Powering Xcode Previews with LLVM's JIT

Lang Hames

Dynamic Development Workflows

LLVM JIT's Role

- Reuse existing compiler pipelines and compiled libraries in dynamic contexts
  - LLDB expression evaluation, Cling / `clang-repl` (interactive C++), CppInterOp (Python + C++)[1], Jank (Clojure + C++), Clasp (LISP + C++)[2], …
    - All know when they're targeting the LLVM JIT
- What about programs that weren't intended to be run under LLVM's JIT?
  - i.e. Projects with regular build systems targeting static compilers
  - Can run them under the JIT and preserve behavior? Can the JIT scale?
  - What could we build with this?

[1] Enabling Interactive C++ in Clang, youtu.be/33ncblQoa4c

[2] Implementing Common Lisp with LLVM, youtu.be/mbdXeRBbgDM

ORC — On Request Compilation

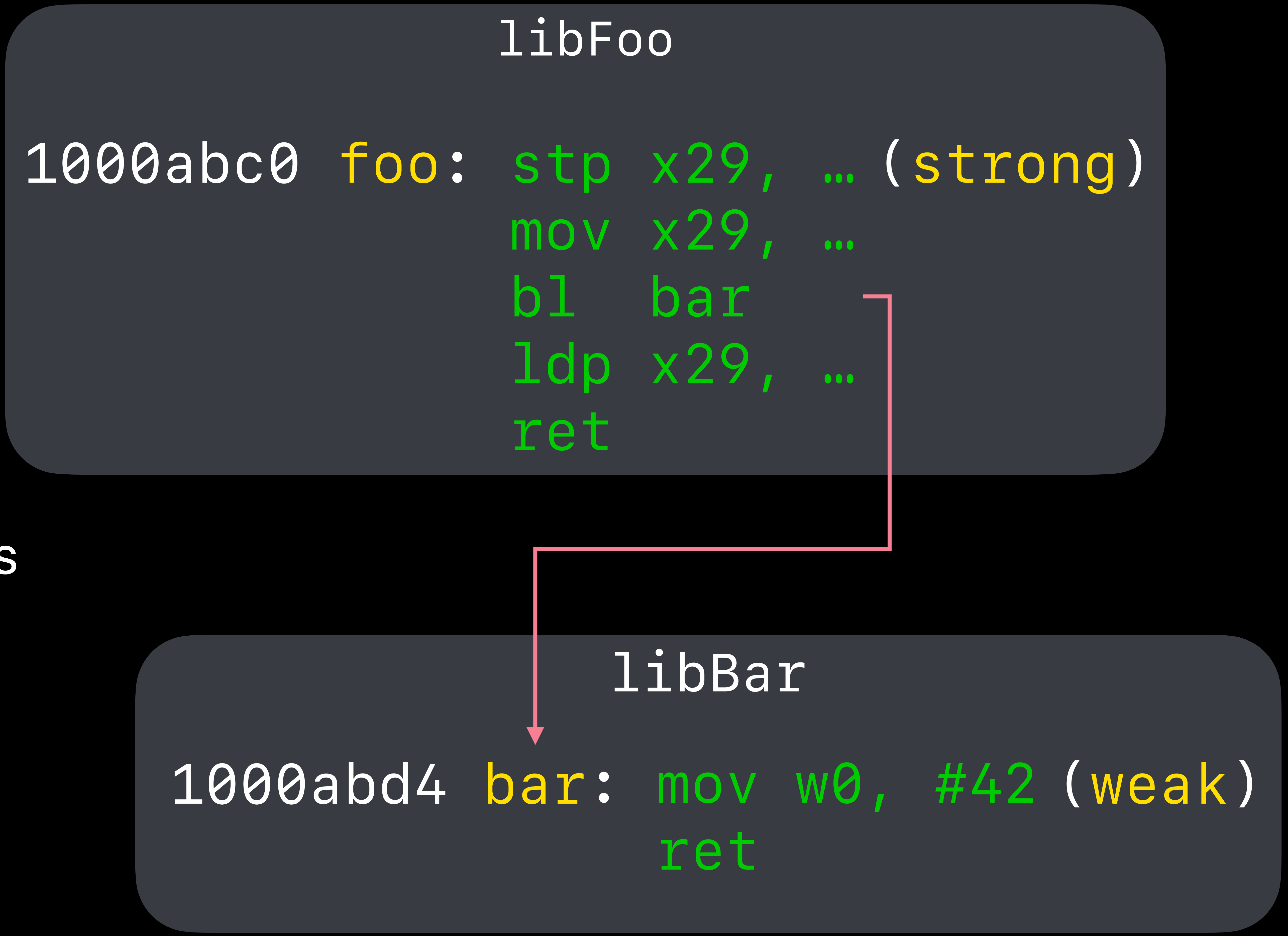
LLVM’s JIT APIs

- A foundation for concurrent, heterogeneous, cross-process/architecture/OS JITing
  - Reuse static compilers in a dynamic context
  - Provide access to LLVM optimizations to existing JITs
  - Mix-and-match — build one JIT’d program using multiple compilers / languages
  - Compilers don’t need to coordinate with one another
- Three components: a coordination layer, a just-in-time linker, and a runtime

Component 1: ORC Core

Coordination — Common Language

- Symbols have…
  - Addresses (once *resolved*)
  - Content (once *emitted*)
  - Dependencies on one another
  - Containers (JITDylibs), linker attributes
- ORC Core is agnostic with regards to…
  which address space,
  how content is produced,
  how dependencies identified



```
                     libFoo

1000abc0 foo: stp x29, …(strong)
              mov x29, …
              bl  bar
              ldp x29, …
              ret
```

```
                     libBar

1000abd4 bar: mov w0, #42(weak)
              ret
```

Component 1: ORC Core

Coordination — APIs

- `MaterializationUnit`s encapsulate the process of producing symbols
  - Declare an interface (a set of symbols and linkages), added to JITDylibs
  - Triggered on first `lookup` of any symbol within them
  - `resolve` callback maps symbols to addresses (content needn't be written yet)
  - `emit` callback notifies ORC that content has been written, what symbol dependencies are
- Lookup may be issued on any thread at any time for any set of symbols
  - Dependence info ensures that reachable symbols are emitted before lookup returns

Component 2: JITLink

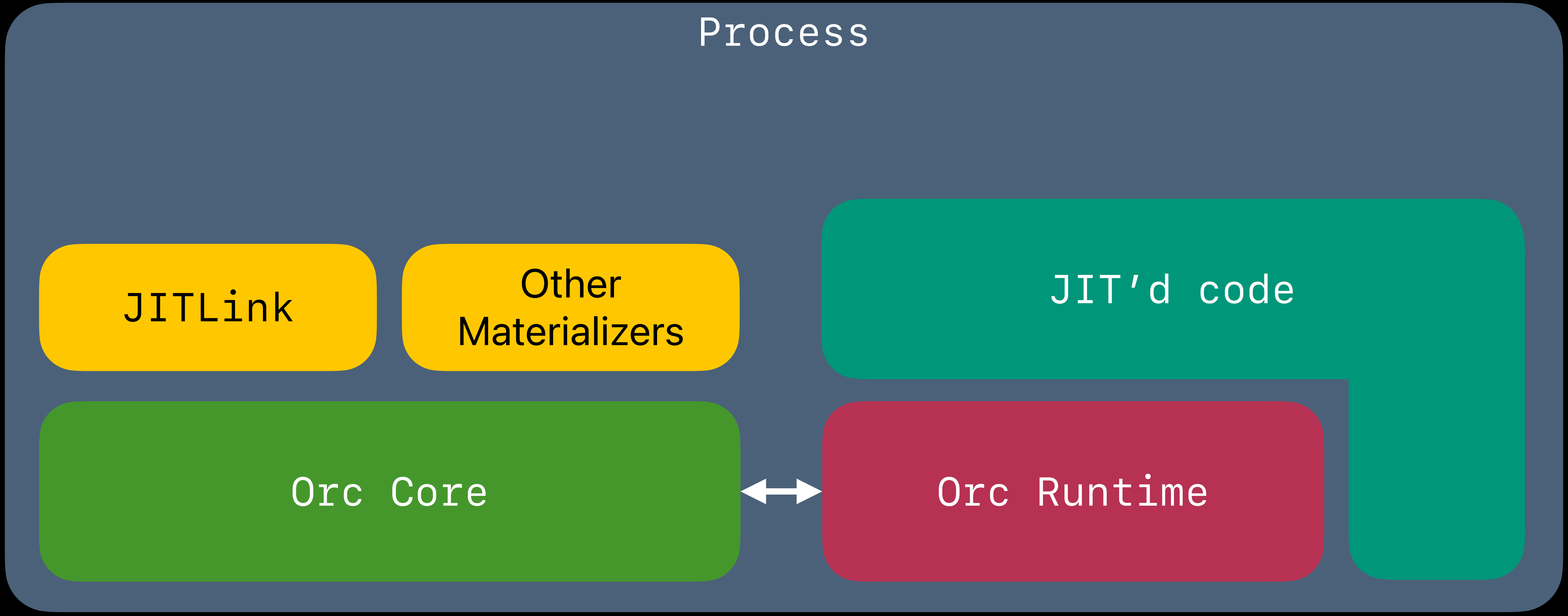
- JITLink links object files into JIT'd memory (it's an object file *materializer*)
  - Trivially JIT LLVM IR: `LLVM IR → CodeGen → Object → JITLink → JIT'd memory`
  - Or: `YourLanguage → YourCompiler → Object → JITLink → JIT'd memory`
  - Precompiled object files, archives can just be loaded directly
- Customizable memory manager controls how linked code is transferred to JIT'd memory
- Plugin interface allows customization of the link process

Component 3: ORC runtime

- Lives in the executing process with JIT’d code
- Supports advanced features
  - Initializers, exceptions, thread locals, …
  - POSIX API emulation: dlopen, dlsym, …
- Supports calls (including via IPC/RPC) from JIT’d code to ORC support functions

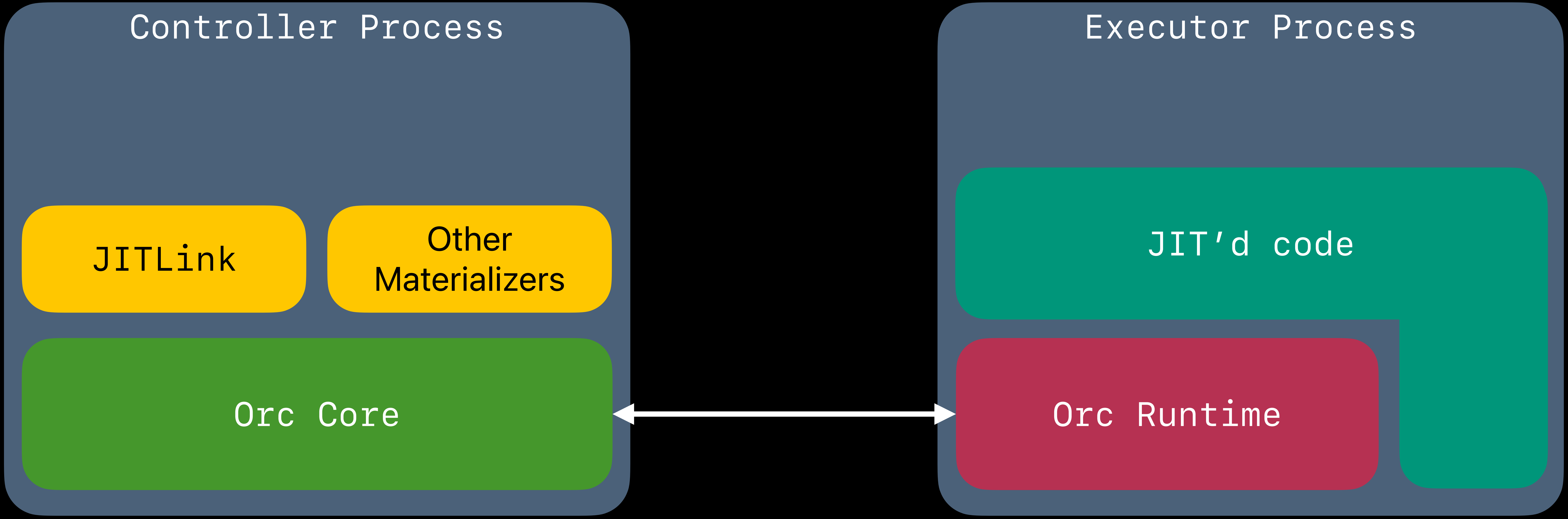
ORC Components

Core, JITLink, and the Orc Runtime



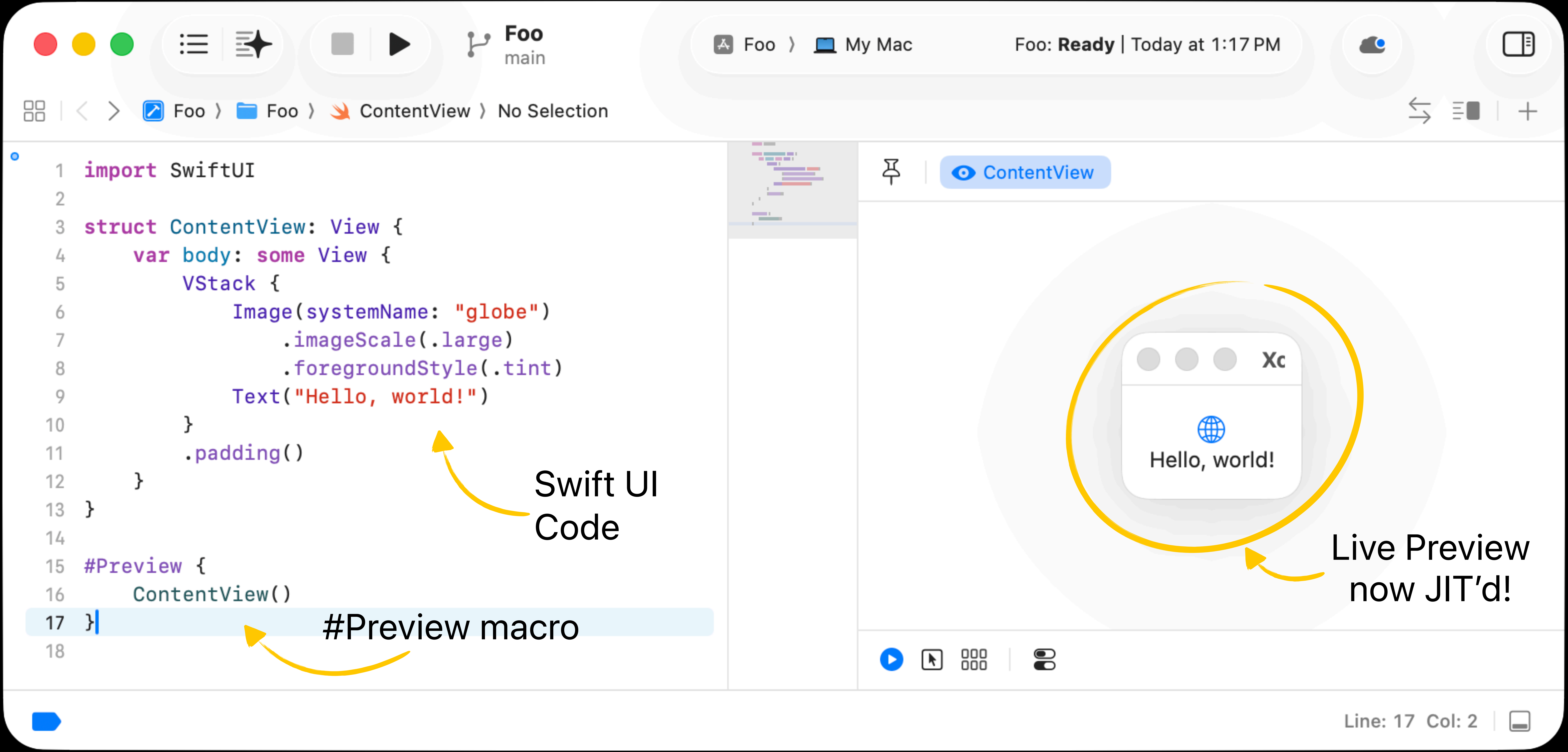
ORC Components

Core, JITLink, and the Orc Runtime



See the ORCv2 Deep Dive talk for more details — youtu.be/i-inxFudrgI

Xcode Previews



Foo
main
Foo 〉 My Mac
Foo: **Ready** | Today at 1:17 PM
Foo 〉 Foo 〉 ContentView 〉 No Selection
ContentView
Xc
Hello, world!
Line: 17 Col: 2

