ORC 2021

Recent developments and future work in LLVM's JIT APIs

Lang Hames — lhames@gmail.com

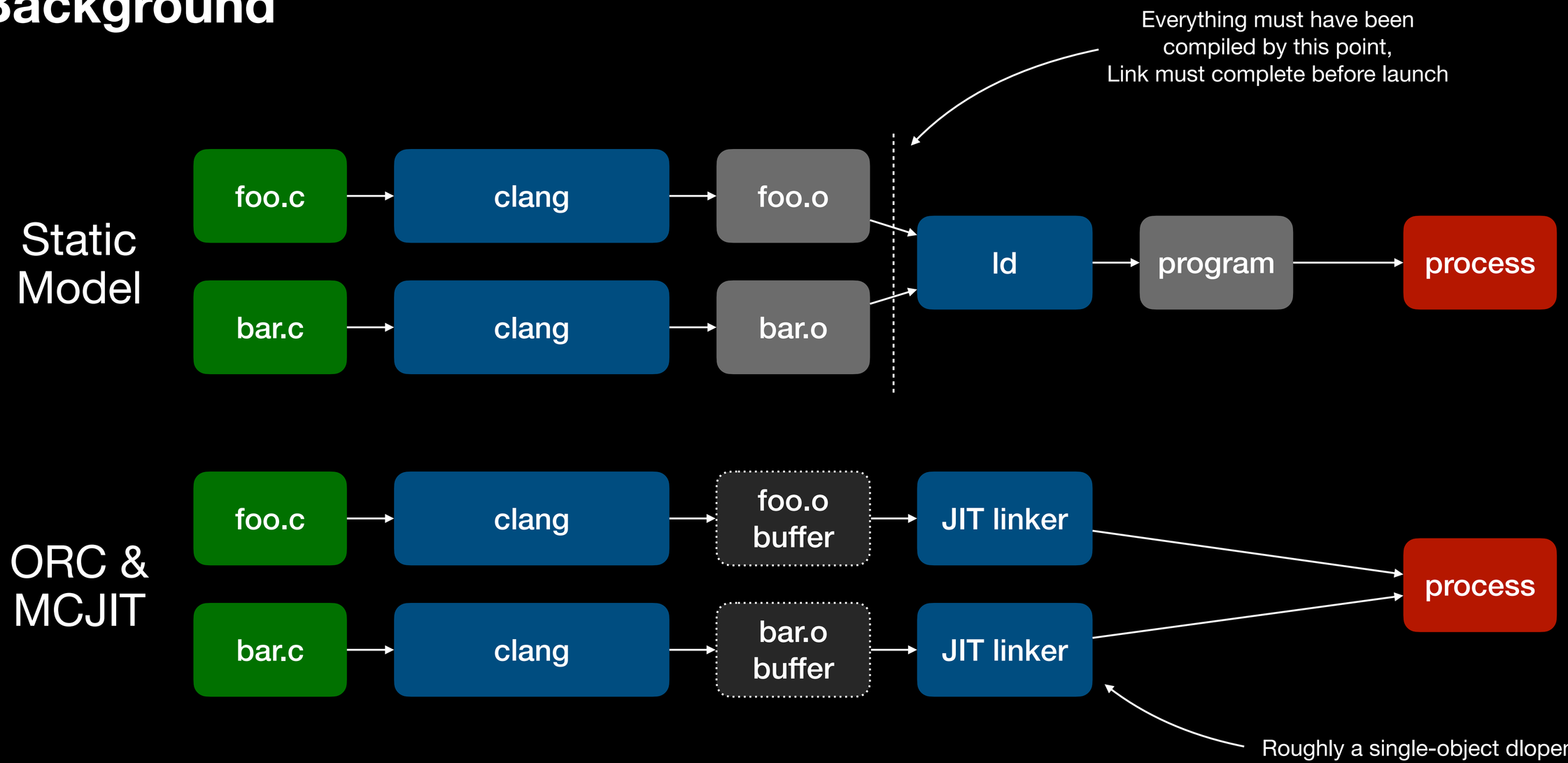
ORC

Background

- Motivation: Re-use existing compilers in more dynamic contexts
- Solution: Use a modified link stage

