OUTLINE

How does JIT work in LLVM
OUTLINE

How does JIT work in LLVM

Motivation
OUTLINE

How does JIT work in LLVM
Motivation
Windows COFF JITLink example
OUTLINE

How does JIT work in LLVM
Motivation
Windows COFF JITLink example
Windows COFF JITLink plugin example
OUTLINE

How does JIT work in LLVM
Motivation
Windows COFF JITLink example
Windows COFF JITLink plugin example
Tips on using JITLink in COFF
OUTLINE

How does JIT work in LLVM
Motivation
Windows COFF JITLink example
Windows COFF JITLink plugin example
Tips on using JITLink in COFF
clang-repl demo
HOW DOES JIT WORK IN LLVM

Usual executable generation pipeline in LLVM
HOW DOES JIT WORK IN LLVM

Usual executable generation pipeline in LLVM
HOW DOES JIT WORK IN LLVM

Usual executable generation pipeline in LLVM

Frontend → Backend → Object files
HOW DOES JIT WORK IN LLVM

Usual executable generation pipeline in LLVM
HOW DOES JIT WORK IN LLVM

JIT execution pipeline in LLVM

Frontend → Backend → JIT Linker

Object files (in memory)
HOW DOES JIT WORK IN LLVM

JIT execution pipeline in LLVM

- Share a huge portion of pipeline with AOT
- Fewer breakage by LLVM internal code changes
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld

• Small code model
  unsupported
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld

- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld

- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
- Developed in ad-hoc fashion
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld

- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
- Developed in ad-hoc fashion
- COFF support existed but very unstable
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld

- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
- Developed in ad-hoc fashion
- COFF support existed but very unstable
  - People used ELF on Windows
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld

- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
- Developed in ad-hoc fashion
- COFF support existed but very unstable
  - People used ELF on Windows

New JIT linker: JITLink
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld
- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
- Developed in ad-hoc fashion
- COFF support existed but very unstable
  - People used ELF on Windows

New JIT linker: JITLink
- Small code model aware memory allocator
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld
- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
- Developed in ad-hoc fashion
- COFF support existed but very unstable
  - People used ELF on Windows

New JIT linker: JITLink
- Small code model aware memory allocator
- Runtime features fully supported including static initializers and thread local storage
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld
- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
- Developed in ad-hoc fashion
- COFF support existed but very unstable
  - People used ELF on Windows

New JIT linker: JITLink
- Small code model aware memory allocator
- Runtime features fully supported including static initializers and thread local storage
- Generic linker object abstraction LinkGraph
MOTIVATION FOR JITLINK

Old JIT linker: RuntimeDyld
- Small code model unsupported
- Static initializers or thread local storage (TLS) supported in limited ways
- Developed in ad-hoc fashion
- COFF support existed but very unstable
  - People used ELF on Windows

New JIT linker: JITLink
- Small code model aware memory allocator
- Runtime features fully supported including static initializers and thread local storage
- Generic linker object abstraction LinkGraph
- Easy to fully implement native object file features
COFF SUPPORT IN JITLINK
COFF SUPPORT IN JITLINK

- Capable of linking object files generated by MSVC
COFF SUPPORT IN JITLINK

- Capable of linking object files generated by MSVC
- COMDATs, WeakExternal, linker directive, dllimport stub, or CRT initializer properly implemented
COFF SUPPORT IN JITLINK

- Capable of linking object files generated by MSVC
- COMDATs, WeakExternal, linker directive, dllimport stub, or CRT initializer properly implemented
- Able to jit-link the **VC runtime library**
  - Loading up msvcrtn.lib ucrt.lib into JIT session
  - Static version of VC runtime works too
COFF SUPPORT IN JITLINK

- Capable of linking object files generated by MSVC
- COMDATs, WeakExternal, linker directive, dllimport stub, or CRT initializer properly implemented
- Able to jit-link the VC runtime library
  - Loading up msvcrtd.lib ucrt.lib into JIT session
  - Static version of VC runtime works too
- Linking Microsoft STL library work out of shelf
COFF SUPPORT IN JITLINK

- Capable of linking object files generated by MSVC
- COMDATs, WeakExternal, linker directive, dllimport stub, or CRT initializer properly implemented
- Able to jit-link the **VC runtime library**
  - Loading up msvcrtn.lib ucrt.lib into JIT session
  - Static version of VC runtime works too
- Linking **Microsoft STL library** work out of shelf
- **Incremental linking** works by default
We're going to build a simple JIT application
WINDOWS COFF JITLINK EXAMPLE

LLVM IR executor

We're going to build a simple JIT application

- Executes the **LLVM IRs** written inside main.ll using JIT
We're going to build a simple JIT application