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundStyle(.tint)
            Text("Hello, world!")
        }
        .padding()
    }
}

#Preview {
    ContentView()
}
```

Swift UI Code

#Preview macro

Live Preview now JIT'd!

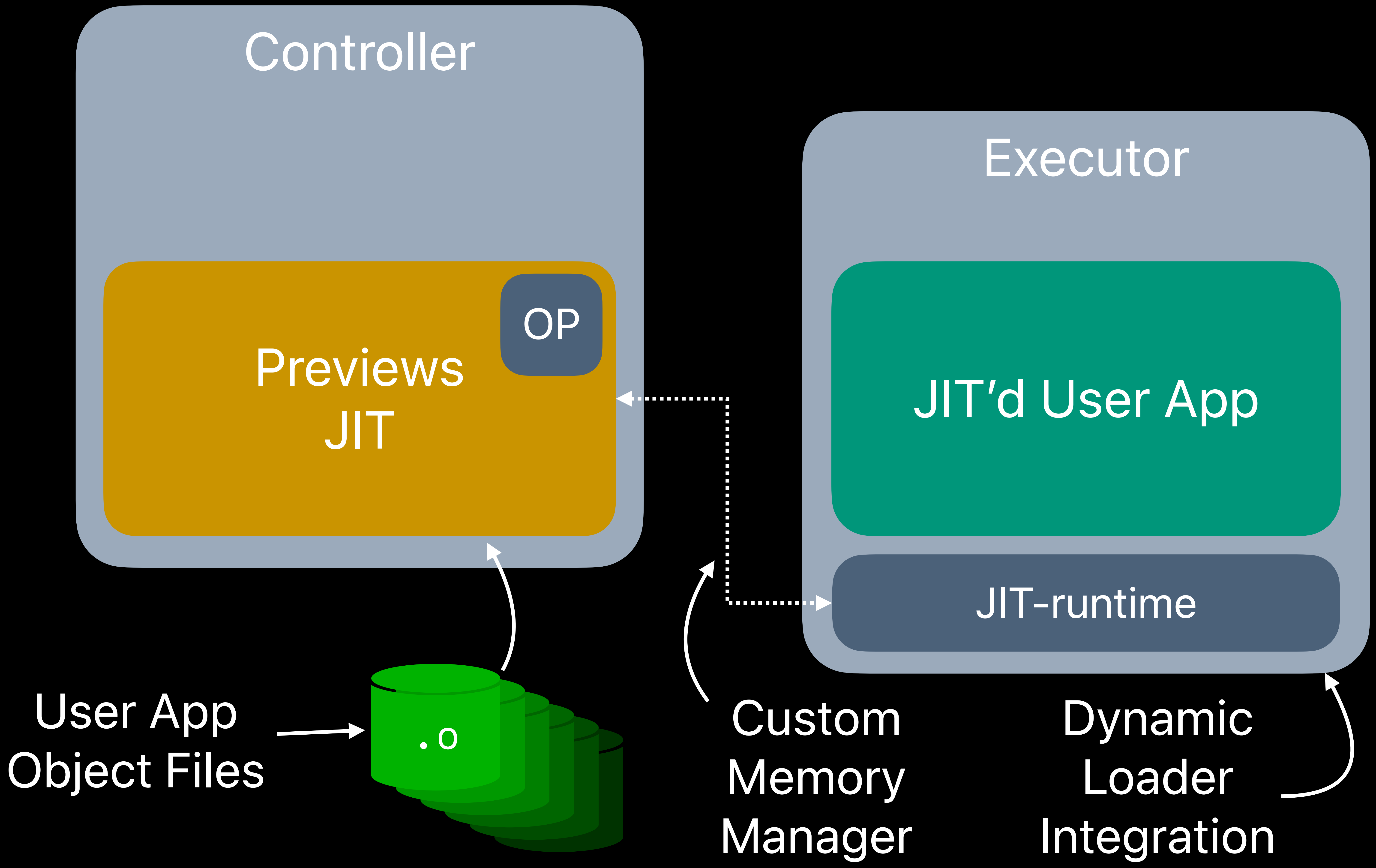
SwiftUI Programs

- Can get **large**…
  - Thousands of files
  - Hundreds of megabytes code and data
  - Hundreds of thousands of relocations
  - Often split into multiple frameworks / dynamic libraries

- Can get *weird*…
  - Mixed languages, static archives
  - Multiple *slices* (`arm64`, `x86-64`…)
  - Entitlements (e.g. hardened runtime)
  - Interesting linker options (`-r`, …)
  - Interesting assembly options (no `.subsections_via_symbols`)

Xcode Previews JIT Setup

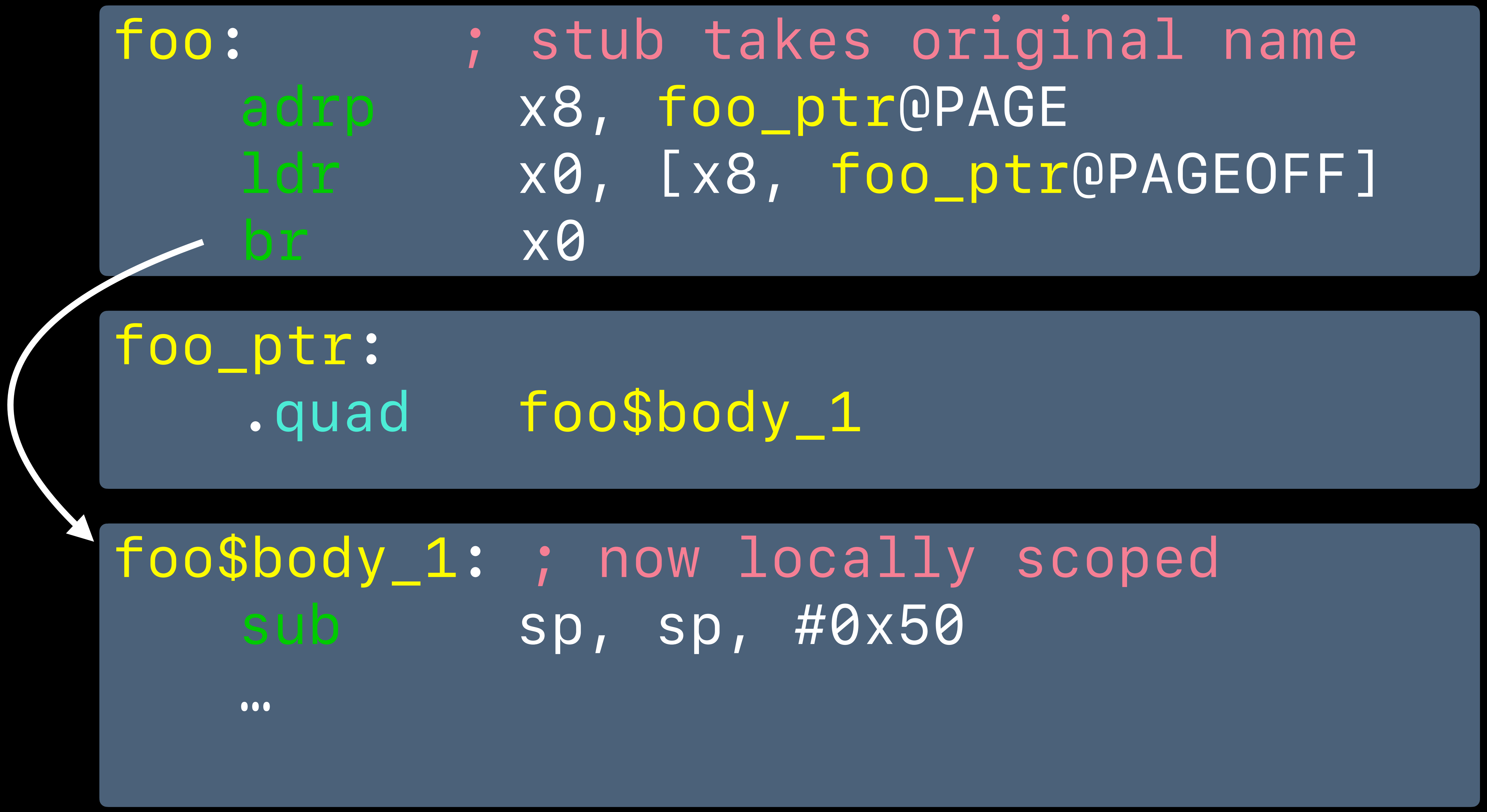
- Cross process
- Object files in (no laziness)
    - Override Plugin (OP) applied
- Custom memory management
- Dynamic Loader Integration presents JIT'd code as-if statically linked
- Concurrency for performance



Override Plugin

Fast Function Body Replacement

- JITLink APIs can rename, add code / data
- On first definition
  - Rename function, set scope to local
  - Introduce stub with original name, stub pointer pointed at original body



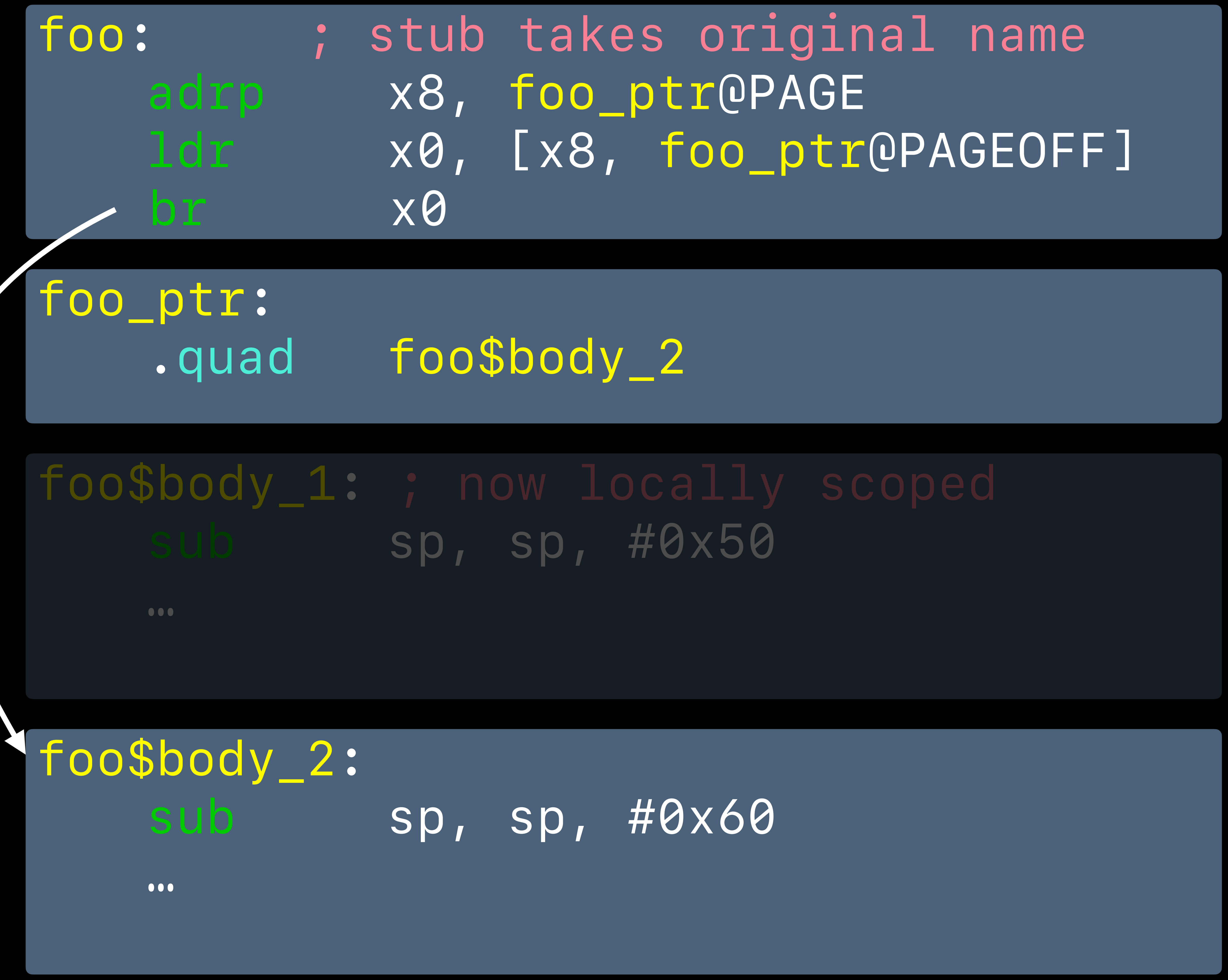
```
foo:          ; stub takes original name
    adrp     x8, foo_ptr@PAGE
    ldr      x0, [x8, foo_ptr@PAGEOFF]
    br       x0
```

```
foo_ptr:
    .quad    foo$body_1
```

```
foo$body_1: ; now locally scoped
    sub      sp, sp, #0x50
    …
```

Function Body Overrides

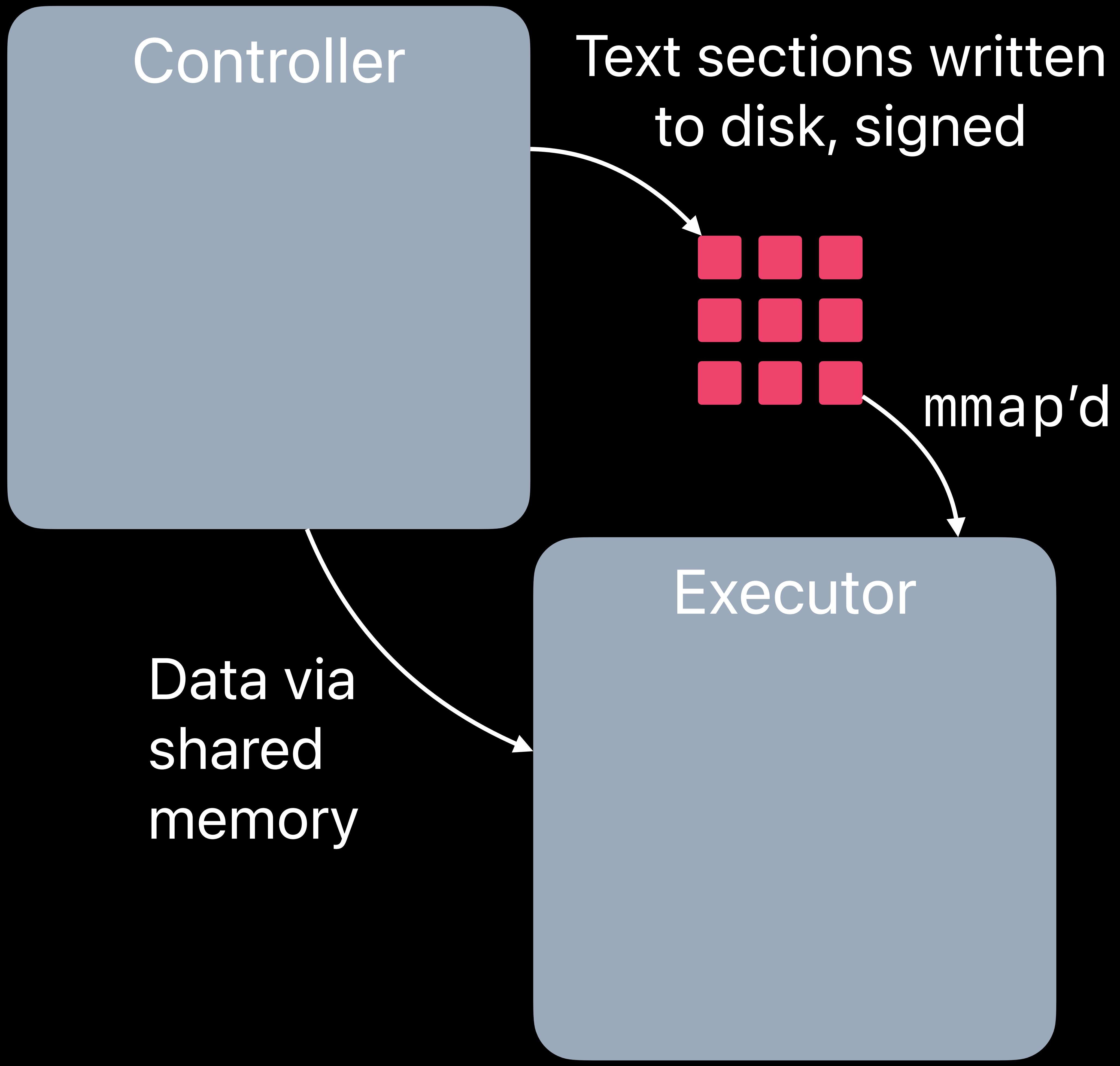
- JITLink APIs can rename, add code / data
- On first definition
  - Rename function, set scope to local
  - Introduce stub with original name, stub pointer pointed at original body
- On subsequent definition
  - Rename function, set scope to local
  - Update pointer to point at new body



