`
`br  x1            (3) Call vfunc`
	`ldr   x8, [x0]      (1) Load vtable`
	`ldrsw x9, [x8, #4]  (2) Load 32-bit offset`
	`add   x1, x8, x9    (3) Add offset to vtable`
	`br    x1            (4) Call vfunc`

Itanium C++ ABI (x86_64)	**Relative VTables C++ ABI (x86_64)**

`movq   (%rdi), %rax    (1) Load vtable`
`callq  *0x10(%rax)     (2) Load and call vfunc`
	`movq   (%rdi), %rcx    (1) Load vtable`
	`mov    %rcx,%rax       (2) Save vtable into rax`
	`movslq 0x8(%rcx),%rcx  (3) Load 32-bit offset`
	`add    %rcx,%rax       (4) Add offset to vtable`
	`callq  *%rax           (5) Call vfunc`

TODO: Perhaps this could be **call** *(%rax,%rcx)

Drawbacks: *Compressed* Binary Size Regressions

Chromium on Fuchsia saw ~1 % size increase (~390 KB) in the *compressed* binary size.

LIkely because vtables *before* were filled with **zeroes**, but are now filled with **random integers** (offsets).

Zeroes likely compress better than pseudo-random integers.

Drawbacks: ABI Change!

Binaries that expose the C++ ABI (or specifically vtables) will not work correctly unless all binaries involved use the same vtable layout.

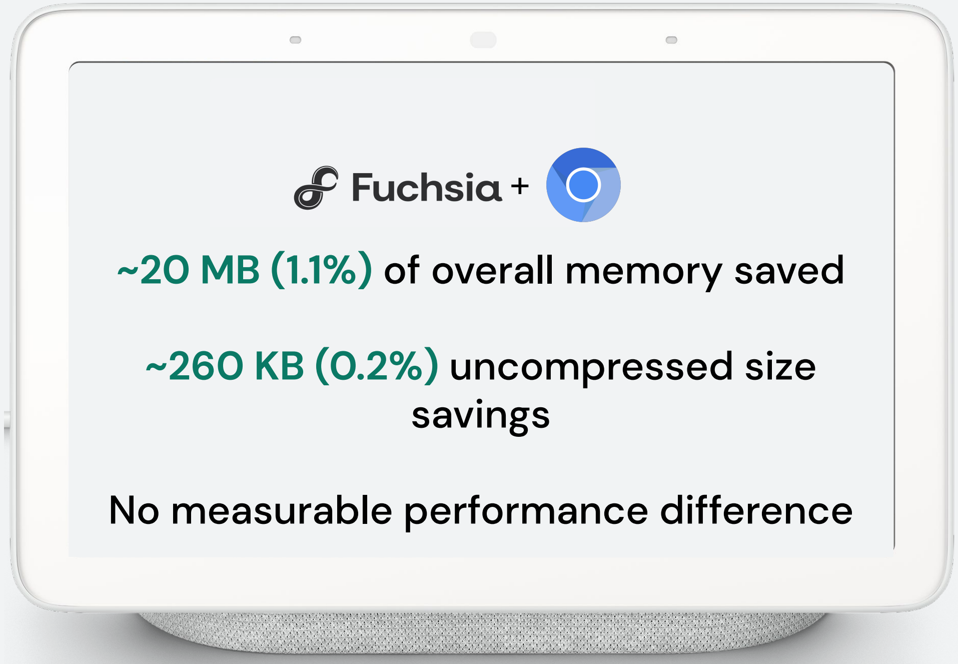
For example, a relative vtables (RV) binary using *libc++* will need a *libc++* compiled with RV. **BUT** a RV binary using sanitizers doesn't need RV-compliant *compiler runtimes* because they do **NOT** expose vtables.

Drawbacks: ABI Change! (but ok for Fuchsia 👍)

In Fuchsia, all binaries can use RV by default because **we do not depend on the C++ ABI** and are free to change it.

Fuchsia operates on a “Bring Your Own Runtime” model, which means user applications can bring their own libraries compiled with whatever ABI they would like (similar to Flatpak).

There is no “system” libc++(abi) that user programs depend on.


Fuchsia +
~20 MB (1.1%) of overall memory saved
~260 KB (0.2%) uncompressed size savings
No measurable performance difference

Ø4

Work Effort

Or “Lessons Learned”

A New Static Relocation: *R_AARCH64_PLT32*

Prior to D77647, there was no way of generating DSO-local a veneer (PLT entry) for functions.

We wanted something similar to X86's **R_X86_64_PLT32**.

This generates a PLT entry and can be statically computed at link time.

A New IR Construct: `dso_local_equivalent` `@func`

A new LLVM IR construct for indicating that the function passed to it will be resolved to a function within the same linkage unit.

Needed a way in IR to semantically represent that a specific reference to a function should be lowered to a PLT entry.

Slightly different from `dso_local` which is attached to function declarations.

