WHAT ARE VIRTUAL CALLS

- Polymorphism in OOP

%vtable = load {...} %p

%vfun = load {...} %vtable

call {...} %vfun(args...)
DEVIRTUALIZATION

- Optimization changing virtual (indirect) calls to direct calls
- Important for performance:
  - more inlining
  - indirect calls are harder to predict
    - Spectre/Meltdown mitigation
- Security implications
# Devirtualization by Frontend

```c
struct A {
    virtual void virt_meth();
};

void bar() {
    A a;
    a.virt_meth(); // devirtualized by the frontend
}
```
void foo(A *a) {
    a->virt_meth();
}

void bar() {
    A a;
    // will be devirtualized after inlining
    foo(&a);
}
**PROBLEM WITH EXTERNAL FUNCTIONS**

```c
void foo(A *a) {
    a->virt_meth();
}
void external_fun(A *a);
void bar() {
    A a;
    // assumes external_fun may clobber a’s vptr
    external_fun(&a);
    foo(&a);
}
```
IT GETS EVEN WORSE

```c
void bar() {
    auto a = new A;
    a->virt_meth();
    // can devirtualize only if the first call was
    // inlined
    a->virt_meth();
}
```
MARK VPTR AS INVARIANT

- `!invariant.load` would be sufficient for Java
- C++’s ctors/dtors/placement new/... require more tricks

```c
void foo() {
    A *a = new A;
    A *b = new(a) B;
    b->virt_meth();
    a->virt_meth(); // undefined behavior
}
```
**MARK VPTR AS INVARIANT**

```cpp
void A::virt_meth() {
  static_assert(sizeof(A) == sizeof(B));
  new(this) B;
}

auto *a = new A;
a->virt_meth();
a->virt_meth(); // Undefined behavior
```
OLD MODEL

- `!invariant.group` – delimitable `!invariant.load`
- `llvm.invariant.group.barrier` intrinsic
  - needs to be used whenever the dynamic type changes
  - stops `!invariant.group` optimizations
  - returns a new SSA value
OLD MODEL'S FLAW

```c
void g() {
    A *a = new A;
    a->virt_meth();
    A *b = new(a) B;
    if (a == b)
        b->virt_meth();
}
```

- we could add barriers to the compared pointers
  barrier(a) == barrier(b)
NEW MODEL
NEW MODEL

- Think of pointers to dynamic objects as fat pointers
- Equip each pointer with optional virtual metadata (pun intended)
- Each !invariant.group load/store must read/write the value associated with the virtual metadata
- Comparison of objects’ addresses must be done on raw pointers
**LAUNDER.INVARIANT.GROUP INTRINSIC**

- Creates a fat pointer with fresh virtual metadata
- Used: whenever the dynamic type could change
  - derived ctor/dtor
  - placement new
  - int to ptr
  - union members
  - std::launder
Strips virtual metadata

Pure (readnone) function

Used: when we stop caring about the dynamic type
  - ptr to int
  - pointer comparisons

Can be safely replaced with launder
INTRINSICS’ PROPERTIES

- %b = launder(%a)
  %c = launder(%b)
  ; => %c = launder(%a)

- %b = launder(%a)
  %c = strip(%b)
  ; => %c = strip(%a)

- %b = strip(%a)
  %c = launder(%b)
  ; => %c = launder(%a)

- %b = strip(%a)
  %c = strip(%b)
  ; => %c = strip(%a) => %b

- Returned pointer aliases the argument

- Both intrinsics can be removed if the result is unused
Can propagate equality

```cpp
auto *a = new A;
auto *b = new (a) B;
a == a
a == b;
std::launder(a) == b;
```

Even when the dynamic type changes

```cpp
auto *a = new A;
*a1 = strip(*a)
*a2 = strip(*a)
; optimizes to true
%b = icmp eq %a1, %a2
```
EXPERIMENTAL RESULTS
## Optimizations Statistics (Old Model)

### Results for LLVM

<table>
<thead>
<tr>
<th>statistic</th>
<th>baseline</th>
<th>devirt</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td># of vtable loads replaced</td>
<td>1451</td>
<td>14254</td>
<td>882.36%</td>
</tr>
<tr>
<td># of vtable uses devirtualized</td>
<td>982</td>
<td>3269</td>
<td>232.89%</td>
</tr>
<tr>
<td># of vfunction loads replaced</td>
<td>1084</td>
<td>9388</td>
<td>766.05%</td>
</tr>
<tr>
<td># of vfunction devirtualized</td>
<td>954</td>
<td>1861</td>
<td>95.07%</td>
</tr>
</tbody>
</table>

### Results for ChakraCore

<table>
<thead>
<tr>
<th>statistic</th>
<th>baseline</th>
<th>devirt</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td># of vtable loads replaced</td>
<td>126</td>
<td>2465</td>
<td>1856.35%</td>
</tr>
<tr>
<td># of vtable uses devirtualized</td>
<td>17</td>
<td>584</td>
<td>3335.29%</td>
</tr>
<tr>
<td># of vfunction loads replaced</td>
<td>45</td>
<td>1082</td>
<td>2304.44%</td>
</tr>
<tr>
<td># of vfunction devirtualized</td>
<td>32</td>
<td>131</td>
<td>309.38%</td>
</tr>
</tbody>
</table>
BENCHMARKS

Distribution of metric averages in exp. run, relative to baseline (geomean 1.008)
BENCHMARKS
OTHER BENCHMARKS

- Google Search benchmarks showed 0.65% improvement (without FDO)
- Spec2006 didn’t show any difference
- 7zip and zippy benchmarks showed 0.6% improvement before fixing the inliner
  - after fixing the inliner, there was no change for 7zip and zippy regressed
  - requires further investigation
WHEN ARE WE GETTING DEVIRTUALIZATION?

- We need a way to perform safe optimizations between modules compiled with and without devirtualization
  - RFC soon
- We hope the next release will have it turned on by default :)

LLVM DEVMTG’18 — SOUND DEVIRTUALIZATION
FURTHER WORK

- Clang’s new experimental flag `-fforce-emit-vtables`
- Calling one virtual function from another will not be devirtualized unless the latter is inlined or final
  - Emit a called-through-vtable specialization of every method (possibly duplicating it for derived types)
  - Perform explicit direct calls (`a->A::virt_meth()`) to virtual methods in the usual way