```
foo:     ; stub takes original name
    adrp    x8, foo_ptr@PAGE
    ldr     x0, [x8, foo_ptr@PAGEOFF]
    br      x0

foo_ptr:
    .quad   foo$body_2

foo$body_1: ; now locally scoped
    sub     sp, sp, #0x50
    …

foo$body_2:
    sub     sp, sp, #0x60
    …
```

Custom Memory Management

- Applies code-signing to JIT'd code
  - Used to ensure that we don't affect the behavior of *Hardened Runtime* apps
  - Uses a custom *Preview* signature type
    Only usable for apps in development mode
- Data transported via shared memory
- Code is written to disk, signed, and then mmap'd in the executing process



Dynamic Loader Integration

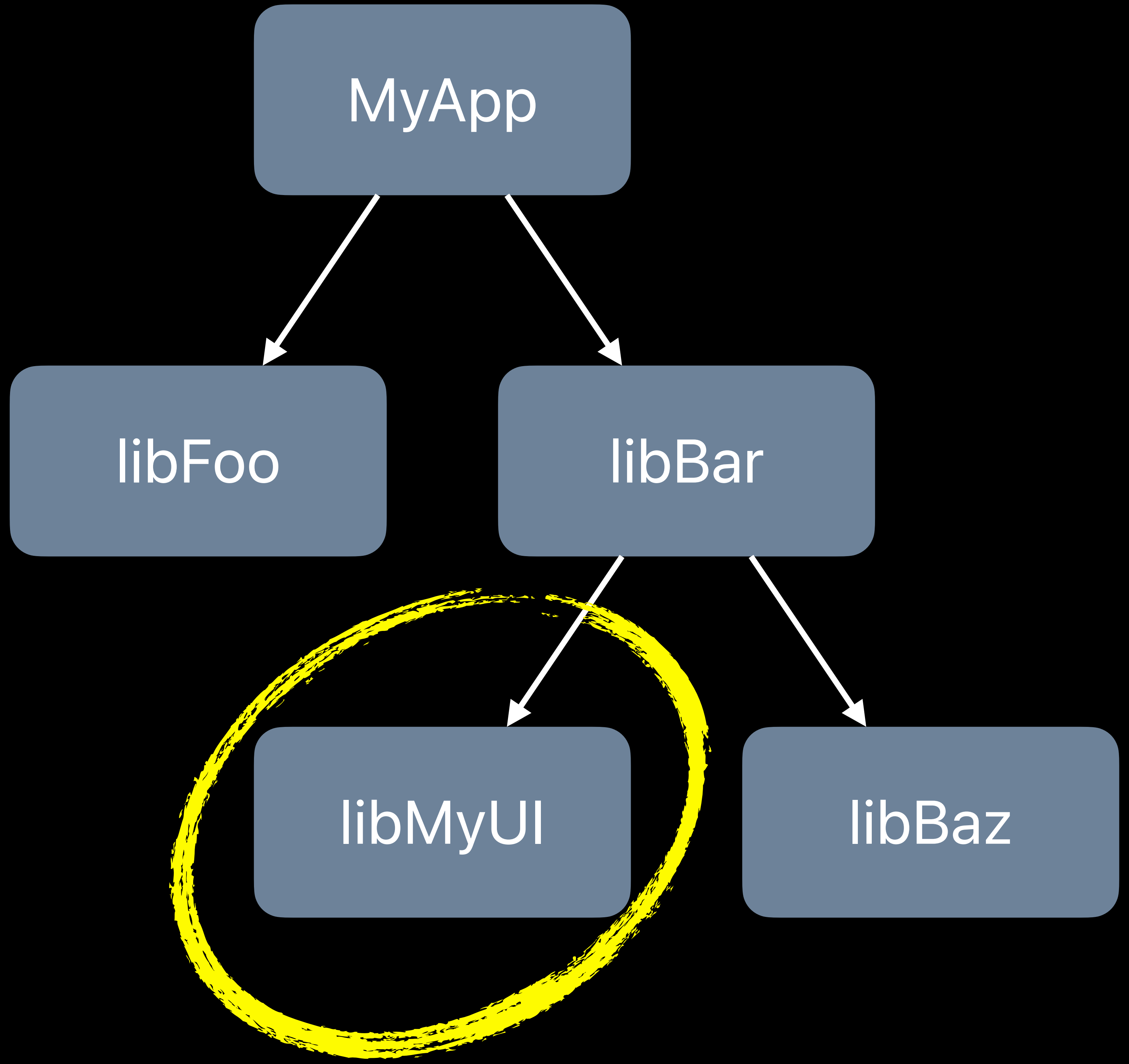
Overview

- What should `dlsym("foo")` return if `foo` is JIT’d?
  - Answer: `&foo` (same as-if precompiled)
  - But: call to `dlsym` might be in precompiled code — we’ll need the dynamic loader’s help
- Teach the dynamic loader to treat JIT’d code as if it were regular dylibs…
  - … A “pseudo-dylib” defined by callbacks rather than a file (callbacks implemented by our JIT)
  - POSIX APIs like `dlsym` naturally supported
  - Precompiled code can bind against JIT’d code — selectively JIT individual dynamic libraries

Dynamic Loader Integration

A Performance Opportunity

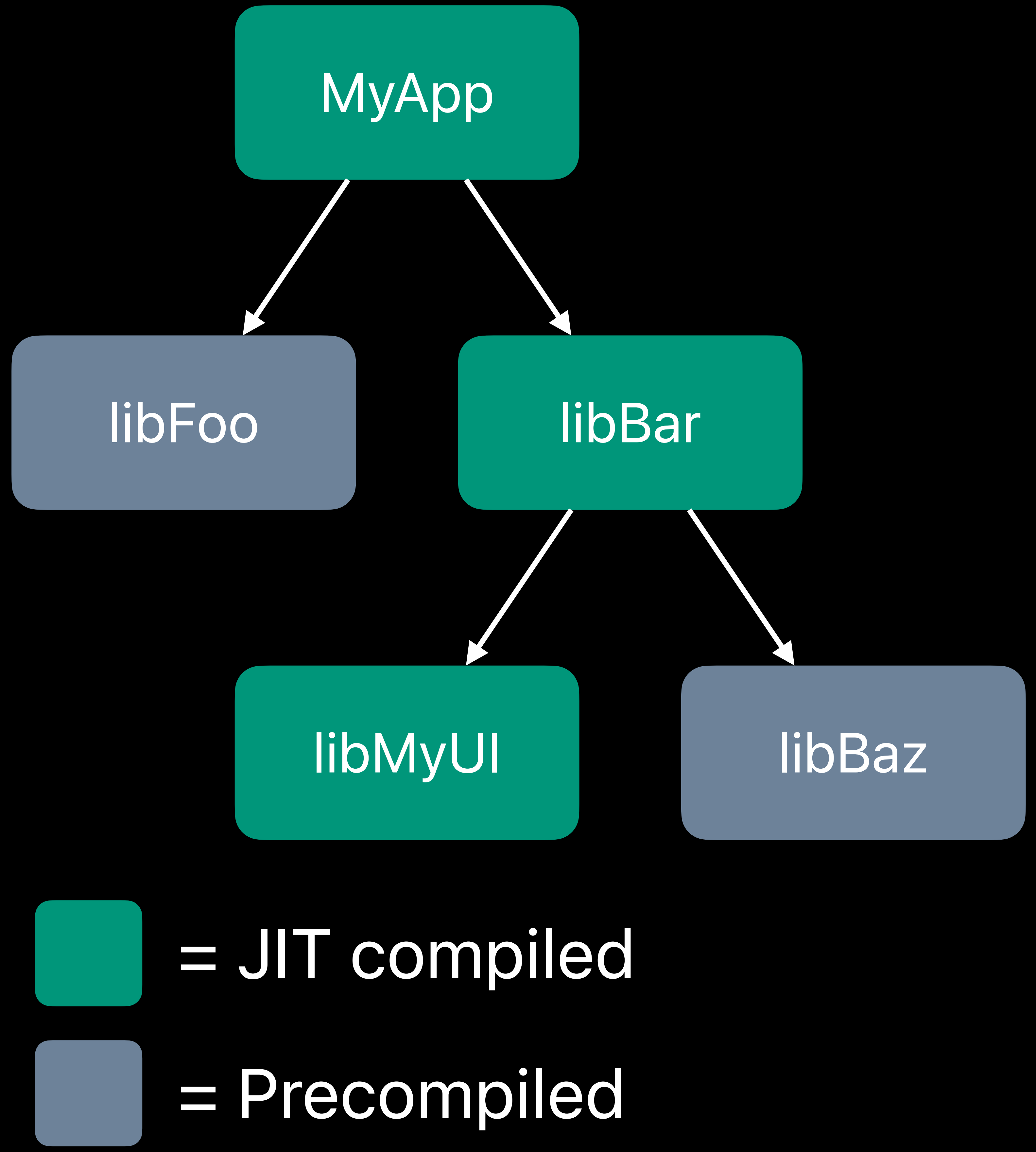
Say that we want to edit UI code in `libMyUI`…



Dynamic Loader Integration

A Performance Opportunity

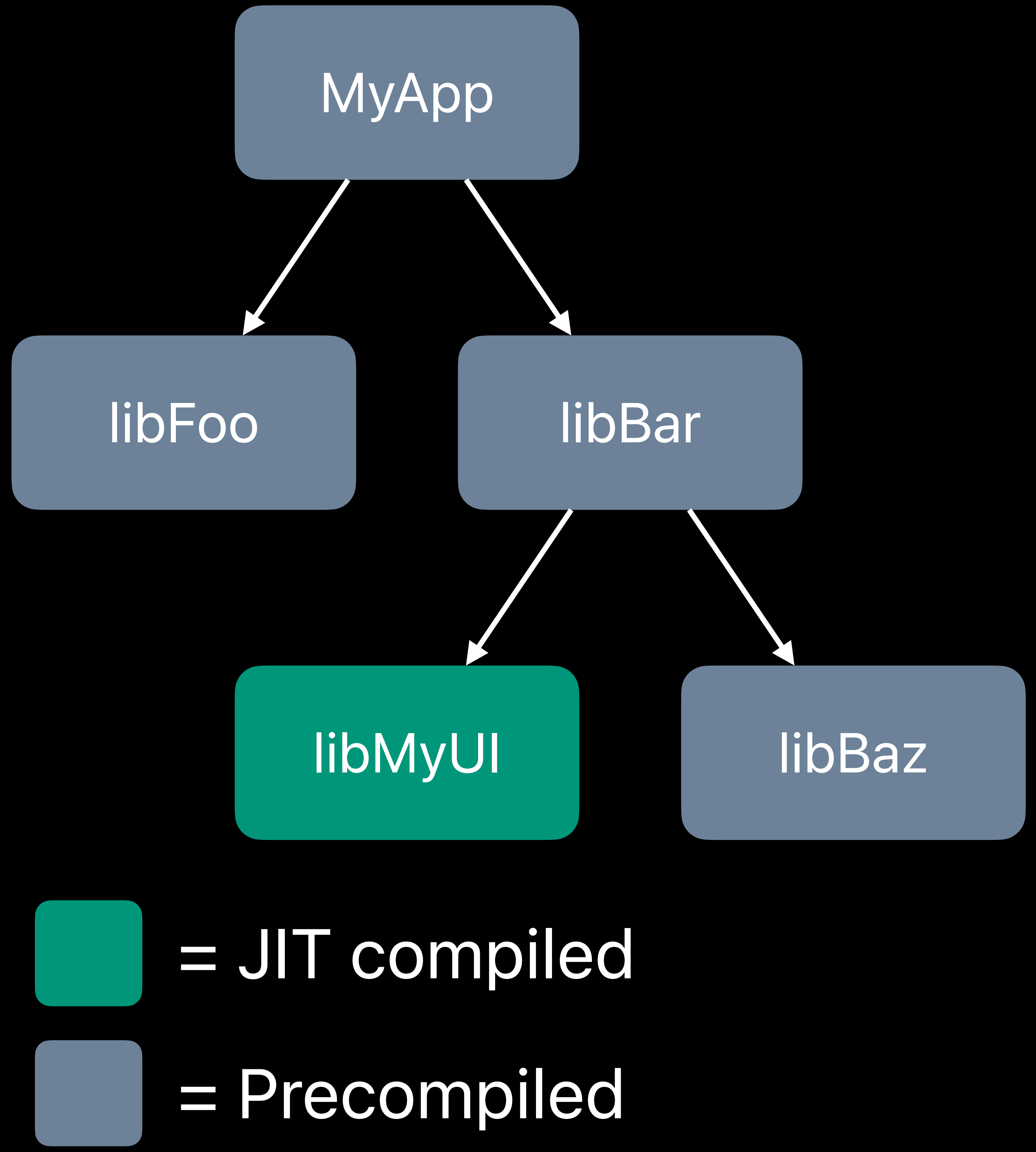
- JIT'd code not visible to precompiled?
  - Must JIT back to the root



Dynamic Loader Integration

A Performance Opportunity

- JIT'd code not visible to precompiled?
  - Must JIT back to the root
- JIT'd code visible to precompiled?
  - JIT only what you want
  - JIT what is changing
    Statically link the rest
    The best of both worlds
- Use TextAPI to statically link against JIT'd libraries[1]



[1] Using TAPI to Understand APIs & Speed Up Builds, youtu.be/B9li6EkD5zA

Dynamic Loader Integration

Dynamic Library Operations

- *Create* a new library at a path
- *Load* library at path — runs initializers
- *Look up* symbols or addresses (binding code, `dlsym`, `dladdr`)
- *Close* a library — runs deinitializers
- *Delete* a library — no longer openable

Dynamic Loader Integration

Dynamic Library Operations via callbacks

- Create → *Register* — takes callbacks, address range, and a “loadable-at-path” predicate
