Automatic Program Reoptimization Support in LLVM ORC JIT

by Sunho Kim
ABOUT ME
ABOUT ME

- Undergrad student from UC San Diego
ABOUT ME

- Undergrad student from UC San Diego
- Worked on just-in-time linker of LLVM through Google Summer of Code 2022
ABOUT ME

- Undergrad student from UC San Diego
- Worked on just-in-time linker of LLVM through Google Summer of Code 2022
- Worked on re-optimization feature through Google Summer of Code 2023 under guidance of Lang Hames and Vassil Vassilev
ABOUT ME

- Undergrad student from UC San Diego
- Worked on just-in-time linker of LLVM through Google Summer of Code 2022
- Worked on re-optimization feature through Google Summer of Code 2023 under guidance of Lang Hames and Vassil Vassilev
  - Which is what this talk will be about
MOTIVATION
MOTIVATION

- Compile with -O2 for only “hot” functions
  - The compilation time of -O0 or -O1 is faster than -O2 in general
MOTIVATION

- Compile with -O2 for only “hot” functions
  - The compilation time of -O0 or -O1 is faster than -O2 in general
- Runtime profile guided optimization
  - De-virtualization, instruction reordering, and other types of PGOs in ORC JIT
MOTIVATION

- Compile with -O2 for only “hot” functions
  - The compilation time of -O0 or -O1 is faster than -O2 in general
- Runtime profile guided optimization
  - De-virtualization, instruction reordering, and other types of PGOs in ORC JIT
- Scientific computing (CERN)
  - Use high precision floating point for early iterations and use low precision floating point in later iterations for places that matter
REVIVING FEATURE FROM 2003?
REVIVING FEATURE FROM 2003?

We use opt to do bytecode-to-bytecode instrumentation. Look at back-edges and insert `llvm_first_trigger()` function call which takes no arguments and no return value. This instrumentation is designed to be easy to remove, for instance by writing a NOP over the function call instruction.

Keep count of every call to `llvm_first_trigger()`, and maintain counters in a map indexed by return address. If the trigger count exceeds a threshold, we identify a hot loop and perform second-level instrumentation on the hot loop region (the instructions between the target of the back-edge and the branch that causes the back-edge). We do not move code across basic-block boundaries.

Second-level instrumentation

We remove the first-level instrumentation by overwriting the CALL to `llvm_first_trigger()` with a NOP.
REVIVING FEATURE FROM 2003?

Quite different but has the same name :)

Code

```plaintext

First-level instrumentation

We use opt to do bytecode-to-bytecode instrumentation. Look at back-edges and insert llvm_first_trigger() function call which takes no arguments and no return value. This instrumentation is designed to be easy to remove, for instance by writing a NOP over the function call instruction.

Keep count of every call to llvm_first_trigger(), and maintain counters in a map indexed by return address. If the trigger count exceeds a threshold, we identify a hot loop and perform second-level instrumentation on the hot loop region (the instructions between the target of the back-edge and the branch that causes the back-edge). We do not move code across basic-block boundaries.

Second-level instrumentation

We remove the first-level instrumentation by overwriting the CALL to llvm_first_trigger() with a NOP.
```

ddunbar [typo] An LLVM.
OVREVIEW OF ORC JIT
Usual executable generation pipeline in LLVM
OVREVIEW OF ORC JIT
Usual executable generation pipeline in LLVM
OVREVIEW OF ORC JIT
Usual executable generation pipeline in LLVM
OVERVIEW OF ORC JIT

Usual executable generation pipeline in LLVM
OVREVIEW OF ORC JIT

Usual executable generation pipeline in LLVM
OVREVIEW OF ORC JIT

JIT execution pipeline in LLVM

Frontend → Backend → JIT Linker

Object files (in memory)
Overview of ORC JIT

JIT execution pipeline in LLVM

- Share a huge portion of pipeline with AOT
- Fewer breakage by LLVM internal code changes
OVERVIEW OF ORC JIT
OVREVIEW OF ORC JIT

- Lazy JIT support
  - Frontend AST or IR module will start compiling when a function defined by it is called in runtime.
OVREVIEW OF ORC JIT