ORC

Background



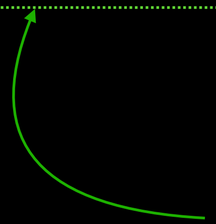
ORC
Background

- Motivation: Re-use existing compilers in more dynamic contexts
- Solution: Use a modified link stage
- Aim: Provide modular APIs for building dynamic compilation environments that replicate the behavior of statically compiled code
  - All object file features supported (MCJIT only supports a subset)
  - Efficient use of resources, including laziness and concurrency
  - Flexible APIs to enable re-use in many different contexts

ORC

Project History

- 2016: ORC v1 introduced as a modular MCJIT with lazy compilation support
- 2018: ORC v2 introduced, a redesign for concurrency
- 2019: JITLink introduced to (eventually) replace RuntimeDyld
- 2021: ORC Runtime introduced to support advanced object format features



New since LLVM Dev 2018

ORC

Today's topics

- ORC Core — Removable code, ExecutorProcessControl, Platforms
- JITLink — Goals, Design, API, Support status
- The ORC Runtime — Goals, Design, API, Support status
- Extra topics (if we have time)
  - Concurrent compilation, lazy compilation, re-optimization, PGO

ORC Core

ORC Core

Recap

- ExecutionSession represents a JIT'd program as a set of JITDylibs
- JITDylibs are symbol tables, containers for uncompiled modules
- Symbol address lookup triggers compilation
- Lookup is for sets of symbols, compiles may run concurrently
- ExecutionSession guarantees that lookups are "safe"
  - You can look up any set of symbols on any thread at any time, and the result is only returned once they're safe to access in the executor process. We use this property a lot.

ORC Core
Removable Code

- Why remove code?
  - One-shot code — Anonymous REPL expressions, various initializers
  - Stale code — Unoptimized code replaced by an optimized version
- Goals for our removable code feature:
  - Always allow removal at JITDylib granularity
  - Optionally track each module (with some administration cost)
- Non-goal: Managing dependencies within JIT'd code — that's the client's job

ORC Core

ResourceTracker API — Client View

- Create from JITDylibs: `auto RT = JD.createResourceTracker()`
- Associate modules: `IRLayer.add(RT, createIRModule(…))`
- Free resources: `if (auto Err = RT.remove()) { … }`
- Transfer resources: `RT1.transferTo(RT2);`

- Each JITDylib has a default ResourceTracker
- Implicit transfer to default tracker on ResourceTracker destruction — no leaks

ORC Core

ResourceTracker API — Client View

```
void addModuleM(JITDylib &JD) {
  auto M = createIRModule(…);
  IRLayer.add(JD, std::move(M));
  //‘M’ can not be removed individually.
  // JD can still be removed in its entirety.
}
```

ORC Core

ResourceTracker API — Client View

```
void addModuleM(JITDylib &JD) {
  auto M = createIRModule(…);
  auto RT = JD.createResourceTracker();
  IRLayer.add(RT, std::move(M));
  //‘M’ can now be removed individually:
  if (auto Err = RT->remove()) { … }
}
```

ORC Core

ResourceTracker API — Resource Manager View

- Each ResourceTracker has a hidden ResourceKey associated with it
- Associate resources with ResourceKeys safely using withResourceKeyDo:

```
MR.withResourceKeyDo([&](ResourceKey K) {
    MyResourceMap[K] = std::move(ResourceObject);
  });
```

- Inherit from ResourceManager to listen for resource notifications:

```
virtual Error handleRemoveResources(ResourceKey K);
virtual void handleTransferResources(ResourceKey DstK,
                                     ResourceKey SrcK);
```