    - Load → *Initialize* — runs initializers
    - Lookup → *Lookup* — lookup symbols (forwarded to ORC Core lookup in our implementation)
        - Symbols needn’t exist until they’re looked up — pseudo-dylibs can be populated lazily
            - Using ORC lazy-reexports you could defer function body compilation until first call
    - Close → *Deinitialize* — runs deinitializers
- Delete → *Deregister* — no longer openable

Dynamic Loader Integration

ORC Runtime Implementation

- The ORC runtime already implemented similar operations (for emulated `dlopen`, `dlsym`, etc.)
- For Xcode Previews we…
  - Added glue code to align interfaces
  - Modified the internals to add caching, adapt to dyld’s locking scheme
  - Added auto-registration / de-registration when JITDylibs are created / destroyed

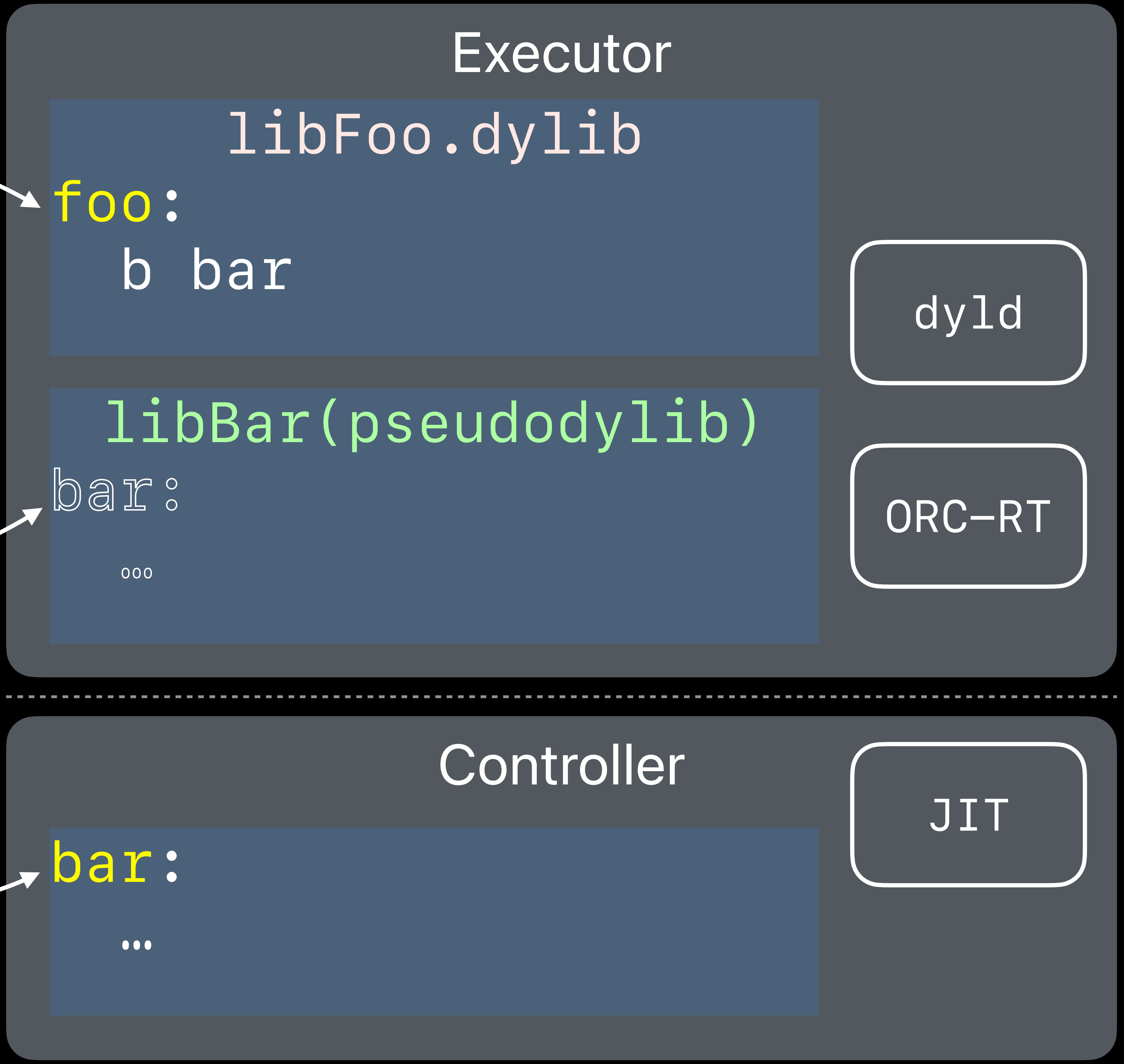
Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

Loading precompiled `libFoo.dylib` reference to `bar` needs to be bound

`bar` is in pseudo-dylib `libBar`, has not been linked yet

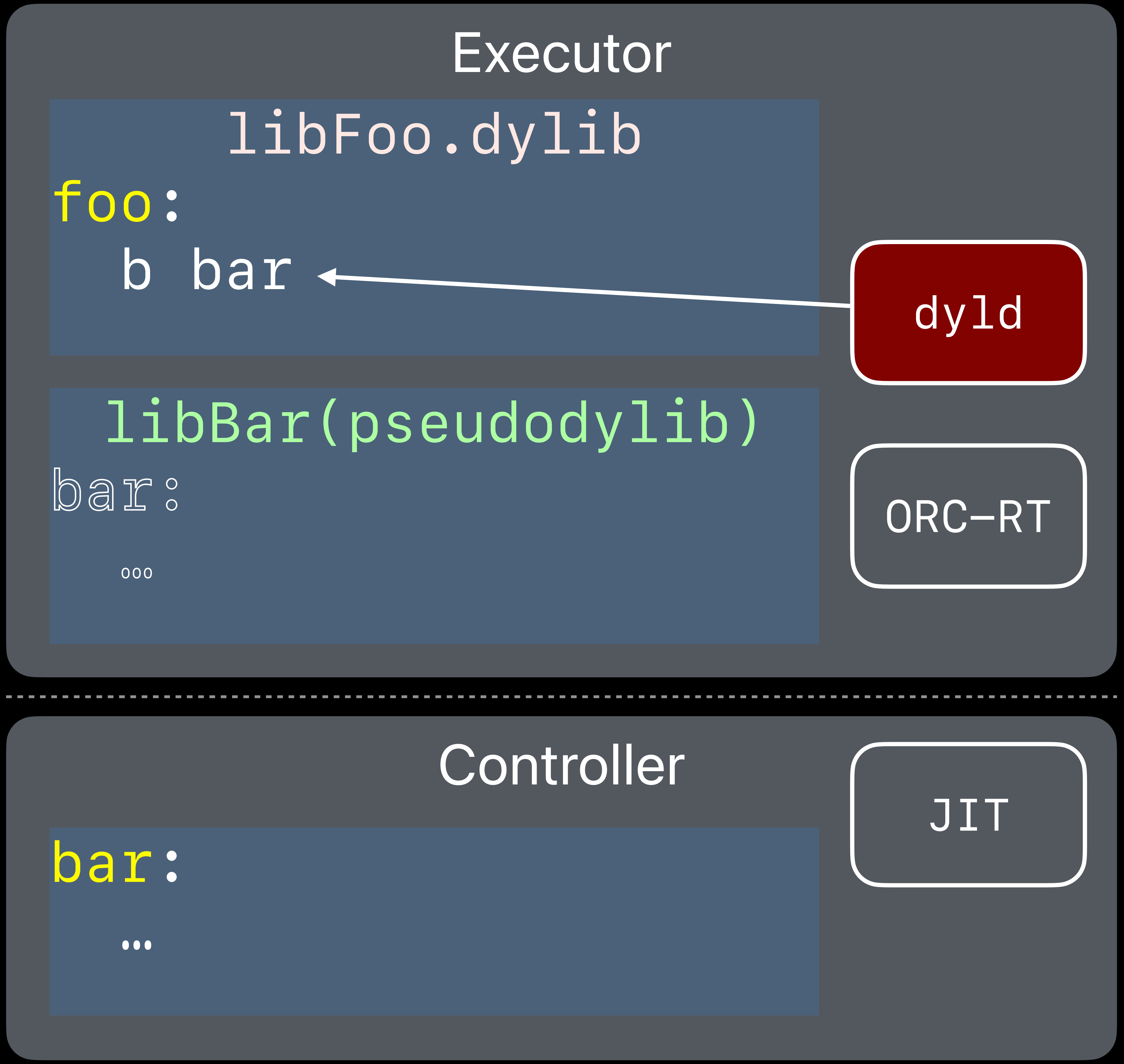
`bar` defined in relocatable object file registered with the Previews JIT



Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

1. dyld encounters bind("bar") operation

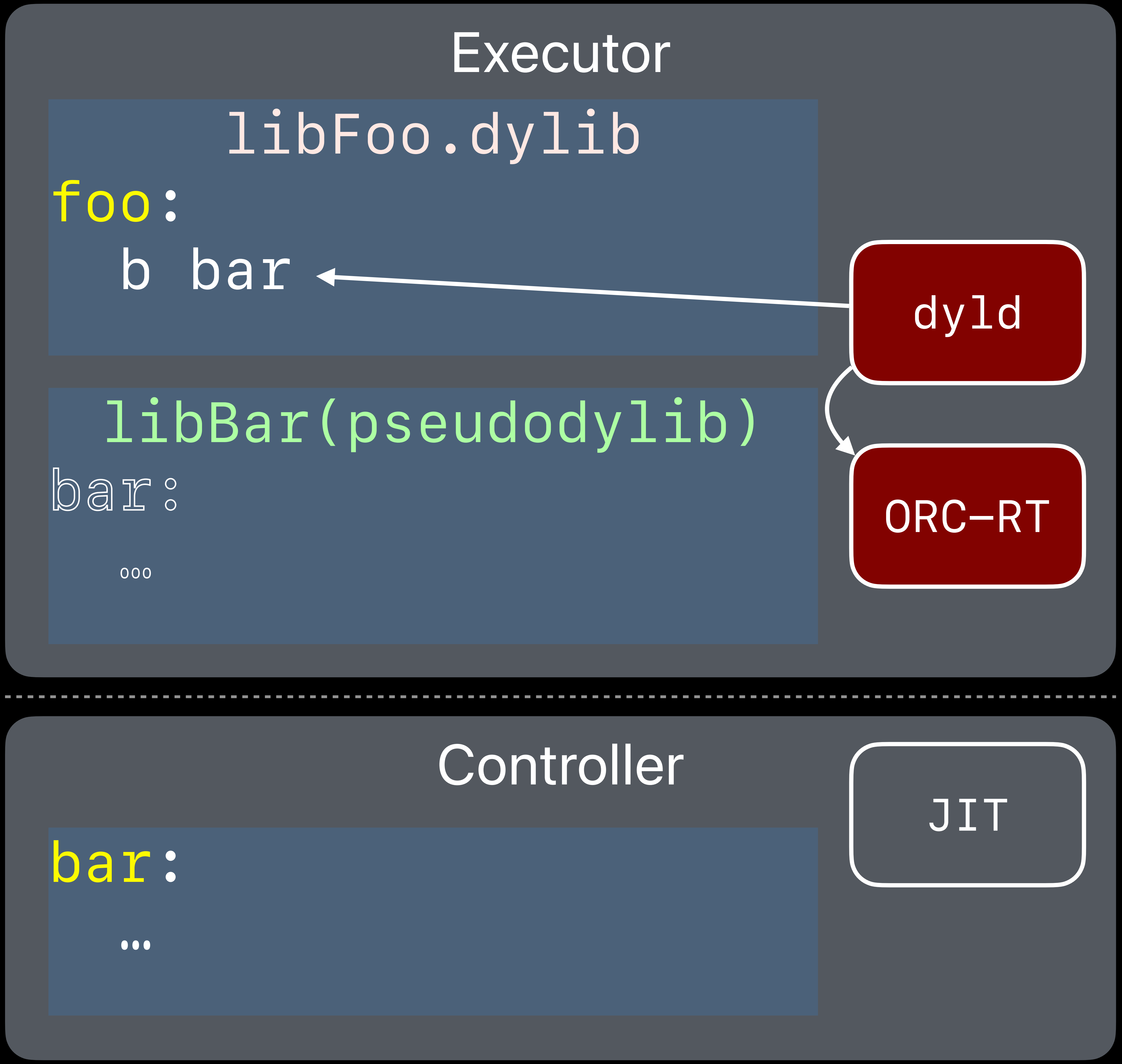


Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

1. `dyld` encounters `bind("bar")` operation

2. Calls `lookup` (implemented by ORC-RT)



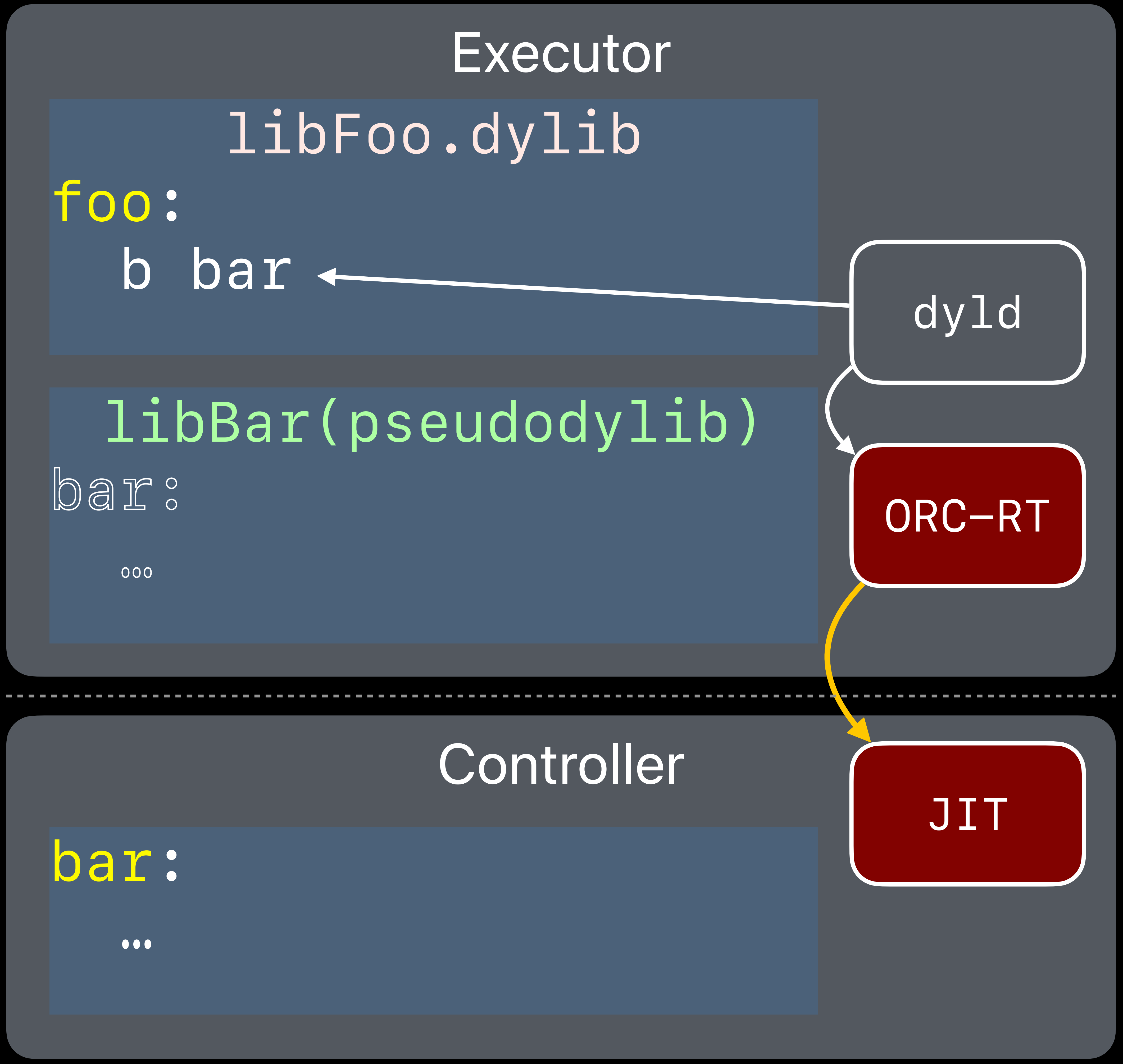
Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

1. dyld encounters bind("bar") operation

2. Calls lookup (implemented by ORC-RT)

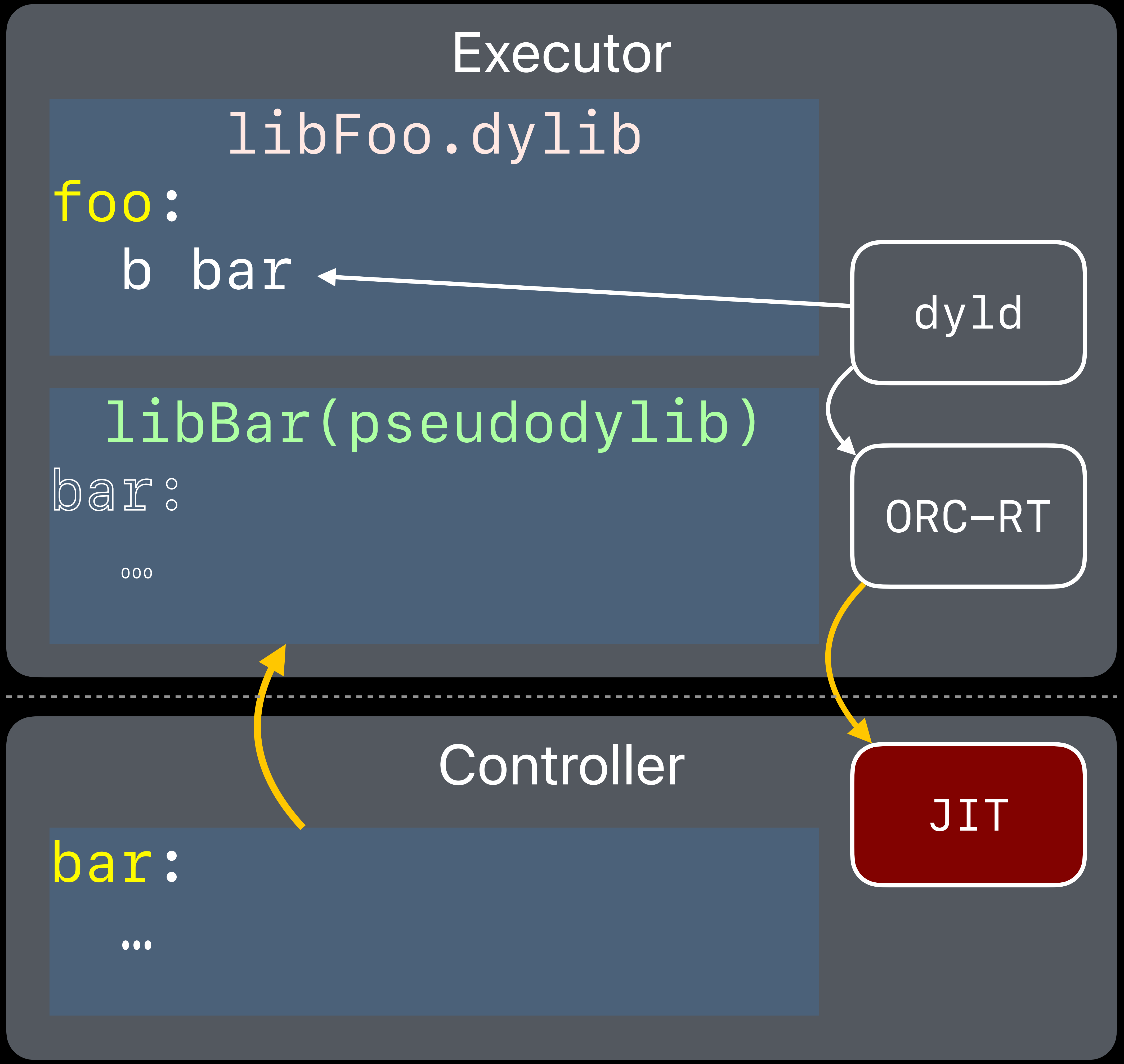
3. ORC-RT forwards (via IPC) to ORC lookup



Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

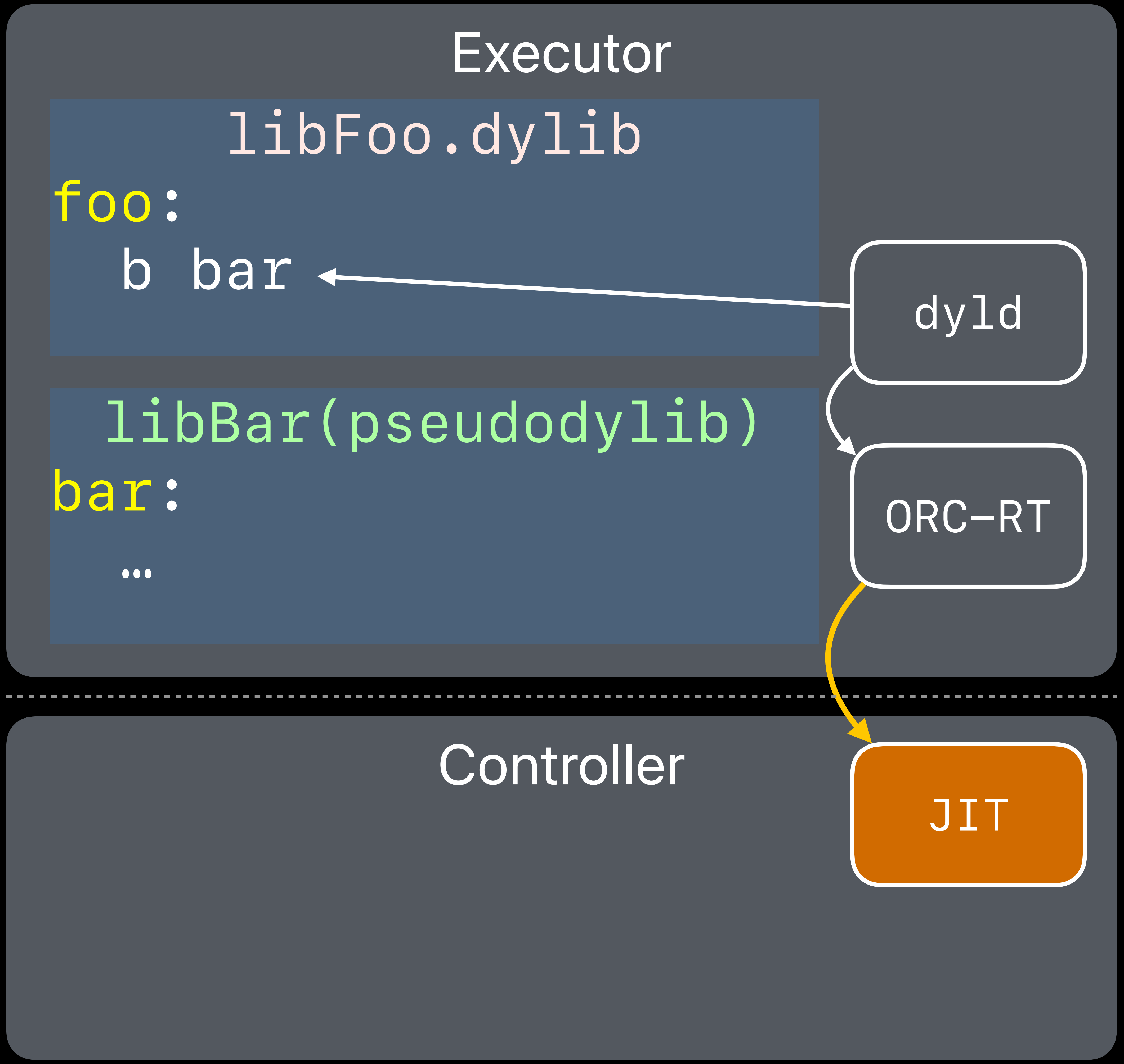
1. dyld encounters bind("bar") operation
2. Calls lookup (implemented by ORC-RT)
3. ORC-RT forwards (via IPC) to ORC lookup
4. ORC lookup triggers linking of bar



Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

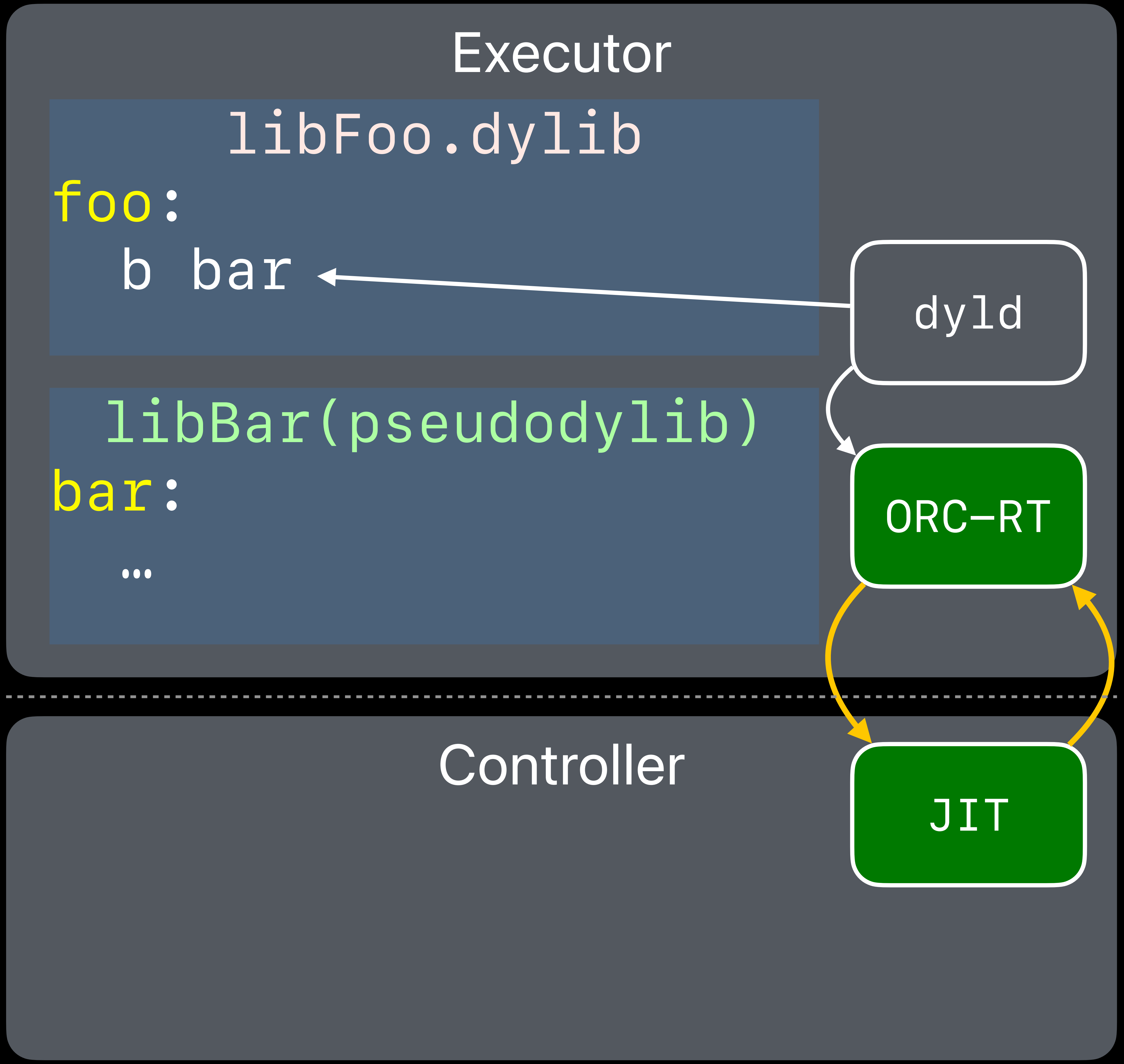
1. `dyld` encounters `bind("bar")` operation
2. Calls `lookup` (implemented by ORC-RT)
3. ORC-RT forwards (via IPC) to ORC lookup
4. ORC lookup triggers linking of `bar`



Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

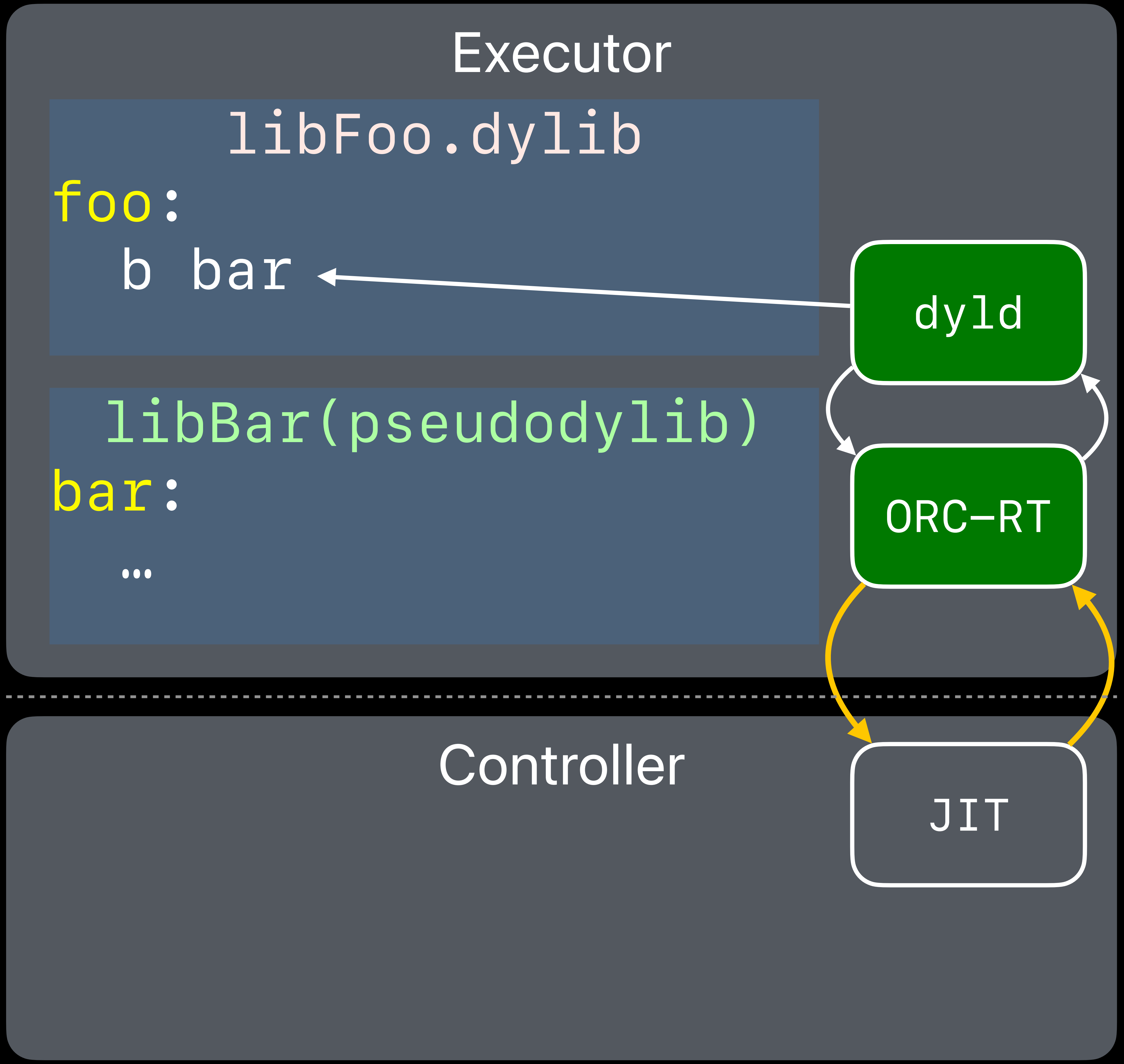
1. `dyld` encounters `bind("bar")` operation