- Executes the **LLVM IRs** written inside main.ll using JIT
- main.ll can be generated from any frontend such as clang or flang
EXECUTES THE LLVM IRs WRITTEN INSIDE MAIN.II USING JIT

MAIN.II CAN BE GENERATED FROM ANY FRONTEND SUCH AS CLANG OR FLANG

START BY IMPLEMENTING LLVM IR EXECUTOR AND ADD ADVANCED JIT USAGES ON TOP OF IT
WINDOWS COFF JITLINK EXAMPLE

LLVM IR executor

https://gist.github.com/sunho/12f14f61309323bfd88832f94056e68d

Setup code template
LLJIT::loadOrcRuntime function can be used to load orc runtime into JIT session.

orc_rt-x86_64.lib file is inside compiler-rt build
After ORC runtime is loaded, many features just work including:
After ORC runtime is loaded, many features just work including:

- sin, cos functions from vc runtime library
After ORC runtime is loaded, many features just work including:

- sin, cos functions from vc runtime library
- std::map, std::mutex, std::cout, and more
After ORC runtime is loaded, many features just work including:

- sin, cos functions from vc runtime library
- std::map, std::mutex, std::cout, and more
- c++ exception support
After ORC runtime is loaded, many features just work including:

- sin, cos functions from vc runtime library
- std::map, std::mutex, std::cout, and more
- c++ exception support
- Structured Exception Handling (SEH) support
WINDOWS COFF JITLINK EXAMPLE
LLVM IR executor

Loading static library built by MSVC into JIT session
WINDOWS COFF JITLINK EXAMPLE

LLVM IR executor

Loading static library built by MSVC into JIT session

```cpp
auto G = ExitOnErr(StaticLibraryDefinitionGenerator::Load(
    J->getObjectLinkingLayer(), "StaticLib1.lib")
);
J->getMainJITDylib().addGenerator(std::move(G));
```
WINDOWS COFF JITLINK EXAMPLE

LLVM IR executor

Loading static library built by MSVC into JIT session

```cpp
auto G = ExitOnErr(StaticLibraryDefinitionGenerator::Load(
    J->getObjLinkingLayer(), "StaticLib1.lib")
);
J->getMainJITDylib().addGenerator(std::move(G));
```

Notice

- object files generated by native compiler successfully linked
WINDOWS COFF JITLINK EXAMPLE

LLVM IR executor

Loading static library built by MSVC into JIT session

```cpp
auto G = ExitOnErr(StaticLibraryDefinitionGenerator::Load(
    J->getObjLinkingLayer(), "StaticLib1.lib")
);
J->getMainJITDylib().addGenerator(std::move(G));
```

Notice

- object files generated by native compiler successfully linked
- native static initializers inside static library worked out of shelf
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Background

Overview of JITLink
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Background

Overview of JITLink

- Different formats of object files: ELF, MachO, COFF
Overview of JITLink

- Different formats of object files: ELF, MachO, COFF
- Different architecture of binary code: x86_64, aarch64, risc-v, ppc
Overview of JITLink

- Different formats of object files: ELF, MachO, COFF
- Different architecture of binary code: x86_64, aarch64, risc-v, ppc
- JITLink **converts** object file into generic linker object representation **LinkGraph**
  - ELFLinkGraphBuilder, COFFLinkGraphBuilder, MachOLinkGraphBuilder
Overview of JITLink

- Different formats of object files: ELF, MachO, COFF
- Different architecture of binary code: x86_64, aarch64, risc-v, ppc
- JITLink **converts** object file into generic linker object representation **LinkGraph**
  - ELFLinkGraphBuilder, COFFLinkGraphBuilder, MachOLinkGraphBuilder
- Then, it performs generic **memory allocation, symbol resolution** as described in **LinkGraph** and perform architecture-specific **relocations** as needed
WINDOWS COFF JITLINK PLUGIN EXAMPLE
Overview of LinkGraph

Block (Code)

mov rdi, 1
mov rsi, message
jmp printf

Block (Data)

"Hello, world"
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Overview of LinkGraph

Block (Code)

```
mov rdi, 1
mov rsi, message
jmp printf
```

Block (Data)

"Hello, world"

Symbol

Message
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Overview of LinkGraph

Block (Code)

```
mov rdi, 1
mov rsi, message
jmp printf
```

Block (Data)

"Hello, world"