- Lazy JIT support
  - Frontend AST or IR module will start compiling when a function defined by it is called in runtime.
- Supports all major object file format and architecture natively.
  - ELF, COFF, MACHO, ARM64, PPC, RISC-V...
OVREVIEW OF ORC JIT

- Lazy JIT support
  - Frontend AST or IR module will start compiling when a function defined by it is called in runtime.
- Supports all major object file format and architecture natively.
  - ELF, COFF, MACHO, ARM64, PPC, RISC-V...
  - In most of the cases, there's no limitation on which object file features can be used in JIT. (e.g. one can use MSVC SEH exception on COFF)
OVREVIEW OF ORC JIT

• Lazy JIT support
  ○ Frontend AST or IR module will start compiling when a function defined by it is called in runtime.

• Supports all major object file format and architecture natively.
  ○ ELF, COFF, MACHO, ARM64, PPC, RISC-V...
  ○ In most of the cases, there’s no limitation on which object file features can be used in JIT. (e.g. one can use MSVC SEH exception on COFF)

• Runtime support
  ○ Supports static initializer, thread local storage (TLS), and runtime symbol lookup ("dlload or dlsym” of JIT symbols)
OVREVIEW OF ORC JIT

- **Lazy JIT support**
  - Frontend AST or IR module will start compiling when a function defined by it is called in runtime.

- **Supports all major object file format and architecture natively.**
  - ELF, COFF, MACHO, ARM64, PPC, RISC-V...
  - In most of the cases, there’s no limitation on which object file features can be used in JIT. (e.g. one can use MSVC SEH exception on COFF)

- **Runtime support**
  - Supports static initializer, thread local storage (TLS), and runtime symbol lookup ("dlload or dlsym" of JIT symbols)

- **Multi-thread, remote process, speculative compilation** ...
WHAT’S NEW

New JIT API that does the following:
WHAT’S NEW

New JIT API that does the following:

| ORC JIT | → | IR function |
WHAT’S NEW

New JIT API that does the following:

ORC JIT → IR function → Binary code
WHAT’S NEW

New JIT API that does the following:

1. ORC JIT
2. IR function
3. Binary code

Reoptimization request
WHAT’S NEW

New JIT API that does the following:
WHAT’S NEW

New JIT API that does the following:

1. **ORC JIT** → **IR function**
2. User-defined transformation
3. Reoptimization request
4. **Reoptimized Binary Code**
BASIC USAGE OF REOPTIMIZATION API
BASIC USAGE OF REOPTIMIZATION API

- LLLayerJIT
BASIC USAGE OF REOPTIMIZATION API

- LLLayerJIT

```cpp
std::unique_ptr<LLLayerJIT> Jit;
Jit->addLayer(ReOptLayer);
Jit->addLayer(std::make_unique<LLIRPartitionLayer>());
Jit->addLayer(std::make_unique<LLCompileOnDemandLayer>());
```
BASIC USAGE OF REOPTIMIZATION API

- LLLayerJIT

```cpp
std::unique_ptr<LLLayerJIT> Jit;

Jit->addLayer(ReOptLayer); // Add re-optimization layer
Jit->addLayer(std::make_unique<LLIRPartitionLayer>());
Jit->addLayer(std::make_unique<LLCompileOnDemandLayer>());
```
BASIC USAGE OF REOPTIMIZATION API

- LLLayerJIT

```cpp
std::unique_ptr<LLLayerJIT> Jit;
Jit->addLayer(ReOptLayer);  // Add re-optimization layer
Jit->addLayer(std::make_unique<LLIRPartitionLayer>());  // Split IR module
Jit->addLayer(std::make_unique<LLCompileOnDemandLayer>());
```
BASIC USAGE OF REOPTIMIZATION API

- **LLLayerJIT**

```cpp
std::unique_ptr<LLLayerJIT> Jit;
Jit->addLayer(ReOptLayer);  // Add re-optimization layer
Jit->addLayer(std::make_unique<LLIRPartitionLayer>().get());  // Split IR module
Jit->addLayer(std::make_unique<LLCompileOnDemandLayer>().get());  // Add lazy-compilation layer
```
BASIC USAGE OF REOPTIMIZATION API
BASIC USAGE OF REOPTIMIZATION API

- ReOptimizeLayer
BASIC USAGE OF REOPTIMIZATION API

- ReOptimizeLayer
  - Insert instrumentation code and re-optimization request code.
BASIC USAGE OF REOPTIMIZATION API