2. Calls `lookup` (implemented by ORC-RT)
3. ORC-RT forwards (via IPC) to ORC lookup
4. ORC lookup triggers linking of `bar`
5. ORC returns address of `bar` to ORC runtime



Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

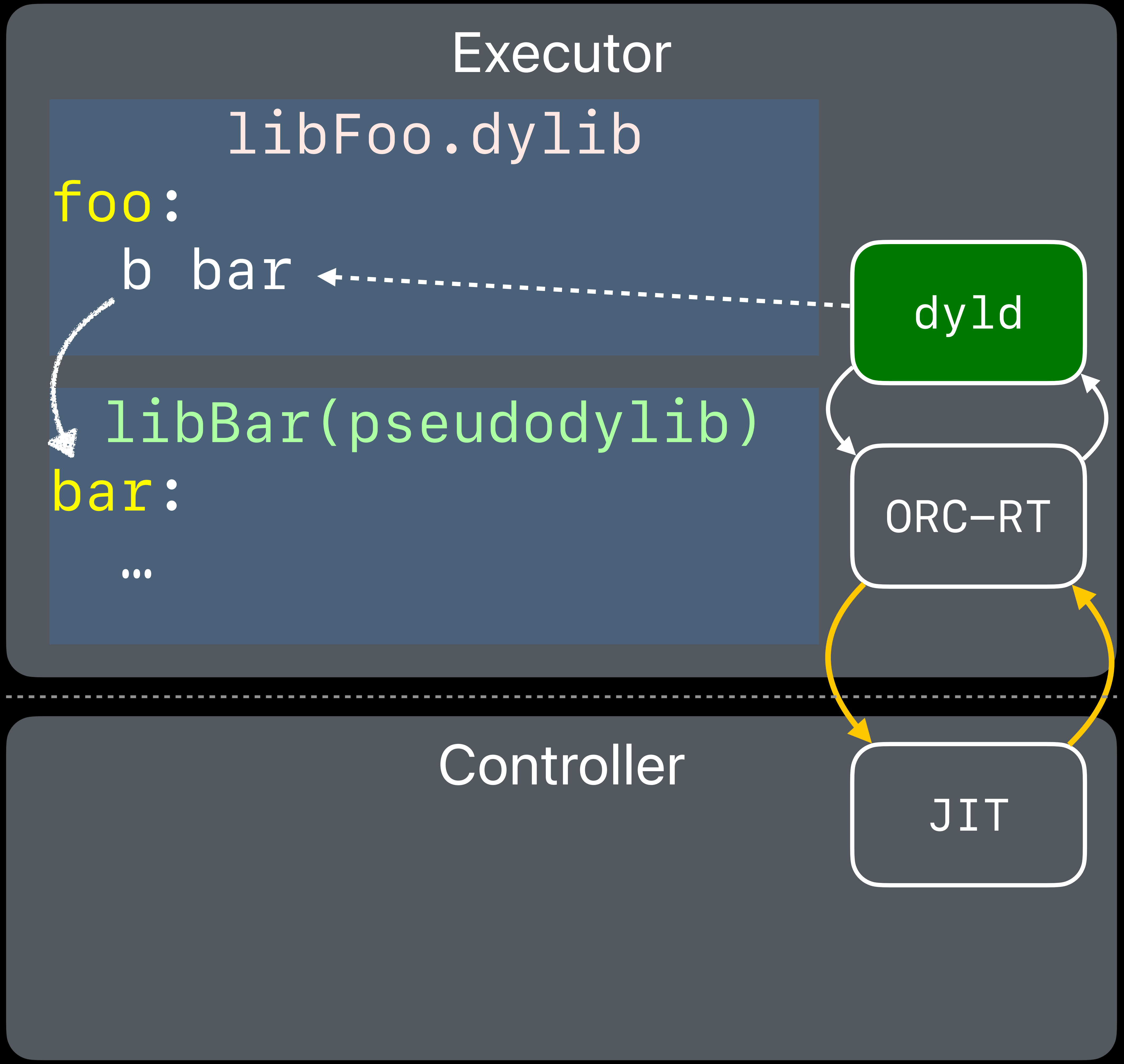
1. `dyld` encounters `bind("bar")` operation
2. Calls `lookup` (implemented by ORC-RT)
3. ORC-RT forwards (via IPC) to ORC lookup
4. ORC lookup triggers linking of `bar`
5. ORC returns address of `bar` to ORC runtime
6. ORC runtime returns address of `bar` to `dyld`



Dynamic Loader Integration

Binding JIT'd Symbols from Precompiled Code

1. `dyld` encounters `bind("bar")` operation
2. Calls `lookup` (implemented by ORC-RT)
3. ORC-RT forwards (via IPC) to ORC lookup
4. ORC lookup triggers linking of `bar`
5. ORC returns address of `bar` to ORC runtime
6. ORC runtime returns address of `bar` to `dyld`
7. `dyld` binds call to `bar`



Dynamic Loader Integration

Practical Impact and Challenges

- Best of both worlds — JIT changing code, statically link the rest
    - Substantial performance win on some previews
- ORC Runtime caches addresses to minimize IPC (this IPC is once-per-object-file-linked)
- Works, but adds IPC in the middle of dlopen 🌶️
    - May trigger IPC call *back into dyld on a different thread* (e.g. to resolve externals in `bar`)
    - Recursion could be avoided by adding something like “bind” operations to the JIT
      (these would be returned to the executing app, triggering lookup on the dlopen thread)

Performance

Performance

The Easy Stuff

- Turn on concurrency
  - Easy to do since ORC was designed for concurrency
  - Found and fixed some race conditions, especially in `MachOPlatform`
  - `DynamicThreadPoolTaskDispatcher` — N materializers, unbound # request handlers
- Improvements to many utility functions
  - E.g. `LinkGraph::splitBlock` was O(n^2) for repeated applications, now O(nlogn)
- Biggest changes were to dependence tracking…

WaitingOnGraph

`WaitingOnGraph`

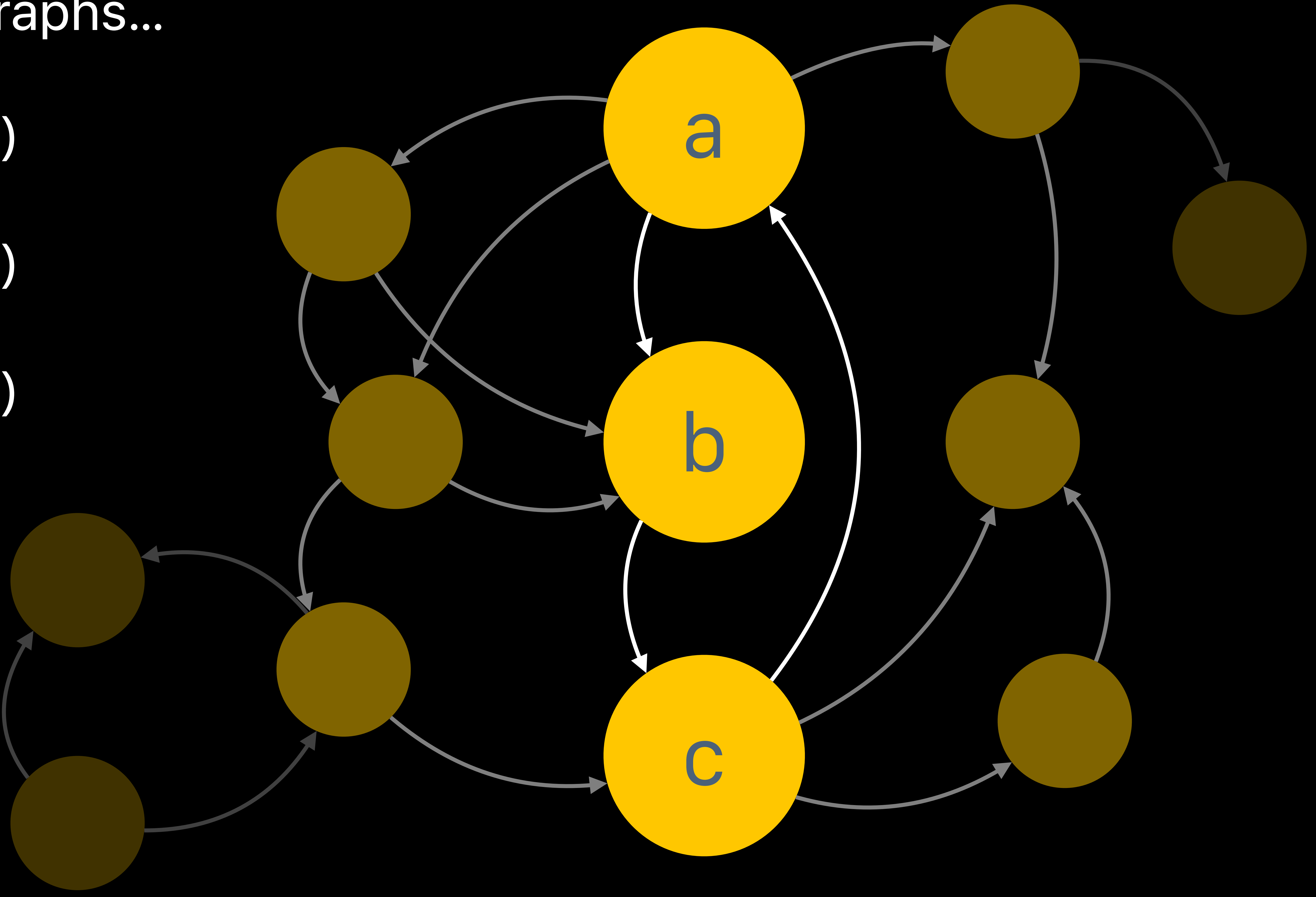
What is it? What *was* it?

- Enables lookup safety guarantee by tracking which symbols each symbol is *waiting on*
  - Better than tracking dependencies: waiting on relationships are *transient*:
    Graph scales with the size of outstanding work, not the size of the program
- Was…
  - Embedded within (and across) `JITDylib` objects (ORC's symbol tables)
