Speculative Compilation in ORC

ORC

- LLVM Modular Just in Time Compilation Library
- Custom compilers, program representations...
- Supports concurrent compilation

JIT Variants

- Eager JIT - high startup time, zero compiler interactions at runtime

- Lazy   JIT - ⚡ startup time, compilation overhead on first call

Can we do better? Can we have benefits of two worlds? 🤔

Let's guess it!

```
void Driving(Signal S)

        switch(S){

        case red:stop();tweet();

        break;

        case yellow:

        like_reply_to_a_tweet();break;

        case green :think_next_tweet();

        break;

}}
```

What if we guess the signal's outcome and do action!

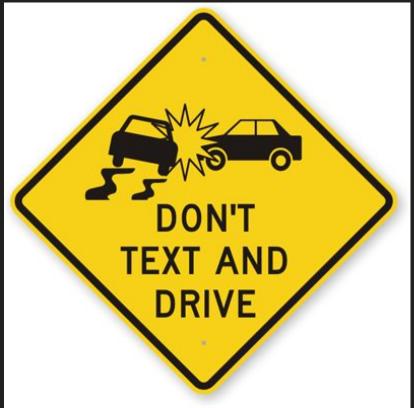
Likewise, we guess control flow path and compile the *likely* functions before calling them.

Let’s guess it!

```
void Driving(Signal S)
        switch(S){
        case red:stop();tweet();
        break;
        case yellow:
        like_reply_to_a_tweet();break;
        case green :think_next_tweet();
        break;
}}
```

What if we guess the signal’s outcome and do action!

Likewise, we guess control flow path and compile the *likely* functions before calling them.



Speculative Decisions

- Compile only the most likely next executable functions
- Speculate based on CFG edge probabilities and hot blocks heuristics
- Implemented as SpeculateQuery function objects, you can try your own ideas 😊
    - Map<Function, LikelyFunctionSymbols> SpeculateQuery(Function& F);

➥ “Jump into JIT through IR Instrumentation”

Mix‘n’Match

```
ObjectLinkingLayer LinkLayer;

IRCompileLayer<...> CompileLayer(LinkLayer, ConcurrentCompiler);

IRSpeculationLayer SpeculateLayer(...,CompileLayer,Speculator,SpeculateQuery)

CompileOnDemandLayer<...> CODLayer(SpeculateLayer, ...);

CODLayer.addModule(Mod, MemMgr, SymResolver);

auto FooSym = CODLayer.findSymbol(“foo”, true);

auto Foo = reinterpret_cast<int(*)()>(FooSym.getAddress());

int Result = Foo(); // <-- Call foo’s stub.
```

heuristics

Before

```
define dso_local void @Driving(i32 %Signal)
  switch i32 %Signal,label %exit[i32 0,label %red
  i32 1, label %yellow i32 2, label %green]
red:
  call void @stop()  call void @tweet()
yellow:
  call void @like_reply_to_a_tweet()
green:
  call void @think_next_tweet()
```

After...

```
@__orc_speculator = external global %Class.Speculator

declare void @__orc_speculate_for(%Class.Speculator* %0, i64 %1)

define dso_local void @Driving(i32 %0) #0 {

  call void @__orc_speculate_for(%Class.Speculator* @__orc_speculator,

  i64 ptrtoint (i32 ()* @Driving to i64)) // Jump into JIT 👈

 ...

  switch i32 %3, label %7 [ i32 0, label %Red

  i32 1, label %Yellow

  i32 2, label %Green]

...
```

Performance

7× Speed-Down 💔💔

Multiple Jumps into JIT ❌

ExecutionSession::Lookup’s are not free✅

We want Performance👊

- Relatively easy fix, guard the orc_speculate_for call
- Jump into JIT only on the first call
- This will give us - what we want

```
@__orc_speculate.guard.for.main = internal local_unnamed_addr global i8 0, align 1

define dso_local void @Driving(i32 %0) {

__orc_speculate.decision.block:

  %guard.value = load i8, i8* @__orc_speculate.guard.for.main

  %compare.to.speculate = icmp eq i8 %guard.value, 0

  br i1 %compare.to.speculate, label %__orc_speculate.block, label %program.entry

__orc_speculate.block:

  call void @__orc_speculate_for(%Class.Speculator* @__orc_speculator,

  i64 ptrtoint (i32 ()* @Driving to i64))

  store i8 1, i8* @__orc_speculate.guard.for.main

  br label %program.entry
```

Performance

We see significant speedup with our proof-of-concept speculative jit

For SPEC 403.gcc benchmark, reduce exec time from 17.4 seconds to 10.5 seconds (4 threads) 😃

What’s Next?

- Finish dynamic profiling support to collect branch probability information
- Reduce the scope of speculation region in a function
- Implementing more SpeculateQueries
- Performance tuning

Thank you so much - Lang Hames and David Blaikie

Thank you LLVM Foundation and Conf Sponsors.