- **ReOptimizeLayer**
  - Insert instrumentation code and re-optimization request code.
  - User callback ReOptimizeFunc does custom re-optimization.
BASIC USAGE OF REOPTIMIZATION API

- **ReOptimizeLayer**
  - Insert instrumentation code and re-optimization request code.
  - User callback ReOptimizeFunc does custom re-optimization.

```cpp
static Error reoptimizeBasic(ReOptimizeLayer &Parent, ReOptMaterializationUnitID MUID,
                            unsigned CurVerison, ResourceTrackerSP OldRT, ThreadSafeModule &TSM) {
    TSM.withModuleDo([&](llvm::Module &M) {
        // Do some re-optimization based on profile data
    });
    return Error::success();
}

auto ReOptLayer = std::make_unique<LLReOptimizeLayer>(ES, RSManger);
ReOptLayer->setReOptimizeFunc(reoptimizeBasic);
```
BASIC USAGE OF REOPTIMIZATION API

- **ReOptimizeLayer**
  - Insert instrumentation code and re-optimization request code.
  - User callback ReOptimizeFunc does custom re-optimization.

```cpp
static Error reoptimizeBasic(ReOptimizeLayer &Parent, ReOptMaterializationUnitID MUID,
   unsigned CurVerison, ResourceTrackerSP OldRT, ThreadSafeModule &TSM) {
    TSM.withModuleDo([&](llvm::Module &M) {
      // Do some re-optimization based on profile data
    });
    return Error::success();
}

auto ReOptLayer = std::make_unique<LLReOptimizeLayer>(ES, RSManager);
ReOptLayer->setReOptimizeFunc(reoptimizeBasic);
```
BASIC USAGE OF REOPTIMIZATION API

- **ReOptimizeLayer**
  - Insert instrumentation code and re-optimization request code.
  - User callback ReOptimizeFunc does custom re-optimization.

```cpp
static Error reoptimizeBasic(ReOptimizeLayer &Parent, ReOptMaterializationUnitID MUID,
    unsigned CurVerison, ResourceTrackerSP OldRT, ThreadSafeModule &TSM) {
    TSM.withModuleDo([&](llvm::Module &M) {
        // Do some re-optimization based on profile data
    });
    return Error::success();
}

auto ReOptLayer = std::make_unique<LLReOptimizeLayer>(ES, RSManager);
ReOptLayer->setReOptimizeFunc(reoptimizeBasic);
```
BASIC USAGE OF REOPTIMIZATION API
BASIC USAGE OF REOPTIMIZATION API

- AddProfilerFunc
BASIC USAGE OF REOPTIMIZATION API

- AddProfilerFunc
  - Called to add instrumentation code to the “first version” of the functions.
BASIC USAGE OF REOPTIMIZATION API

- AddProfilerFunc
  - Called to add instrumentation code to the “first version” of the functions.
  - Default is “reoptimizelfCallFrequent” which requests re-optimization when call count is high.
BASIC USAGE OF REOPTIMIZATION API
BASIC USAGE OF REOPTIMIZATION API

Example: do -O2 optimization if function was called more than 10

```c++
static Error reoptimizeToO2(ReOptimizeLayer &Parent, ReOptMaterializationUnitID MUID,
    unsigned CurVerison, ResourceTrackerSP OldRT, ThreadSafeModule &TSM) {
    TSM.withModuleDo([&](llvm::Module &M) {
        auto PassManager = buildPassManager();
        PassManager.run(M);
        return Error::success();
    });
    ReOptLayer->setReOptimizeFunc(reoptimizeToO2);
    ReOptLayer->setAddProfilerFunc(reoptimizeIfCallFrequent);
```
BASIC USAGE OF REOPTIMIZATION API

Example: do -O2 optimization if function was called more than 10

```cpp
static Error reoptimizeToO2(ReOptimizeLayer &Parent, ReOptMaterializationUnitID MUID,
    unsigned CurVerison, ResourceTrackerSP OldRT, ThreadSafeModule &TSM) {
    TSM.withModuleDo([](llvm::Module &M) {
        auto PassManager = buildPassManager();
        PassManager.run(M);
    });
    return Error::success();
}
ReOptLayer->setReOptimizeFunc(reoptimizeToO2);
ReOptLayer->setAddProfilerFunc(reoptimizeIfCallFrequent);
```
BASIC USAGE OF REOPTIMIZATION API

