ORC

LLVM’s Next Generation of JIT API
Contents

• LLVM JIT APIs Past, Present and Future
• I will praise MCJIT
• Then I will critique MCJIT
• Then I’ll introduce ORC
• Code examples available in the Building A JIT tutorial on llvm.org
Use Cases

<table>
<thead>
<tr>
<th>Kaleidoscope</th>
<th>LLDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple and safe</td>
<td>Cross-target compilation</td>
</tr>
<tr>
<td>High Performance JITs</td>
<td>Interpreters and REPLs</td>
</tr>
<tr>
<td>Ability to configure</td>
<td>Lazy compilation</td>
</tr>
<tr>
<td>optimizations and codegen</td>
<td>Equivalence with static compile</td>
</tr>
</tbody>
</table>
We can support all of these requirements

But not behind a single interface…
void addModule(Module*);

void *getPointerToFunction(Function*);

void addGlobalMapping(Function*, void*);

// Many terrible things that, trust me, you really don’t want to know about.
JIT Implementations

• Legacy JIT (LLVM 1.0 — 3.5)
  • Introduced ExecutionEngine
  • Lazy compilation, in-process only

• MCJIT (LLVM 2.9 — present)
  • Implements ExecutionEngine
  • Cross-target, no lazy compilation

• ORC (LLVM 3.7 — present)
  • Forward looking API
  • Does NOT implement ExecutionEngine
MCJIT Design

- Static Pipeline with JIT Linker
- Efficient code and tool re-use
- Supports cross-target JITing
- Does not support lazy compilation
MCJIT Implementation

- Only accessible via ExecutionEngine
- Caused ExecutionEngine to bloat
- Can not support all of ExecutionEngine
ExecutionEngine

Symbol Query Horrors...

```c
void *getPointerToFunction(Function*)
uint64_t getFunctionAddress(const std::string&)
void *getPointerToNamedFunction(StringRef)
void *getPointerToFunctionOrStub(Function*)
uint64_t getAddressToGlobalIfAvailable(StringRef)
void *getPointerToGlobalIfAvailable(StringRef)
void *getPointerToGlobal(const GlobalValue*)
uint64_t getGlobalValueAddress(const std::string&)
void *getOrEmitGlobalVariable(const GlobalVariable*)
```
MCJIT Implementation

- Only accessible via ExecutionEngine
- Caused ExecutionEngine to bloat
- Can not support all of ExecutionEngine
- Limited visibility into internal actions
- No automatic memory management
ORC — On Request Compilation

A Modular MCJIT
Modularizing MCJIT

MCJIT
- CodeGen
- MC
- Object
- RuntimeDyld
- Raw Bits
Modularizing MCJIT

Compile Layer
- CodeGen
- MC

Link Layer
- RuntimeDyld
- Raw Bits

Compile Layer forwards symbol queries to the link layer
Initial Benefits

- Layers can be tested in isolation
Initial Benefits

- Layers can be tested in isolation
- E.g. Unit test the link layer
Initial Benefits

- Layers can be tested in isolation
- E.g. Unit test the link layer
- Observe events without callbacks
  - E.g. Add a notification layer
The Layer Interface

- Handle `addModule(Module*, MemMgr*, Resolver*)`
- Memory manager owns executable bits
- Resolver provides symbol resolution
- `JITSymbol findSymbol(StringRef, bool)`
- `void removeModule(Handle)`
Example: Basic Composition

...

ObjectLinkingLayer LinkLayer;
SimpleCompiler Compiler(TargetMachine());
IRCompileLayer<...> CompileLayer(LinkLayer, Compiler);
...