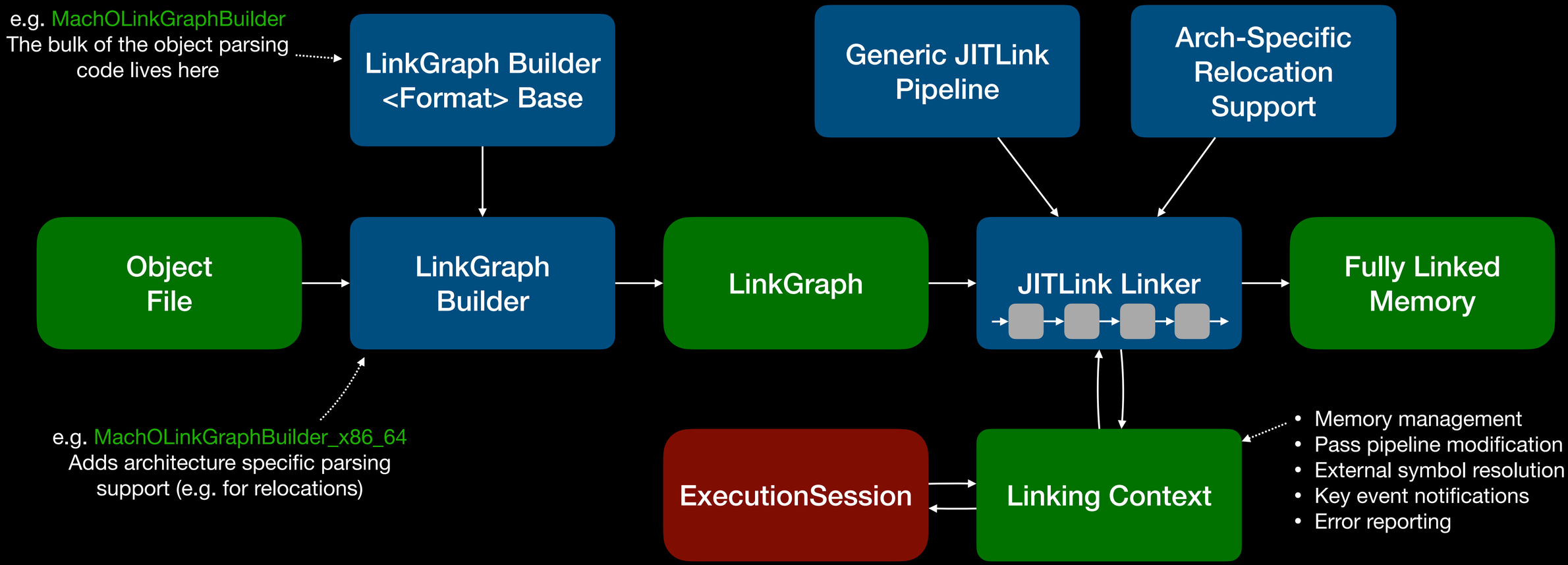
JITLink

JITLink
Overview

- JIT linker implementation, aims to replace RuntimeDyld (MCJIT's JIT linker)
- Provides an open graph-based linker data structure — the LinkGraph
- Linker implemented as a series of passes that can be extended by plugins
- Format and architecture support carefully separated
- Aiming for full relocation support to enable use of native code model
- LinkGraph exposes enough information to implement initializers, TLV, …
- Error checking built in from the ground up using llvm::Error