    - Not unit testable
    - Not profilable
  - An arbitrary directed graph, due to `add–dependencies` …

WaitingOnGraph

add-dependencies

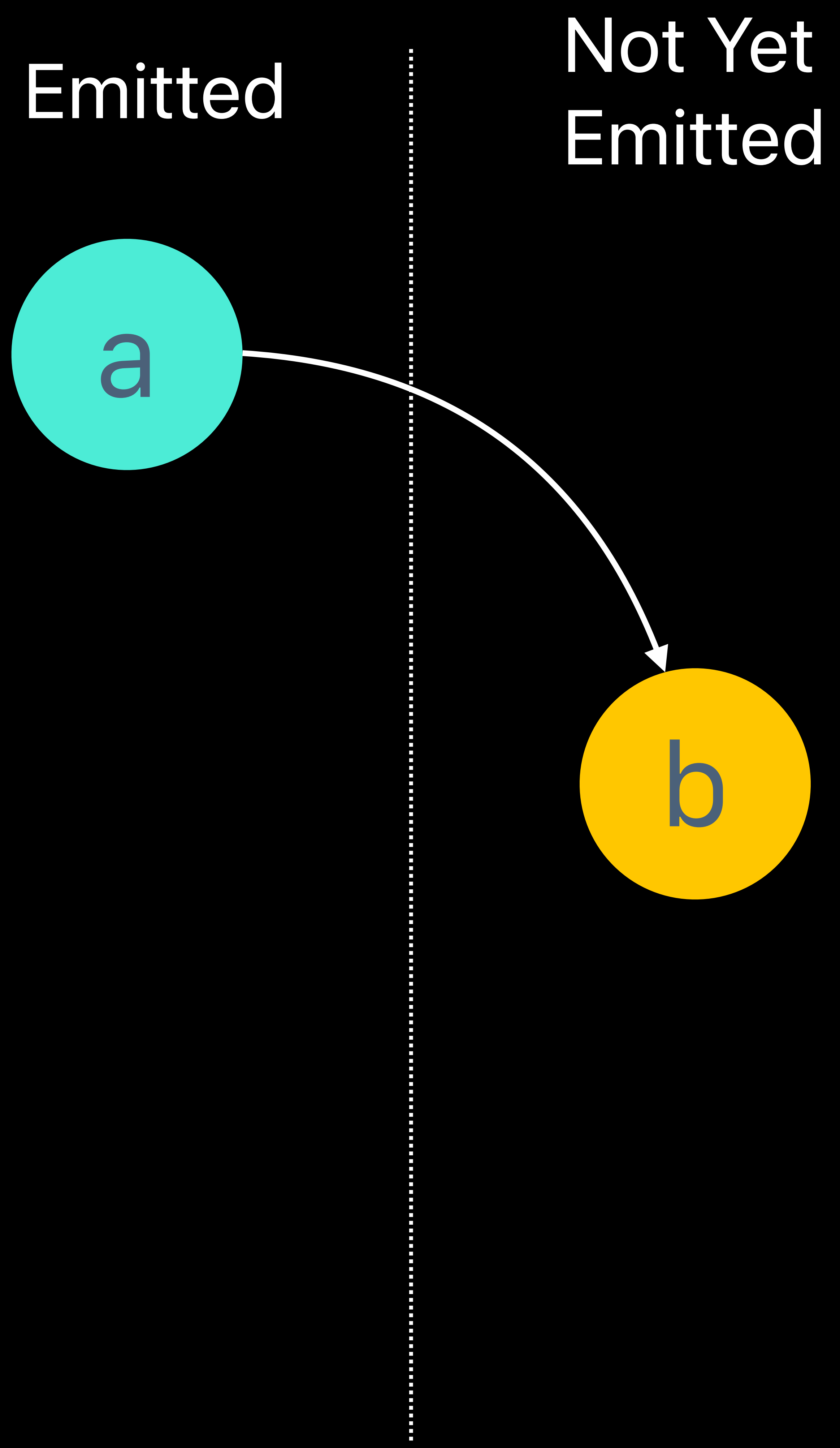
- Add-dependencies permits arbitrary graphs…
  - add-dependencies({a → {b}})
  - add-dependencies({b → {c}})
  - add-dependencies({c → {a}})
  - …



WaitingOnGraph

Merging `add-dependencies` into `emit`

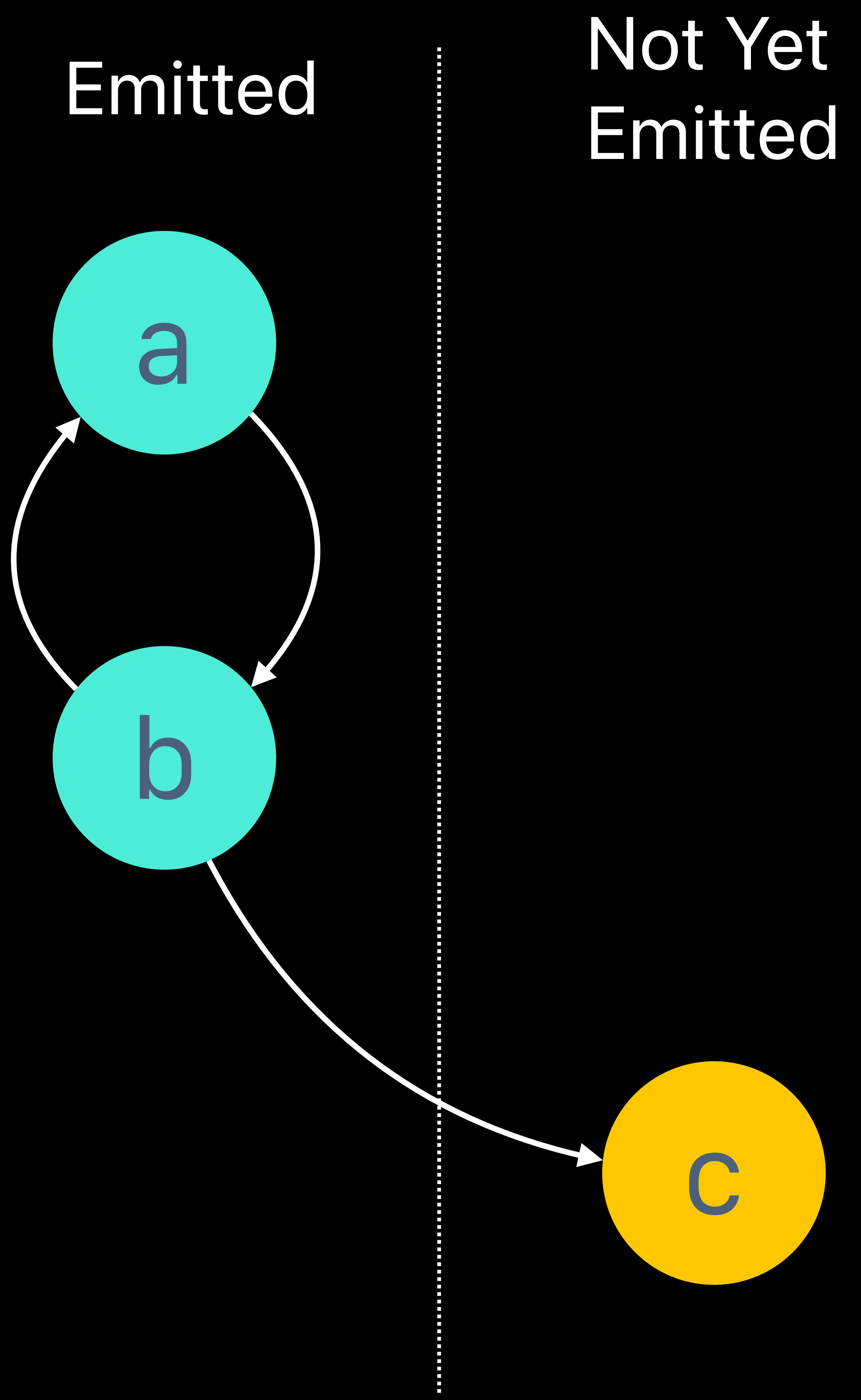
- Causes the graph to become bipartite
  - *emitted* → *not-yet-emitted* nodes
  - temporary cycles removed before `emit` returns
- e.g. `emit({(a → {b})});`
  - `emit({(b → {c, a})})…`



WaitingOnGraph

Merging `add-dependencies` into `emit`

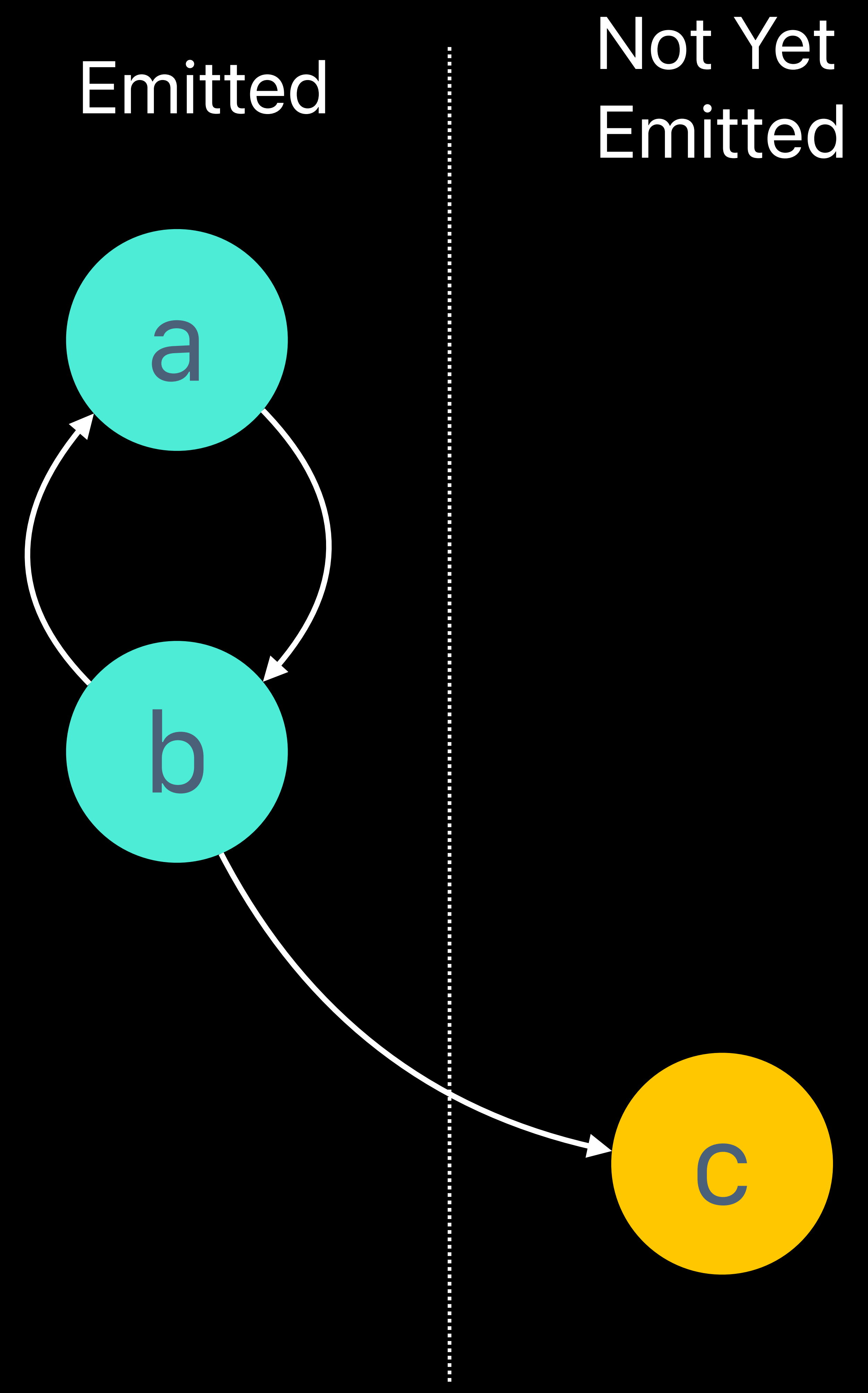
- Causes the graph to become bipartite
  - *emitted* → *not-yet-emitted* nodes
  - temporary cycles removed before `emit` returns
- e.g. `emit({(a → {b})});`
  - `emit({(b → {c, a})})…`



WaitingOnGraph

Merging `add-dependencies` into `emit`

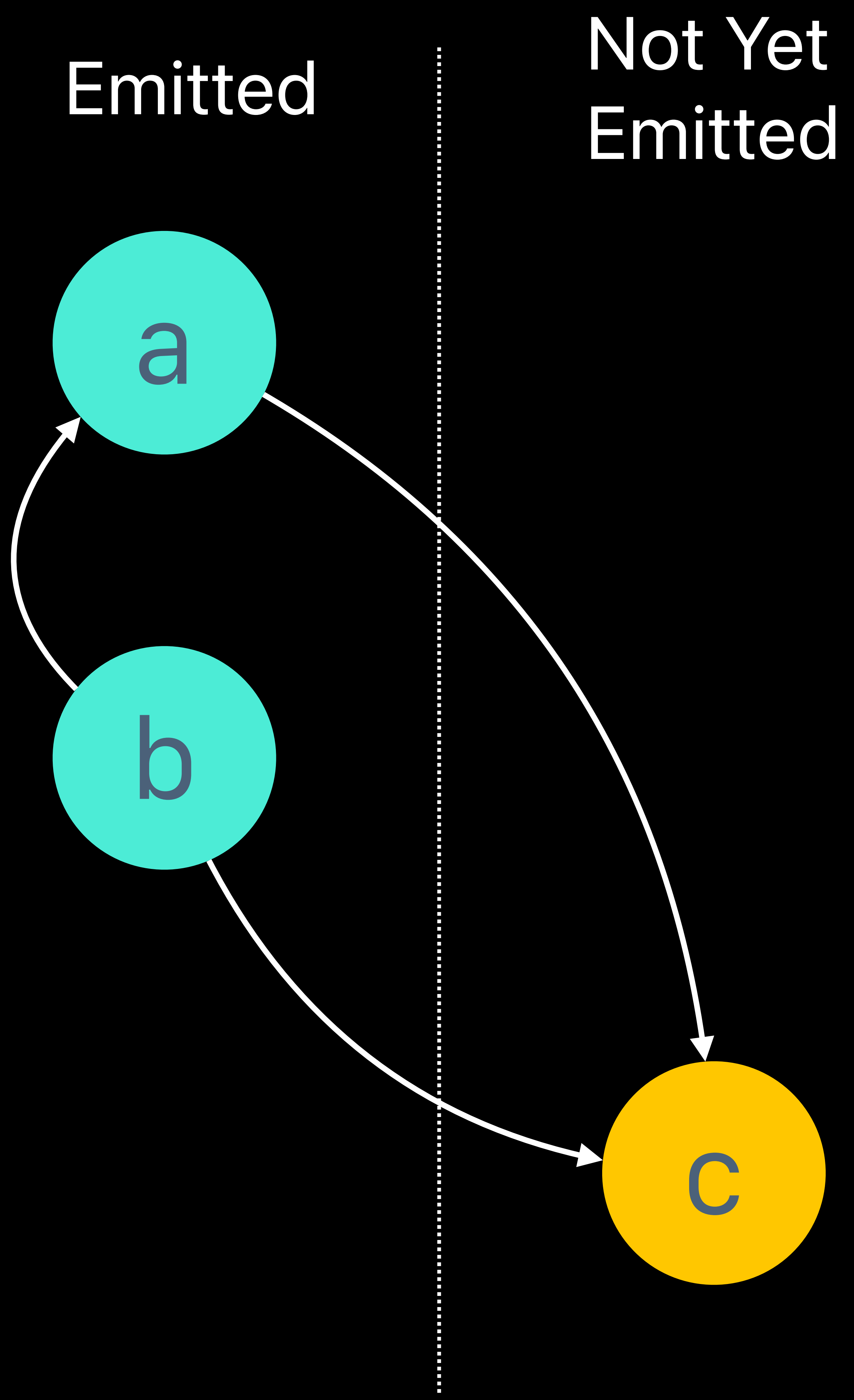
- Causes the graph to become bipartite
  - *emitted* → *not-yet-emitted* nodes
  - temporary cycles removed before `emit` returns
- e.g. `emit({(a → {b})});`
  - `emit({(b → {c, a})})`…
    - Propagate edges to not-yet-emitted nodes…



WaitingOnGraph

Merging `add-dependencies` into `emit`

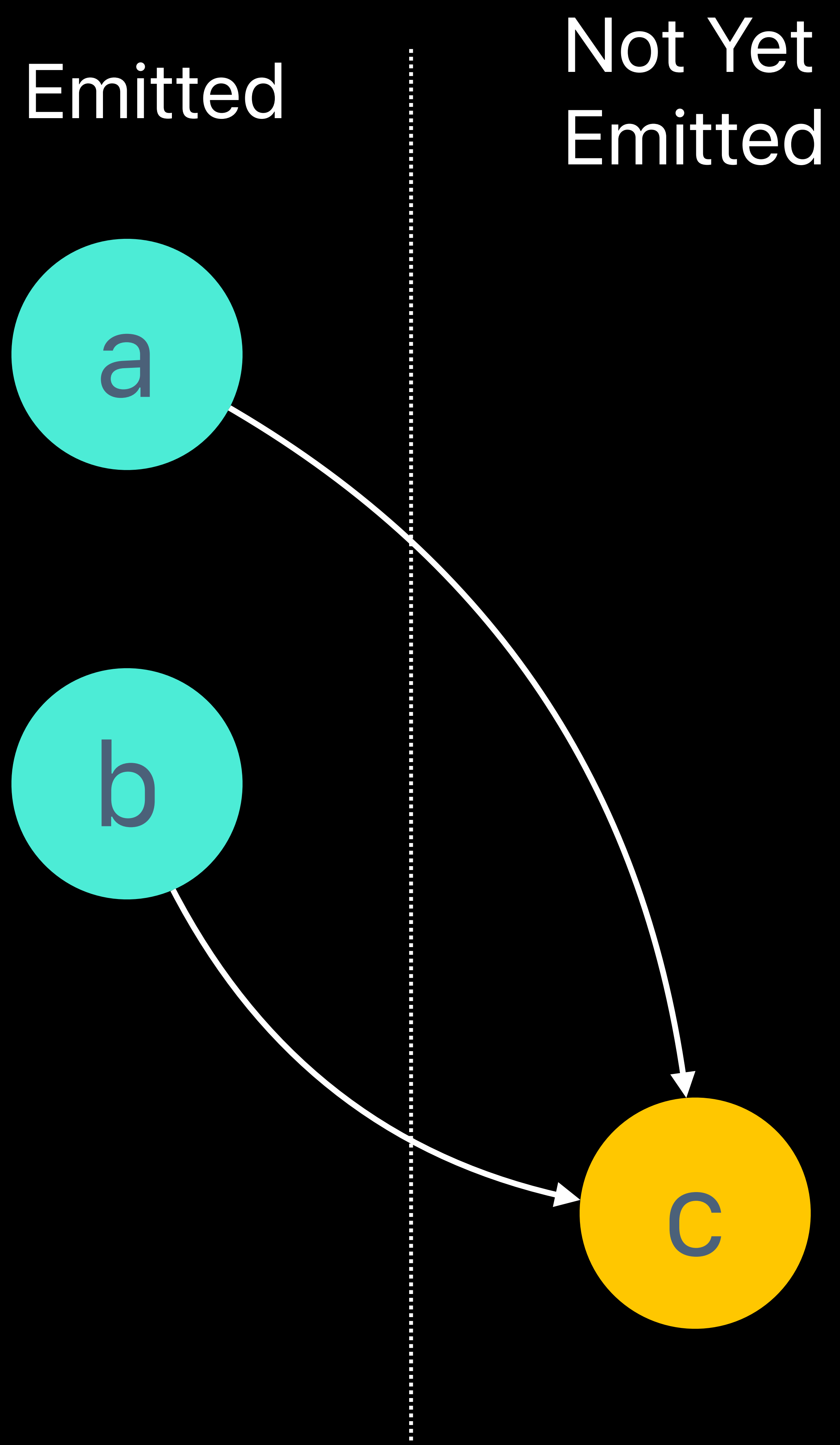
- Causes the graph to become bipartite
  - *emitted* → *not-yet-emitted* nodes
  - temporary cycles removed before `emit` returns
- e.g. `emit({(a → {b})});`
  - `emit({(b → {c, a})})`…
    - Propagate edges to not-yet-emitted nodes…
      - a depends on c, not b



WaitingOnGraph

Merging `add-dependencies` into `emit`

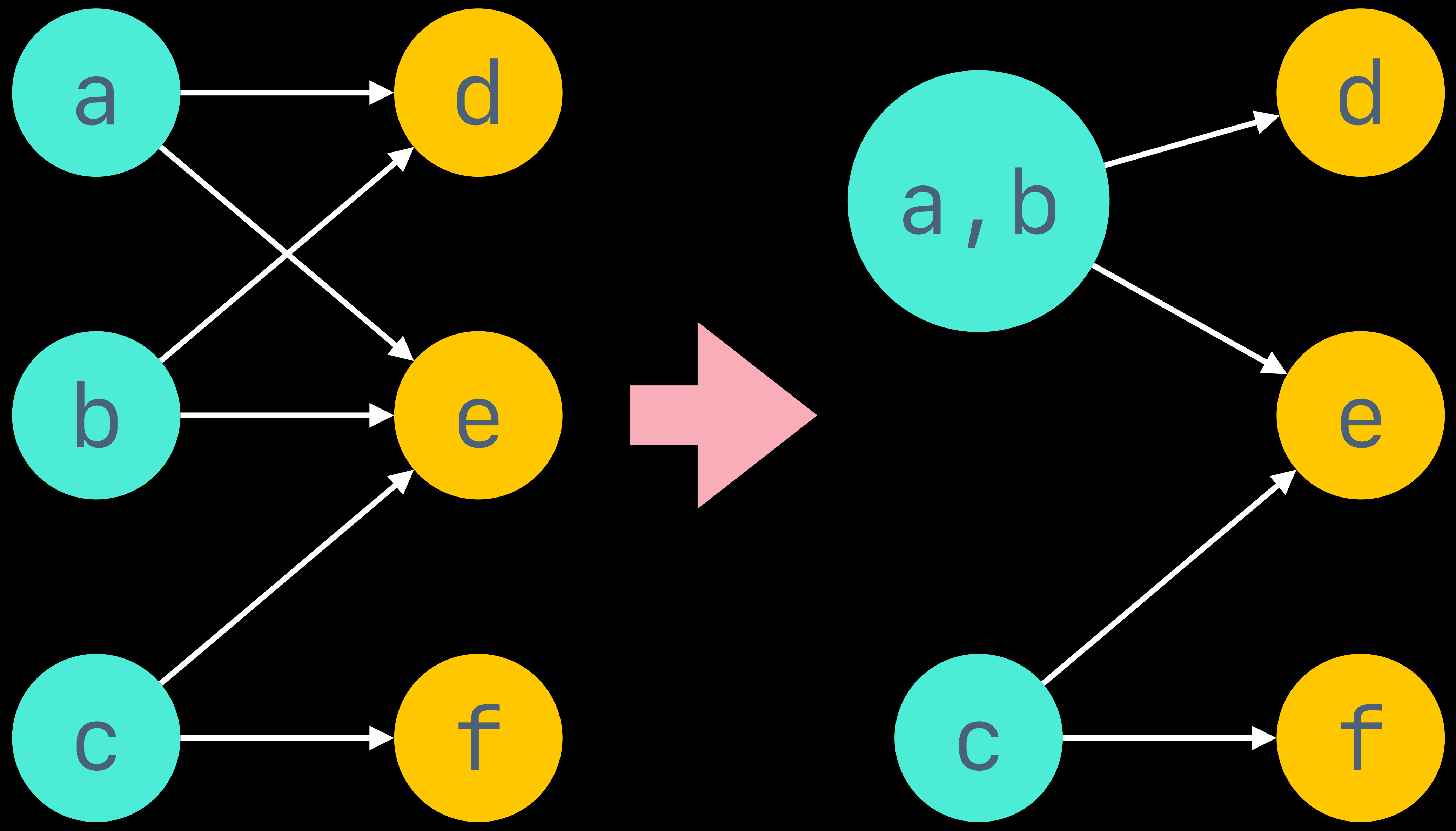
- Causes the graph to become bipartite
  - *emitted* → *not-yet-emitted* nodes
  - temporary cycles removed before `emit` returns
- e.g. `emit({(a → {b})});`
  - `emit({(b → {c, a})})`…
    - Propagate edges to not-yet-emitted nodes…
      - a depends on c, not b
      - b depends on c, not a (redundant, so discard)
- Preprocess `emit` arg: same algorithm, outside lock



`WaitingOnGraph`

Further improvements — Coalescing

- Shrink graph by merging nodes with same edges
    - E.g. `a, b` share edge sets `{d, e}`, so merge
- Currently applied to…
    - `emit` *input* after preprocessing, but before taking global lock
    - `emit` *output* before releasing global lock