Example: do -O2 optimization if function was called more than 10

```
static Error reoptimizeToO2(ReOptimizeLayer &Parent, ReOptMaterializationUnitID MUID,
      unsigned CurVerison, ResourceTrackerSP OldRT, ThreadSafeModule &TSM) {
    TSM.withModuleDo([&](llvm::Module &M) {
      auto PassManager = buildPassManager();
      PassManager.run(M);
    });
    return Error::success();
}
ReOptLayer->setReOptimizeFunc(reoptimizeToO2);
ReOptLayer->setAddProfilerFunc(reoptimizeIfCallFrequent);```
DEMO: CLANG-REPL WITH REOPT

- clang-repl is LLVM’s in-tree c++ interpreter based on ORC JIT API
- The code originally from CERN’s cling which has been used to analyze LHC data.
INTERNALS
INTERNALS

- **Redirection** to new symbol happens at **JIT linker** (JITLink) level
INTERNALS

- **Redirection** to new symbol happens at **JIT linker** (JITLink) level
- When it sees “**direct jump to symbol**” relocation, it records the **call sites**.
• **Redirection** to new symbol happens at **JIT linker** (JITLink) level
• When it sees “**direct jump to symbol**” relocation, it records the **call sites**.
• When reoptimization happens, **rewrite jump offset** of all call sites.
INTERNALS

- **Redirection** to new symbol happens at **JIT linker** (JITLink) level
- When it sees “**direct jump to symbol**” relocation, it records the **call sites**.
- When reoptimization happens, **rewrite jump offset** of all call sites.
- When this is not possible, **fall back to trampoline approach**.
INTERNALS

- **Redirection** to new symbol happens at **JIT linker** (JITLink) level.
- When it sees "**direct jump to symbol**" relocation, it records the **call sites**.
- When reoptimization happens, **rewrite jump offset** of all call sites.
- When this is not possible, **fall back to trampoline approach**.
  - indirect call to target or required offset is too large.
INTERNALS

- **Redirection** to new symbol happens at **JIT linker** (JITLink) level
- When it sees “**direct jump to symbol**” relocation, it records the **call sites**.
- When reoptimization happens, **rewrite jump offset** of all call sites.
- When this is not possible, **fall back to trampoline approach**.
  - indirect call to target or required offset is too large.
  - when platform prevents **writable and executable** memory for security reason.
INTERNALS
main

Call *func_

Call func_impl_v1

func

jmp *func_ptr

func_ptr

0x4242424242

func_impl_v1
main

Call *func_

Call func_impl_v1

func

jmp *func_ptr

func_ptr

0x4242424242

func_impl_v1
Call *func_
Call func_impl_v1

main

func
jmp *func_ptr
func_ptr
0x4242424242

func_impl_v1
main

Call *func_

Call func_impl_v1

func

jmp *func_ptr

func_ptr

0x4242424242

func_impl_v1
INTERNALS

main

Call *func_

Call func_impl_v1

func

jmp *func_ptr

func_ptr

0x4242424242

func_impl_v1

func_impl_v2
INTERNALS

main

Call *func_

Call func_impl_v1

func

jmp *func_ptr

func_ptr

0x4242424344

func_impl_v1

func_impl_v2
Call *func_*

Call func_impl_v1

func

jmp *func_ptr

func_ptr

0x4242424344

func_impl_v1

func_impl_v2
main

Call *func_

Call func_impl_v2

func

jmp *func_ptr

func_ptr

0x4242424344

func_impl_v1

func_impl_v2
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table

class Animal {
public:
    virtual void meow() {} 
};

int main() {
    Animal* animal;
    animal->meow();
}
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table

```cpp
class Animal {
public:
    virtual void meow() {};
};

int main() {
    Animal* animal;
    animal->meow();
}
```
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table