llvm-jitlink demo 1…

JITLink
Pipeline



JITLink

LinkGraph

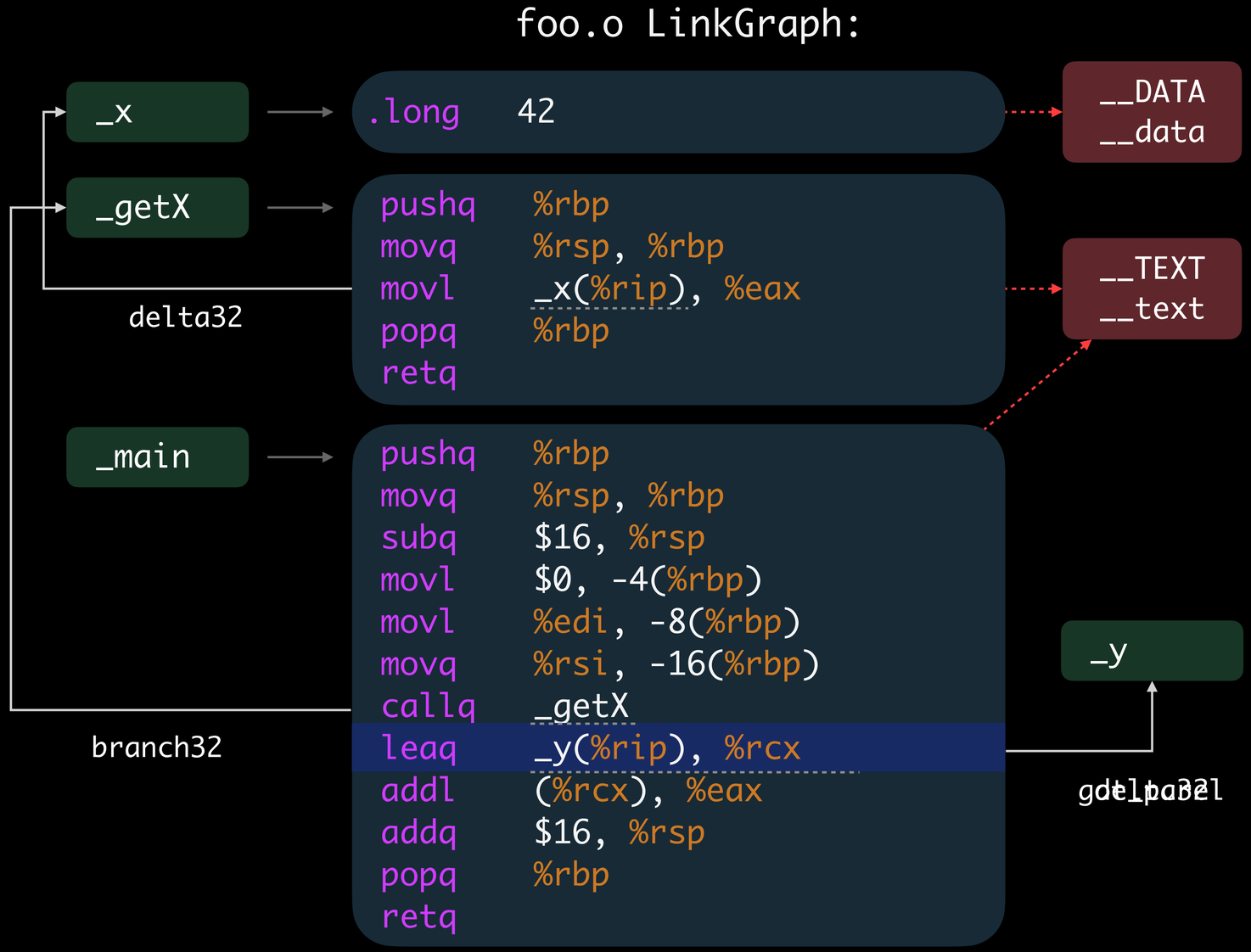
foo.c source:

```
static int x = 42;

extern int y;

int getX() { return x; }

int main(int argc,
         char *argv[]) {
  return getX() + y;
}
```



JITLink

Plugins and Passes

- Plugins are added to the ObjectLinkingLayer (typically at start-up)
- Plugins receive a callback for each graph to modify the pass pipeline
- Plugins can install passes to inspect/mutate the LinkGraph at each of five phases
  - *Pre-prune*: Before dead-stripping
  - *Post-prune*: After dead-stripping, before memory allocation
  - *Post-allocation*: After allocation, before external symbol resolution
  - *Pre-fixup*: After external symbol resolution, before fixups are applied
  - *Post-fixup*: On final content, after it has been shipped to target and protected

JITLink

Pass Example

```
Error interposeBranchEdges(jitlink::LinkGraph &G) {
  for (auto *B : G.blocks())
    for (auto &E : B->edges())
      if (E.getKind() == branch32 && E.getTarget().hasName())
        E.setTarget(getStubFor(E.getTarget()));
  return Error::success();
}
```