Example: Basic Composition

class MyJIT {
    ...
    ObjectLinkingLayer LinkLayer;
    SimpleCompiler Compiler(TargetMachine());
    IRCompileLayer<...> CompileLayer(LinkLayer, Compiler);
    ...
};
Example: Basic Composition

... 
ObjectLinkingLayer LinkLayer;
SimpleCompiler Compiler(TargetMachine());
IRCompileLayer<...> CompileLayer(LinkLayer, Compiler);
...
Example: Basic Composition

... ObjectLinkingLayer LinkLayer;
SimpleCompiler Compiler(TargetMachine());
IRCompileLayer... CompileLayer(LinkLayer, Compiler);

CompileLayer.addModule(Mod, MemMgr, SymResolver);
auto FooSym = CompileLayer.findSymbol("foo", true);
auto Foo = reinterpret_cast<int(*)(())>(FooSym.getAddress());
int Result = Foo(); // ←— Call into JIT’d code.
...
Memory Managers

- Own executable code, free it on destruction
- Inherit from RuntimeDyld::MemoryManager
- Custom memory managers supported
- SectionMemoryManager provides a good default
Symbol Resolvers

```cpp
auto Resolver =
    createLambdaResolver(
        [&](StringRef Name) {
            return CompileLayer.findSymbol(Name, false);
        },
        [&](StringRef Name) {
            return getSymbolAddressInProcess(Name);
        });
```

- First lambda implements in-image lookup
- Second implements external lookup
The Story So Far

• Layers wrap up JIT functionality to make it composable
• Build custom JITs by composing layers
• Memory managers handle memory ownership
• Symbol resolvers handle symbol resolution
Adding New Features

- New layers provide new features
- Compile On Demand Layer
  - `addModule` builds function stubs that trigger lazy compilation
  - Symbol queries resolve to stubs

Compile On Demand

Compile

Link
Without Laziness

```cpp
ObjectLinkingLayer LinkLayer;
SimpleCompiler Compiler(TargetMachine());
IRCompileLayer<...> CompileLayer(LinkLayer, Compiler);

CompileLayer.addModule(Mod, MemMgr, SymResolver);
auto FooSym = CompileLayer.findSymbol("foo", true);
auto Foo = reinterpret_cast<int(*)()>(FooSym.getAddress());
int Result = Foo(); // Call foo.
```
ObjectLinkingLayer LinkLayer;
SimpleCompiler Compiler(TargetMachine());
IRCompileLayer<...> CompileLayer(LinkLayer, Compiler);
CompileOnDemandLayer<...> CODLayer(CompileLayer, ...);

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.
COD Layer Requirements

• Indirect Stubs Manager
  • Create named indirect stubs (indirect jumps via pointers)
  • Modify stub pointers

• Compile Callback Manager
  • Create compile callbacks (re-entry points in the compiler)
Compile Callbacks

bar:
... call foo ...

foo:
  jmp *foo$ptr

foo$impl:
... ret

foo$cc:
  push foo_key
  jmp Resolver

Resolver

ORC/LLVM
auto StubsMgr = ... ;
auto CCMgr = ... ;

auto CC = CCMgr.getCompileCallback();
StubsMgr.createStub("foo", CC.getAddress(), Exported);

CC.setCompileAction([&]() -> TargetAddress {
    printf("Hello world");
    return 0;
});

auto FooSym = StubsMgr.findStub("foo", true);
auto Foo = static_cast<int(*)()>(FooSym.getAddress());
int Result = Foo();

Prints “Hello world”, then jumps to 0
callbacks and stubs

```cpp
auto StubsMgr = ...;
auto CCMgr = ...;

auto CC = CCMgr.getCompileCallback();
StubsMgr.createStub("foo", CC.getAddress(), Exported);

CC.setCompileAction([&]() -> TargetAddress {
    CompileLayer.addModule(FooModule, MemMgr, Resolver);
    return CompileLayer.findSymbol("foo", true).getAddress();
});

auto FooSym = StubsMgr.findStub("foo", true);
auto Foo = static_cast<int(*)()>(FooSym.getAddress());
int Result = Foo();
```