```cpp
class Animal {
public:
    virtual void meow() {};
};

int main() {
    Animal* animal;
    animal->meow();
}
```
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table

class Animal {
    public:
        virtual void meow() {}  
};

int main() {
    Animal* animal;
    animal->meow();
}

define i32 @main() {
    %1 = alloca ptr
    %2 = load ptr, ptr %1
    %3 = load ptr, ptr %2
    %4 = getelementptr inbounds ptr, ptr %3, i64 0
    %5 = load ptr, ptr %4, align 8
    call void %5
    ret i32 0
}
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table

class Animal {
    public:
        virtual void meow() {};
    }

    int main() {
        Animal* animal;
        animal->meow();
    }

define i32 @main() {
    %1 = alloca ptr
    %2 = load ptr, ptr %1
    %3 = load ptr, ptr %2
    %4 = getelementptr inbounds ptr, ptr %3, i64 0
    %5 = load ptr, ptr %4, align 8
    call void %5
    ret i32 0
}
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table

- **Performance implication**: hard to inline them since the destination address is decided in runtime
ADVANCED USAGE OF REOPTIMIZATION API

Virtual method table

- **Performance implication**: hard to inline them since the destination address is decided in runtime
  - Not just indirection cost but also lose opportunity for potential optimizations as values are not within the same basic block
ADVANCED USAGE OF REOPTIMIZATION API

De-virtualization
ADVANCED USAGE OF REOPTIMIZATION API

De-virtualization

- Looks at candidate destination addresses and inline some of them
ADVANCED USAGE OF REOPTIMIZATION API

De-virtualization

- Looks at candidate destination addresses and inline some of them
- If the function address is the known one, use the inlined body
ADVANCED USAGE OF REOPTIMIZATION API

De-virtualization

- Looks at candidate destination addresses and inline some of them
- If the function address is the known one, use the inlined body
- Otherwise fall back to indirect call
ADVANCED USAGE OF REOPTIMIZATION API

JIT implementation
ADVANCED USAGE OF REOPTIMIZATION API

JIT implementation

ORC JIT

JITted code

call %1
__orc_rt_increment_func_callcnt(%1)
__ort_rt_reoptimize(1)

JIT code buffer
ADVANCED USAGE OF REOPTIMIZATION API

JIT implementation

ORC JIT

orc_rt_reoptimizer.o

extern "C"
__orc_rt_increment_func_callcnt(void*);
extern "C"
__orc_rt_reoptimize(int);

"JIT-linked"

JITted code

call %1
__orc_rt_increment_func_callcnt(%1)
__ort_rt_reoptimize(1)

JIT code buffer
ADVANCED USAGE OF REOPTIMIZATION API

JIT implementation

ORC JIT

REMOTE RPC

Recorded destination addresses

JITted code

orc_rt_reoptimizer.o

"JIT-linked"

extern "C"
__orc_rt_increment_func_callcnt(void*);
extern "C"
__orc_rt_reoptimize(int);

JIT code buffer

call %1
__orc_rt_increment_func_callcnt(%1)
__ort_rt_reoptimize(1)
DEMO: CLANG-REPL WITH DEVIRTUALIZATION

- Showcasing the de-virtualization within clang-repl
BENCHMARKS

*all time values are average of 10 trials
## BENCHMARKS

<table>
<thead>
<tr>
<th>Program</th>
<th>-O1</th>
<th>Reoptimization ON</th>
<th>-O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boost Spirit (n=1)</td>
<td>1.97s</td>
<td>2.12s</td>
<td>2.24s</td>
</tr>
<tr>
<td>Boost Spirit (n=500)</td>
<td>22.46s</td>
<td>21.71s</td>
<td>21.55s</td>
</tr>
</tbody>
</table>

*all time values are average of 10 trials*
# BENCHMARKS

<table>
<thead>
<tr>
<th>Program</th>
<th>-O1</th>
<th>Reoptimization ON</th>
<th>-O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boost Spirit (n=1)</td>
<td>1.97s</td>
<td>2.12s</td>
<td>2.24s</td>
</tr>
<tr>
<td>Boost Spirit (n=500)</td>
<td>22.46s</td>
<td>21.71s</td>
<td>21.55s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>-O0</th>
<th>Reoptimization ON + Devirtualization OFF</th>
<th>-O2</th>
<th>Reoptimization ON + Devirtualization ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ray Tracer</td>
<td>158.9s</td>
<td>66.6s</td>
<td>66.0s</td>
<td>62.5s</td>
</tr>
</tbody>
</table>

*all time values are average of 10 trials*
## BENCHMARKS

<table>
<thead>
<tr>
<th>Program</th>
<th>-O1</th>
<th>Reoptimization ON</th>