- Interpose all branches…
- OR record all branch locations and rewrite call-sites directly to avoid indirection…
- OR insert a nop-sled at the start of each function to overwrite later…
- OR some combination, e.g. interpose cold sites, record hot ones

JITLink

Debugger Support

- Recent work by Stefan Graenitz enabled use of the GDBLoaderJIT system with JITLink/ELF using the DebugObjectManagerPlugin (see llvm/examples/OrcV2Examples/LLJITWithRemoteDebugging)
- MachO support for a modified version of this is under development
  - JITLink's support for MachO subsections complicates the design: target layout cannot be described in terms of section addresses alone
- Longer term designs also under discussion
  - E.g. modifications to GDBLoaderJIT to allow lazy registration of debug info

JITLink

Format and Architecture Support

- MachO
  - x86-64 — Near complete support
  - arm64 — Missing TLV, compact unwind, pointer authentication
- ELF
  - x86-64 — Partial support, approaching RuntimeDyld
  - Generic ELFLinkGraphBuilder makes new backends relatively easy
- COFF… Volunteers needed

ORC Runtime

ORC Runtime

Overview

- Aims:
  - Support advanced object file features
  - Re-home code that currently lives in LLVM, e.g. JIT re-entry functions
  - Provide new utilities, e.g. memory-access APIs
  - Provide a universal “run-JIT’d-program” API
- Solution: Add orc-runtime to compiler-rt, load via ORC alongside user code

ORC Runtime

Advanced Object File Features

- Features…
  - Static initializers — Knows how to run them given init section address
  - Static deinitializers — Interposes __cxa_atexit to record JIT destructors
  - Exceptions — Knows how to register with libunwind
  - Native Thread Locals — Contains a JIT-friendly TLV runtime
  - Language runtimes — Knows how to register with ObjC, Swift runtimes
- Works cooperatively with JITLink to enable these features

ORC Runtime

Utility Code

- Feature-set requiring executor support is growing — supporting it via RPC and LLVM libraries would be cumbersome
- Instead, let’s put this support in a runtime library and load it via the JIT
- Needn’t constantly relink the executor — runtime can co-evolve with the JIT
- Compiler-rt is a good home — the build system is already runtime friendly
  - Can write low-level code (e.g. TLV runtime, re-entry code) as assembly
  - High level code can be written in C or C++

ORC Runtime

Universal “Run JIT Program” utility

- Universal — In particular, should run the same in-process or out-of-process
  - Calling via RPC (e.g. for runConstructors/runDestructors) can affect which threads JIT’d constructors/destructors run on
- Define run_jit_program function in terms of JIT versions of dlfcn.h functions…

```
void *JPH = jit_dlopen(ProgramJD); // Run initializers
auto *JPMain = (int(*)(int, char*[]))jit_dlsym(JPH, Main);
int JPResult = JPMain(JPArgC, JPArgV); // Run main
jit_dlclose(JPH); // Run deinitializers
```

llvm-jitlink demo 2…

ORC Core

ExecutorProcessControl

- An abstraction for the executor process
  - Reports the triple, page size, and other features to support JIT setup
  - Provides remote dlopen, symbol search operations for runtime bootstrap
  - Provides remote memory-management
  - Allows function calls between the JIT and executor processes
- These primitives enable communication between JITLink and the linked ORC runtime in the executor process

ORC Core

ExecutorProcessControl example

```
// Setup is easy:

auto EPC = JITInProcess
  ? SelfExecutorProcessControl::Create()
  : RemoteExecutorProcessControl::Connect(…);

auto J = LLJITBuilder(std::move(EPC)).create();

// EPC’s interface is small and the operations are simple.
// Writing EPC implementations for different IPC/RPC
// systems is easy.
```

ORC Core

Platform

- JITLink and the ORC runtime need to share object format specific information
  - Initializers, language runtimes, eh-frame, TLV, etc.
- We don’t want to pollute ORC Core with the details
- JITLink plugins provide a way to inspect format specific details (LinkGraph)
- ExecutorProcessControl allows functions to be called in each direction to communicate the information
- The new Platform type coordinates all this