PC-Relative RTTI Offsets

```
// Aarch64 (Itanium C++ ABI)
        .section      .data.rel.ro._ZTV1A
_ZTV1A:   // vtable for A
        .xword 0                  // offset to top
        .xword _ZTI1A             // RTTI
        .xword _ZN1A3fooEv        // A::foo()
        .xword _ZN1A3barEv        // A::bar()

       .section      .data.rel.ro._ZTI1A
_ZTI1A:   // typeinfo for A
       // vtable for  __cxxabiv1::__class_type_info
       .xword  _ZTVN10__cxxabiv117__class_type_infoE+16
       .xword  _ZTS1A  // typeinfo name
```

```
// Aarch64 (Relative VTables C++ ABI)
        .section      .rodata._ZTV1A
_ZTV1A:   // vtable for A
        .word  0                            // Offset to top
        .word  _ZTI1A.rtti_proxy-(_ZTV1A+8) // A_RTTI-A_vtable
        .word  _ZN1A3fooEv@PLT-(_ZTV1A+8)   // A::foo()-A_vtable
        .word  _ZN1A3barEv@PLT-(_ZTV1A+8)   // A::bar()-A_vtable

        .hidden       _ZTI1A.rtti_proxy
        .section      .data.rel.ro._ZTI1A.rtti_proxy
_ZTI1A.rtti_proxy:      // typeinfo for A (rtti_proxy)
        .xword  _ZTI1A  // typeinfo for A

        .section      .data.rel.ro._ZTI1A
_ZTI1A:  // typeinfo for A
        .xword  _ZTVN10__cxxabiv117__class_type_infoE+8
        .xword  _ZTS1A      // typeinfo name
```

RTTI change requires libc++abi change

`__dynamic_cast` needs to account for the extra arithmetic for the offset calculation.

RV with libc++abi requires at least revision [61aec69a65dec949f3d2556c4d0efaa87869e1ee].

This is only required change outside of Clang/LLVM.

```
#if __has_feature(cxx_abi_relative_vtable)
    // The vtable address will point to the first virtual function, which is 8
    // bytes after the start of the vtable (4 for the offset from top + 4 for the typeinfo component).
    const int32_t* vtable =
        *reinterpret_cast<const int32_t* const*>(static_ptr);
    int32_t offset_to_derived = vtable[-2];
    const void* dynamic_ptr = static_cast<const char*>(static_ptr) + offset_to_derived;

    // The typeinfo component is now a relative offset to a proxy.
    int32_t offset_to_ti_proxy = vtable[-1];
    const uint8_t* ptr_to_ti_proxy =
        reinterpret_cast<const uint8_t*>(vtable) + offset_to_ti_proxy;
    const __class_type_info* dynamic_type =
        *(reinterpret_cast<const __class_type_info* const*>(ptr_to_ti_proxy));
#else
    void **vtable = *static_cast<void ** const *>(static_ptr);
    ptrdiff_t offset_to_derived = reinterpret_cast<ptrdiff_t>(vtable[-2]);
    const void* dynamic_ptr = static_cast<const char*>(static_ptr) + offset_to_derived;
    const __class_type_info* dynamic_type = static_cast<const __class_type_info*>(vtable[-1]);
#endif
```

Ø5

Future Improvements

Whole Program Devirtualization (WPD)

WPD attempts to replace loading and indexing into the vtable for a virtual function with calling the virtual function directly.

The WPD pass searches for these loads by finding loads/GEPs that accept virtual pointers.

This will not find instances of RV loads, which use a special intrinsic called `llvm.load.relative()`.

Optimizations tend to optimize for IR patterns around the Itanium C++ ABI. In general, it's difficult to catch regressions to optimizations with respect to ABI changes.

Use the GOT instead of `.rtti_proxy`

The `.rtti_proxy`s functionally serve the same purpose as the Global Offset Table.

Both act as DSO-local addresses that contain references to other addresses.

We should use an existing linker-generated data structure than a custom one.

The symbol table would be less polluted with `.rtti_proxy` symbols.

Compatibility with HWASan on Globals

HWASan works on globals by inserting a tag into the top byte of an ***IR*** global.

Relative vtables work by taking the offset between two globals.

If the top-byte on a vtable is non-zero, then the result for the offset calculation may not fit in 32 bits and result in this error:

```
>>> ld.lld: error: <stdin>:(.rodata..Lrodata_obj.hwasan+0x0): relocation
R_AARCH64_PREL32 out of range: -72057594037730896 is not in [-2147483648,
4294967295]; references hidden defined in /tmp/test.o
```

Extending Support for Other Platforms

Currently only supported for **64-bit ELF** binaries on **AArch64** and **X86_64**.

Other architectures/binary formats will need to support 32-bit PC-relative relocations (similar to R_AARCH64_PLT32).

`dso_local_equivalent` is currently only lowered on ELF platforms.

Raising PIC-friendly Awareness

More memory savings can be achieved by moving more “read-only” data structures PC-relative.

Table lookup optimizations now use relative offsets in PIC-mode.

Profile formatting is now PIC-friendly.

The RTTI struct can be PIC-friendly.

Can this be extended to other languages like Rust or Go? (This is already used in Swift).

Introduce C/C++ attributes that allow for making user structs/classes “relative”?

Thank you!

`-fexperimental-relative-c++-abi-vtables`

Thanks also for the code reviews:

Peter Collingbourne, John McCall, Petr Hosek, Roland McGrath,
Jake Ehrlich, Peter Smith, Fangrui Song