- Effective in practice:
  Many nodes depend on same heavily used symbols



`WaitingOnGraph`

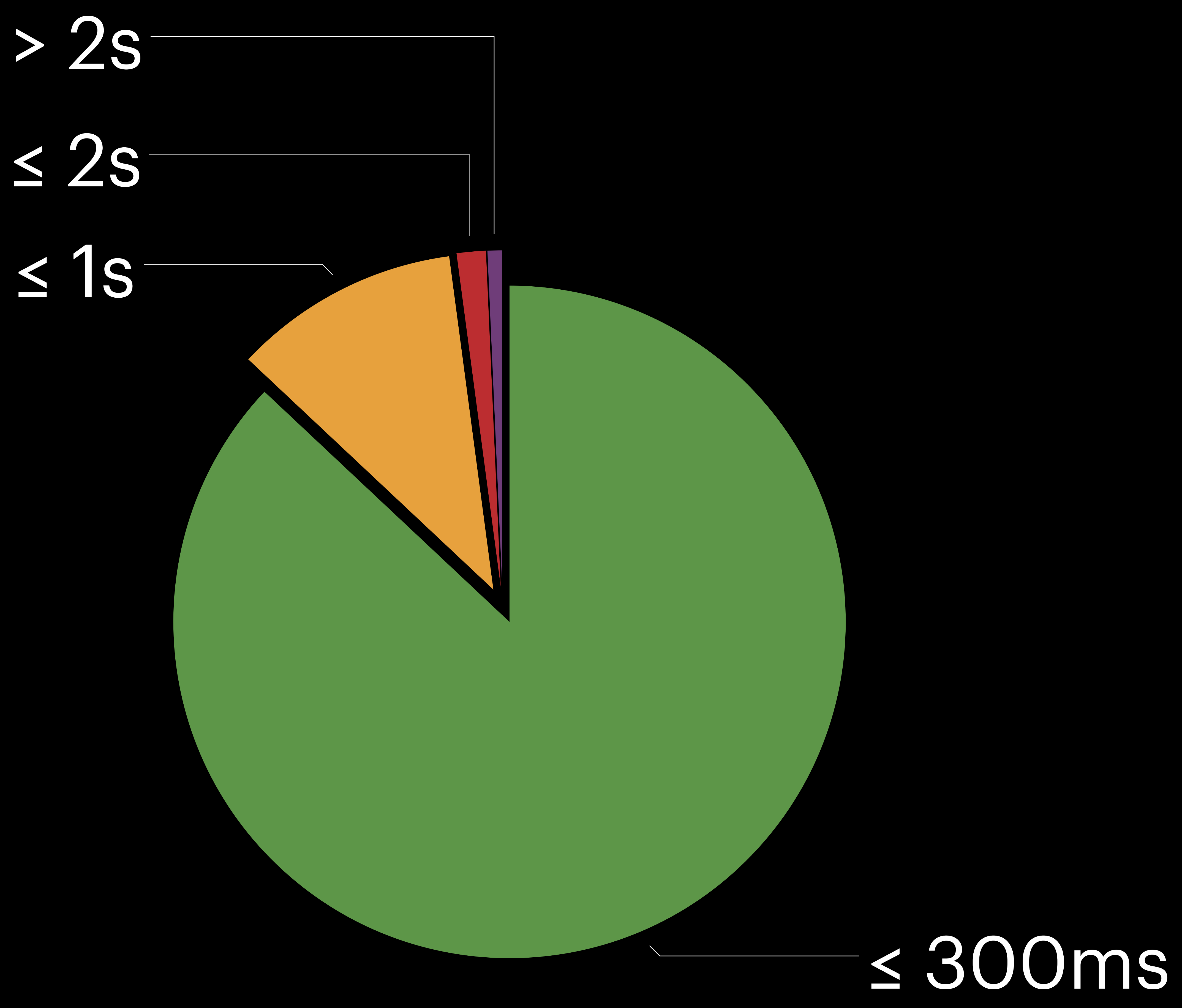
Wrapping Up

- `WaitingOnGraph` extracted from `JITDylibs` into its own class template
  - Directly unit and perf testable, tests covering previous error cases added
- Node labels have been changed to eliminate redundant reference counting
- Significant improvements on pathological cases (e.g. from >500s to ~2s)
  - Laziness would further simplify this problem
    - Lazy stubs don't wait on their implementations, they're terminals in `WaitingOnGraph`

Performance Results

Time for Previews JIT Update

- Rough numbers (includes some build time)
- Many small projects contribute to fast times
- Previews that take too long lead to users avoiding the feature, suppressing slow times
- Pathological cases remain
- Performance work will continue



The Weird Cases…

Naming archives ".o"

SUPPORTED

- ORC's APIs are strict: `addObjectFile` expects objects; `linkArchives`, archives
- Darwin's linker, `ld`, is chill — just wants you to succeed
  - Extensions don't matter, as long as your paths resolve to something linkable (objects, archives, universal binaries, etc.)
- We've added `orc::loadLinkableFile(Path, Triple, LoadArchives)`
  - Handles objects, archives, universal objects, universal archives, non-universal archives of universal objects…

`ld -r`

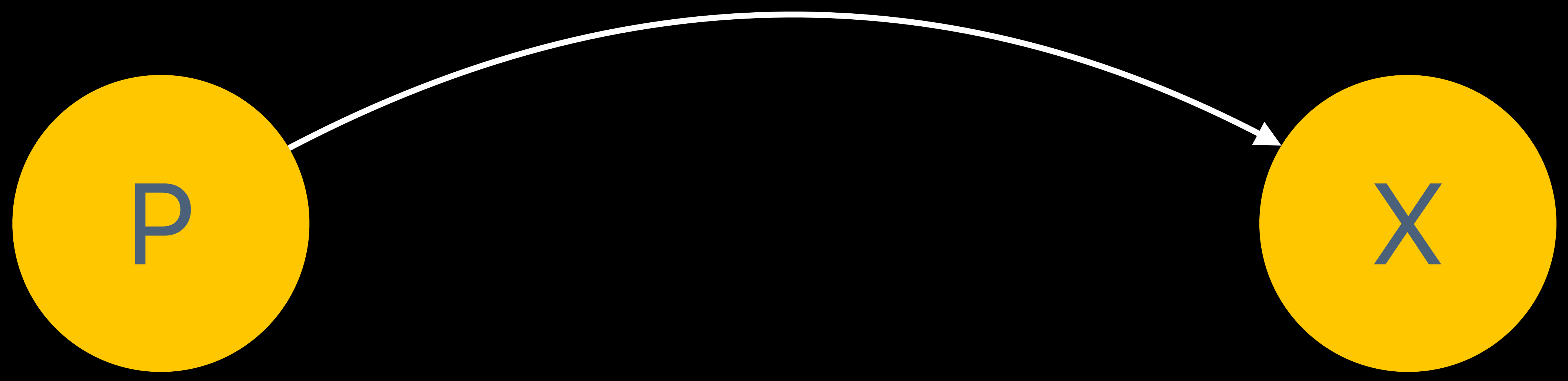
Local symbol names may not be unique

SUPPORTED

- ld -r merges relocatable object files
- `ld -r`′d objects may contain duplicate symbol names (local linkage only)
  - E.g. two C files containing `static int X = 1`, combined using `ld -r`, will have two Xs.
- Swift package manager does it, so transitively everyone does it
- We′ve removed all assumptions that locally scoped symbol names are unique

Pointer Authentication

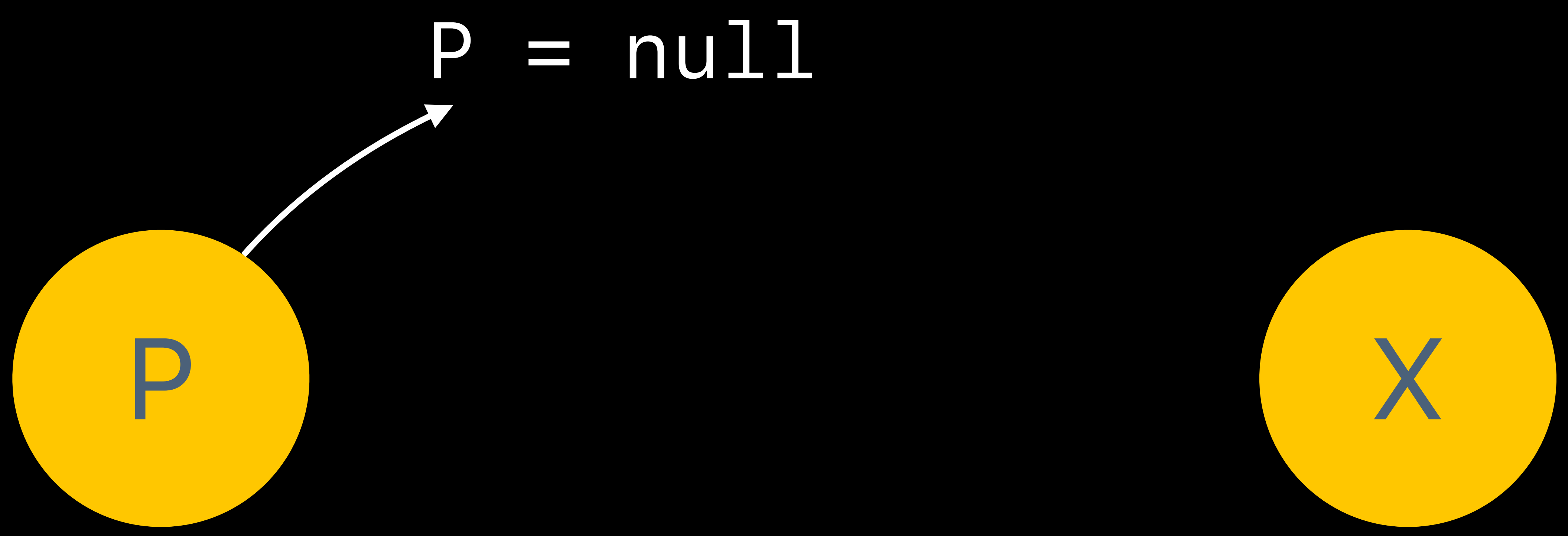
- arm64e pointer authentication is supported
  - *Without* introducing a trivial oracle
  - Authentication edges become instructions in a signing function run as initializer



Pointer Authentication

SUPPORTED

- arm64e pointer authentication is supported
  - Without introducing a trivial oracle
  - Authentication edges become instructions in a signing function run as initializer
    - Writes fixed values to fixed locations
    - Does anyone know if this is exploitable?


P = null