Edge (Relocation)

```
IMAGE_REL_AMD64_REL32
```

Symbol

Message
class ExamplePlugin : public ObjectLinkingLayer::Plugin {
public:
void modifyPassConfig(MaterializationResponsibility &MR,
    jitlink::LinkGraph &G,
    jitlink::PassConfiguration &Config) override {
    Config.PrePrunePasses.push_back([&](jitlink::LinkGraph &G) {
        G.dump(llvm::outs());
        return Error::success();
    });
}
}
class ExamplePlugin : public ObjectLinkingLayer::Plugin {
  public:
  void modifyPassConfig(MaterializationResponsibility &MR,
    jitlink::LinkGraph &G,
    jitlink::PassConfiguration &Config) override {
    Config.PrePrunePasses.push_back([&](jitlink::LinkGraph &G) {
      G.dump(llvm::outs());
      return Error::success();
    });
  }
}
Basic plugin

class ExamplePlugin : public ObjectLinkingLayer::Plugin {
public:
    void modifyPassConfig(MaterializationResponsibility &MR,
        jitlink::LinkGraph &G,
        jitlink::PassConfiguration &Config) override {
        Config.PrePrunePasses.push_back([&](jitlink::LinkGraph &G) {
            G.dump(llvm::outs());
            return Error::success();
        });
    }
};

Callback gets called in specific linking phase
Unwind frame visualizer plugin

- Print the summary of unwind frame of each function contained inside object file
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Unwind frame visualizer plugin

Windows unwind frame

example.obj

.pdata

RUNTIME_FUNCTION

RUNTIME_FUNCTION

RUNTIME_FUNCTION
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Unwind frame visualizer plugin

Windows unwind frame

eexample.obj

- Each RUNTIME_FUNCTION has code range of function and unwind info
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Unwind frame visualizer plugin

Windows unwind frame

example.obj

.pdata

- Each RUNTIME_FUNCTION has code range of function and unwind info
- RUNTIME_FUNCTIONs emitted to .pdata section
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Unwind frame visualizer plugin

Windows unwind frame

example.obj

- Each RUNTIME_FUNCTION has code range of function and unwind info
- RUNTIME_FUNCTIONs emitted to .pdata section
- JITLink plugin system allows access to this .pdata section
WINDOWS COFF JITLINK PLUGIN EXAMPLE
Unwind frame visualizer plugin

Coding time

- PrePrune
- PostPrune
- PostAllocation
- PreFixup
- PostFixup
- dead
- strip
- allocate
- block mem
- resolve
- symbols
- apply
- relocation
Exception instrumentation plugin

- Show disassembly of function that just raised exception
FUNC call _CxxThrowException

WINDOWS COFF JITLINK PLUGIN EXAMPLE
Exception instrumentation plugin
Exception instrumentation plugin

```
Func
  call <_CxxThrowException>
```

Trampoline
Windows COFF JITLink Plugin Example

Exception instrumentation plugin

Func
call <_CxxThrowException>

Trampoline

Save register
call <ThrowIntercept>
Restore register
call <_CxxThrowException>
**WINDOWS COFF JITLINK PLUGIN EXAMPLE**

Exception instrumentation plugin

- **Func**
  - `call <_CxxThrowException>`

- **ThrowIntercept**
  - `call <ThrowIntercept>`
  - `Restore register`
  - `call <_CxxThrowException>`

- **Trampoline**
  - `Save register`
  - `call <ThrowIntercept>`
  - `Restore register`
  - `call <_CxxThrowException>`

- Code that prints disassembly of **Func**
 WINDOWS COFF JITLINK PLUGIN EXAMPLE
Exception instrumentation plugin

```
std::vector<jitlink::Edge> Edges;
std::vector<char> CodeBuf;

// Write x86 assembly code to CodeBuf
WriteSaveRegsCode(CodeBuf);
WriteCallFuncCode(CodeBuf);
```
**WINDOWS COFF JITLINK PLUGIN EXAMPLE**

Exception instrumentation plugin

```cpp
std::vector<jitlink::Edge> Edges;
std::vector<char> CodeBuf;

// Write x86 assembly code to CodeBuf
WriteSaveRegsCode(CodeBuf);
WriteCallFuncCode(CodeBuf);
```

**CodeBuf (content bytes of block)**

```
0: pushq %rbp
1: movq %rsp, %rbp
4: subq $512, %rsp
b: movq %rcx, -16(%rbp)
f: movq %rdx, -24(%rbp)
13: movq %rsi, -32(%rbp)
17: movq %rdi, -40(%rbp)
...
6f: e8 00 00 00 00 callq <ThrowIntercept>
```
WINDOW COFF JITLINK PLUGIN EXAMPLE

Exception instrumentation plugin

```cpp
std::vector<jitlink::Edge> Edges;
std::vector<char> CodeBuf;

// Write x86 assembly code to CodeBuf
WriteSaveRegsCode(CodeBuf);
WriteCallFuncCode(CodeBuf);
```