ORC Core

Platform Example

- Usage:

```
ES.setPlatform(
  MachOPlatform::Create(ObjLinkingLayer, EPC,
                        PlatformJD, OrcRuntimePath)));
```

- MachOPlatform…
  - Installs MachO specific JITLink plugins
  - Adds runtime archive to PlatformJD
  - Provides entry-points for the runtime, e.g. getInitializerSequence for dlopen
- Arbitrary MachO programs now “Just Work”

ORC Runtime

Status

- MachO only, highly experimental
- ELF implementation should be relatively easy to derive from MachO
  - I hear ELF TLVs are more complicated
- No plans for COFF yet

ORC 2021

Recap

- JITLink and the ORC runtime move us towards full emulation of static compilation environment features
- JITLink ELF backends are getting easier to write
- ELF ORC Runtime and ELF Platform are waiting to be written
- Concurrent compilation works, performance will probably need tuning
- Speculative compilation and re-optimization have proof-of-concept implementations, but not in-tree helpers yet
- Lots of opportunities to get involved!

ORC 2021
Getting involved

- Check out…
  - The Building A JIT Tutorial Series
  - The ORCv2 and JITLink design documents
  - The LLVM ORCv2 examples — `llvm/examples/OrcV2Examples`
  - The test cases — `llvm/tests/ExecutionEngine`
  - The ORC Runtime — `compiler-rt/lib/orc`
- Discuss on llvm-dev and in `#jit` on the LLVM discord server

Extra Topics

Extra Topics

Concurrent Compilation

- Use ExecutionSession::setDispatchTask to enable concurrency
  - By default tasks are run immediately on the current thread
  - Move tasks to a thread pool or dispatch queue to enable concurrency
- Use ThreadSafeModule + ThreadSafeContext for LLVM IR
  - Modules sharing a ThreadSafeContext cannot be compiled concurrently
- Simplest and safest scheme is one LLVMContext per Module

Extra Topics

Laziness

- ORC is not lazy by default: it materializes symbols on *lookup*, not first call
- Introduce laziness (and break link-time dependencies) using stubs
  - Rename function `func` to `func$body`
  - Introduce stub `func` that uses ORC to look up the address of `func$body` on first call, then jumps to the body
  - Re-use lookup safety guarantee: stub safe to call on any thread at any time
- ORC has built-in support for this: the lazyReexport utility (see llvm/example/OrcV2Examples/LLJITWithLazyReexports)

Extra Topics

Laziness

- We still compile on whole-module boundaries
- Extract functions into their own modules to get per-function laziness
- Extraction can (and should) be done lazily
  - Use MaterializationResponsibility::getRequestedSymbols to identify requested symbols for extraction
  - Use MaterialiaztionResponsibility::replace to return the remainder of the module to the JITDylib without compiling it
- MaterializationResponsibility operations make extraction thread-safe

Extra Topics

Re-optimization

- To recompile hot functions we can…
  - Name function bodies for their optimization levels
    e.g. `func$body.00`, `func$body.03`
  - Use function names to construct optimization pipelines
  - Use stubs (or call-site rewriting) to redirect execution to new bodies
- Instrumentation to detect hot functions can be added to IR
- Doable with existing APIs, but no built-in support — good open project!

Extra Topics

Speculation

- Together, concurrency and laziness enable *speculative compilation*
  - Issue an early no-op lookups for functions that are likely to be needed later
  - Hides compile latency at the cost of some laziness
- Identify speculation opportunities with profiling, instrumentation, CFG analysis
- LLVM GSoC 2019 — Praveen Velliengiri wrote a proof of concept implementation for speculative compilation
  (See llvm/examples/SpeculativeJIT)

Extra Topics

Profile Guided Optimization in ORC — Educated guesses

- For instrumentation-based PGO, collecting profiles may require extra work:
  - Static linker aggregates sections (e.g. __llvm_prf_data) across all objects
  - Use JITLink plugins to identify per-object sections
  - Use ORC runtime to aggregate identified sections before processing
- Sampling-based PGO not investigated, may “just work”?
- Profiles can be used to guide standard optimizations
- Profiles might also be useful for constructing traces — another open project