```
_sign_ptrs:
  mov x0, x
  autia x0
  mov x1, p
  str x1, x0
```

So much more!

SUPPORTED

- Compact unwind support — C++ exceptions on Darwin/arm64
- `.subsections_via_symbols` directive — can now be omitted
- Weak-loading (`-weak-l`), hidden-linking (`-hidden-l`) — see `llvm-jitlink` for examples
- `-all_load`, `-ObjC` options — force loading of all (or all Obj-C) objects in an archive
- Objective-C stub synthesis (`call _objc_msgSend$foo`)

Conclusion

Xcode Previews

- ORC can…
  - JIT-load programs that were intended to be statically linked
  - Scale to non-trivial programs
  - Support unusual build configurations, execution environments
- With dynamic loader support, precompiled code can interact with JIT'd code as-if precompiled
- Many improvements made for Previews should flow to other ORC clients:
  clang-repl, Jank, Clasp, Julia, Mojo, PostgreSQL, …

Conclusion

Developer Workflow Opportunities

- JIT mode for edit/test — rather than building what has changed, build only what you *need*
  - Faster compiles, no need to select build options/targets to avoid unnecessary compilation
- Incremental builds still required to validate — ideally we share compiled code between modes
  - Use Content Addressable Storage with fine-grained sharing — youtu.be/E9GdNKjGZ7Y
- Straw-man — take the LTO approach
  - Generate ".o" files with symbol interface (via TextAPI) & compile command only, feed to JIT
- Per-function requests to front-ends? How would this affect build systems? Tooling?

Conclusion

ORC — Future Work

- Move LLDB from MCJIT to ORC — would allow LLDB to benefit from these improvements
- In-tree memory manager implementations could be improved (esp. to reduce fragmentation)
- New ORC Runtime — all asynchronous operations, new features
- Dynamic loader integration for ELF? COFF?
- There are, shockingly, still some open JIT bugs
- Contributions very welcome!
  - Github Issues, PRs, Discourse, Discord (#jit)

Wait, how does ORC laziness work?

- Symbols are produced when you ask for their address: they're lazily generated upon *reference*
- `lazyReexport` produces stubs that look-up and then call function body symbols at runtime
  - I.e. Stubs are produced upon reference, and defer reference of function body until first call
- With this scheme, laziness inherits lookup safety:
  - Call any stub on any thread at any time
    - Safe regardless of which compiles are invoked, or what's happening on any other thread