Lazily compiles "foo" from existing IR
Callbacks and Stubs

```cpp
auto StubsMgr = ... ;
auto CCMgr = ... ;

auto CC = CCMgr.getCompileCallback();
StubsMgr.createStub("foo", CC.getAddress(), Exported);

CC.setCompileAction([&]() -> TargetAddress {
    CompileLayer.addModule(IRGen(FooAST), MemMgr, Resolver);
    return CompileLayer.findSymbol("foo", true).getAddress();
});

auto FooSym = StubsMgr.findStub("foo", true);
auto Foo = static_cast<int(*)()>(FooSym.getAddress());
int Result = Foo();
```

Lazily compiles “foo” from AST
Laziness Recap

• Callbacks and Stubs
  • Provide direct access to lazy compilation
  • Push laziness earlier in the compiler pipeline

• CompileOnDemand provides off-the-shelf laziness for IR

• ORC supports arbitrary laziness with a clean API
Adding New Layers

- Transform Layer
  - `addModule` runs a user-supplied transform function on the module
  - Symbol queries are forwarded
  - Above C.O.D.: Eager optimizations
  - Below C.O.D.: Lazy optimizations

Transform

Compile On Demand

Compile

Link
Layers and Modularity

Pick features “off the shelf”

Mix and match components:
experiment with new designs

Create, modify and share new features
without breaking existing clients
Remote JIT Support
Remote JIT Support

- Execute code on a different process / machine / architecture
- Enables JIT code to be sandboxed
- MCJIT supported remote compilation, but required a lot of manual work
- OrcRemoteTarget client/server provides high level API
  - Remote mapped memory, stub and callback managers
  - Symbol queries
  - Execute remote functions
Local Laziness

```cpp
auto StubsMgr = ...;
auto CCMgr = ...

auto CC = CCMgr.getCompileCallback();
StubsMgr.createStub("foo", CC.getAddress(), Exported);

CC.setCompileAction([&]() -> TargetAddress {
    CompileLayer.addModule(IRGen(FooAST), MemMgr, Resolver);
    return CompileLayer.findSymbol("foo", true).getAddress();
});

auto FooSym = StubsMgr.findStub("foo", true);
auto Foo = static_cast<int(*)()>(FooSym.getAddress());
int Result = Foo();
```
auto RT = ...;
auto StubsMgr = RT.createStubsMgr();
auto CCMgr = RT.createCallbackMgr();

auto CC = CCMgr.getCompileCallback();
StubsMgr.createStub("foo", CC.getAddress(), Exported);

CC.setCompileAction([&] () -> TargetAddress {
  CompileLayer.addModule(IRGen(FooAST), RT.createMemMgr(),
                          Resolver);
  return CompileLayer.findSymbol("foo", true).getAddress();
});

auto FooSym = StubsMgr.findStub("foo", true);
int Result = RT.callIntVoid(FooSym.getAddress());
Demo
Remote JIT Support

• Remote JITing with ORC is easy

• Remoteness is orthogonal to other features, including laziness

• Security implications are serious
  • Sandbox the server, authenticate the client, encrypt the channel
  • Treat like mains electricity: very useful, but safety first!
Future Opportunities

• New development modes: edit/test vs edit/compile/test
• Remote interpreters for development on embedded devices
• Distributing work for clusters
  • Compute
  • Database queries
ORC vs MCJIT

- Same underlying architecture: static compiler + JIT linker
- ORC
  - Offers a strict superset of features
  - A more flexible API
  - Supports remoteness and laziness
  - Has better memory management
- OrcMCJITReplacement provides a transition path
Future Goals

• Kill off ExecutionEngine, design a new in-tree JIT (for LLI and C-API)
• New layers and components (e.g. hot function recompilation)
• API cleanup: Core abstractions are in place but need polish
• More architectural and relocation support (Fix RuntimeDyldELF!)
• Check out the Building A JIT tutorial
• Get involved: http://llvm.org/bugs, OrcJIT component