<table>
<thead>
<tr>
<th>CodeBuf (content bytes of block)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: pushq %rbp</td>
</tr>
<tr>
<td>1: movq %rsp, %rbp</td>
</tr>
<tr>
<td>4: subq $512, %rsp</td>
</tr>
<tr>
<td>b: movq %rcx, -16(%rbp)</td>
</tr>
<tr>
<td>f: movq %rdx, -24(%rbp)</td>
</tr>
<tr>
<td>13: movq %rsi, -32(%rbp)</td>
</tr>
<tr>
<td>17: movq %rdi, -40(%rbp)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>6f: e8 00 00 00 00 callq &lt;ThrowIntercept&gt;</td>
</tr>
</tbody>
</table>
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Exception instrumentation plugin

```cpp
1  std::vector<jitlink::Edge> Edges;
2  std::vector<char> CodeBuf;
3  
4  // Write x86 assembly code to CodeBuf
5  WriteSaveRegCode(CodeBuf);
6  WriteCallFuncCode(CodeBuf);
7  
8  // Add relocation edge to ThrowIntercept
9  auto ThrowInterceptSymbol =
10     &G.addExternalSymbol("ThrowIntercept", 0, jitlink::Linkage::Strong);
11  Edges.push_back(jitlink::Edge(jitlink::x86_64::PCRel32, CodeBuf.size() - 4,
12     +ThrowInterceptSymbol, 0));
```

CodeBuf (content bytes of block)

```assembly
1  0: pushq %rbp
2  1: movq %rsp, %rbp
3  4: subq $512, %rsp
4  b: movq %rcx, -16(%rbp)
5  f: movq %rdx, -24(%rbp)
6  13: movq %rsi, -32(%rbp)
7  17: movq %rdi, -40(%rbp)
8  ...
9  6f: e8 00 00 00 00 callq <ThrowIntercept>
```

jitlink::x86_64::PCRel32

Edge

ThrowIntercept

Symbol
WINDOWS COFF JITLINK PLUGIN EXAMPLE

Exception instrumentation plugin

Coding time
TIPS ON USING JITLINK IN COFF

ORC Runtime at startup

MainJD

- COFFPlatform loads up vcruntime lib files and ORC runtime lib file.
- Uses JIT-linked orc runtime function to register and run static initializers
- ORC runtime itself uses JIT-linked STL library

New object file

```
CRT$XCB
CRT$XCC
```
TIPS ON USING JITLINK IN COFF
ORC Runtime at startup

Tips

- Care is needed to make sure ORC and vc runtime library files are available
  - by default, vc runtime libraries automatically detected from VC toolchain directories (can fail)
- Customizing vc runtime loading can be done by COFFVCRuntimeBootstrapper class
- It is still possible to use in-process vc runtime symbols, but need to export required symbols manually by using linker directive
  - #pragma comment(linker, "/export:??_7type_info@@6B@")
TIPS ON USING JITLINK IN COFF

JITDYLIB: Emulated DYLIB inside JIT session

Challenges with COFF small code model

- Compilers assume that all symbols within the same executable or dylib are allocated close together
- It is not possible to "patch" instructions to use GOT pointer on demand when the required displacement exceeds 2Gb
- COFF x86 relocation points to the middle of instruction bytes
  - x86 encoding is not possible to be read backwards to know the start of instruction (for instructions of interest because of presence of RAX prefix)
  - -> can’t patch this part
TIPS ON USING JITLINK IN COFF

JITDYLIB: Emulated DYLIB inside JIT session

- Emulated dylib inside JIT session
- dlopen and dlclose JITDYLib inside JITted code

JITLink memory manager enforces the distance from ImageBase to not exceed larger amount

```
0x1000
| Some bytes |

0x1001
| mov rdi, 1 |
| mov rsi, message |
| jmp printf |

0x1053
| mov rdi, 1 |
| mov rsi, message |
| jmp printf |
```
TIPS ON USING JITLINK IN COFF

JITDYLIB: Emulated DYLIB inside JIT session

**JITDYLIB A**

- ImageBase: 0x1000
  - "Some bytes"
- CallFunc: 0x1001
  - call LocalFunc
- CallFunc2: 0x1053
  - call *__imp_LocalFunc
- LoadVar: 0x1068
  - mov rax, *__imp_LocalVar

**JITDYLIB B**

- ImageBase: 0xFFFFFFFF1000
  - "Some bytes"
- LocalFunc: 0xFFFFFFFF1001
  - mov rdi, 1
  - mov rsi, message
  - jmp printf
- LocalVar: 0xFFFFFFFF1053
  - "Some bytes"

**Notes**

- Through jump stub
- Through import stub pointer
- Through import stub pointer
TIPS ON USING JITLINK IN COFF

JITDYLIB: Emulated DYLIB inside JIT session

Tips

- Call function of another JITDYLib through usual call or dllimport attribute (___imp___)
- Access data of another JITDYLib only through dllimport attribute (___imp___)
- Same practices are required in AOT world too but less clear in JIT world
CLANG-REPL DEMO

- Everything is in-tree in LLVM including clang-repl executable
THANKS