<th>-O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boost Spirit (n=1)</td>
<td>1.97s</td>
<td>2.12s</td>
<td>2.24s</td>
</tr>
<tr>
<td>Boost Spirit (n=500)</td>
<td>22.46s</td>
<td>21.71s</td>
<td>21.55s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>-O0</th>
<th>Reoptimization ON + Devirtualization OFF</th>
<th>-O2</th>
<th>Reoptimization ON + Devirtualization ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ray Tracer</td>
<td>158.9s</td>
<td>66.6s</td>
<td>66.0s</td>
<td>62.5s</td>
</tr>
</tbody>
</table>

*all time values are average of 10 trials
ISSUE: INLINE MORE VS COMPILATE FAST
ISSUE: INLINE MORE VS COMPILE FAST

- We’d like to reoptimize by function level for the sake of compilation latency.
ISSUE: INLINE MORE VS COMPILE FAST

• We’d like to reoptimize by function level for the sake of compilation latency.
  ○ which means re-compilation by function level = splittered IR module
ISSUE: INLINE MORE VS COMPILE FAST

- We’d like to reoptimize by function level for the sake of compilation latency.
  - which means re-compilation by function level = splitted IR module
- ORC JIT currently have no standard way to inline out-of-module functions.
ISSUE: INLINE MORE VS COMPILE FAST

- We’d like to reoptimize by function level for the sake of compilation latency.
  - which means re-compilation by function level = splitted IR module
- ORC JIT currently have no standard way to inline out-of-module functions.
- Lack of inlining that would have happened in non-reopt mode.
ISSUE: INLINE MORE VS COMPILE FAST

- We’d like to reoptimize **by function level** for the sake of compilation latency.
  - which means re-compilation by function level = **splitted IR module**
- ORC JIT currently have **no standard way to inline out-of-module functions**.
- **Lack of inlining** that would have happened in non-reopt mode.
- The runtime performance drop observed to be **as bad as 3x slower**.
ISSUE: INLINE MORE VS COMPILE FAST

- We’d like to reoptimize by function level for the sake of compilation latency.
  - which means re-compilation by function level = splitted IR module
- ORC JIT currently have no standard way to inline out-of-module functions.
- Lack of inlining that would have happened in non-reopt mode.
- The runtime performance drop observed to be as bad as 3x slower.
- Current solution: don’t delete function when splitting module but just mark them externally_available.
 ISSUE: INLINE MORE VS COMPILER FAST

- We’d like to reoptimize by function level for the sake of compilation latency.
  - which means re-compilation by function level = **splitted IR module**
- ORC JIT currently have **no standard way to inline out-of-module functions**.
- **Lack of inlining** that would have happened in non-reopt mode.
- The runtime performance drop observed to be **as bad as 3x slower**.
- **Current solution:** don’t delete function when splitting module but just mark them externally_available.
  - but this introduces **compilation overhead** when module is large
  - $O(n^2)$ function duplicates where $n$ is number of functions
FUTURE GOALS
FUTURE GOALS

- LTO framework for ORC JIT
  - Can be used to tackle inlining issue.
  - Also can bring more performance to non-reopt applications.
FUTURE GOALS

- LTO framework for ORC JIT
  - Can be used to tackle inlining issue.
  - Also can bring more performance to non-reopt applications.
- Look into optimizing function with a huge loop up front
  - The penalty we get when we couldn’t re-optimize certain function are substantial
  - Penalty = cost for instrumentation + lost optimizations
FUTURE GOALS

- LTO framework for ORC JIT
  - Can be used to tackle inlining issue.
  - Also can bring more performance to non-reopt applications.
- Look into optimizing function with a huge loop up front
  - The penalty we get when we couldn’t re-optimize certain function are substantial
  - Penalty = cost for instrumentation + lost optimizations
- Generic JIT profile guided optimization framework
  - Could we possibly overhaul LLVM’s existing PGO infrastructure in order to reuse it?
THANKS

Code used today is available at:
https://github.com/sunho/LLVM-JIT-REOPT-Example