June 8th 2022

Write JITLink support for a new format/architecture

Mentors: Vassil Vassilev, Lang Hames, Stefan Gränitz
Student: Sunho Kim

Google
Summer of Code

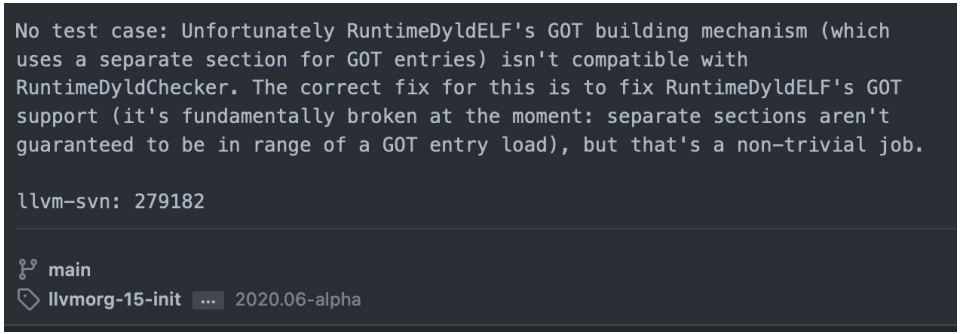
Just in time linker

- Do the same job of LLD but just in time
  - Receives object file (“.o file”) and link in memory to an executable form
- Benefit of using object file format
  - Can use the same compilation pipeline with AOT llvm world
  - Not a lot of overhead; no need to store to file system
- Not a new concept introduced by JITLink
  - Already implemented in RuntimeDyLD which is used in various third parties notably Cling, Julia, Swift, postgresql, mono, and more

Issues of Old JIT Linker

- RuntimeDyLD
- First committed in 2011
- Many features not working correctly: hard to make asynchronous, weak and hidden symbols, static initializer, thread locals, small code model

Issues of Old JIT Linker

- Some horrors
  - https://github.com/llvm/llvm-project/blob/main/llvm/lib/ExecutionEngine/RuntimeDyld/RuntimeDyldELF.cpp#L1217 (RuntimeDyldELF::processRelocationRef)
  - 

Issues of Old JIT Linker

- Complaints written by Lang Hames who maintained RuntimeDyLd for several years
  - The error checking was spotty at best. RuntimeDyld would often fail silently, leading to difficult to debug crashes in JIT'd code.
  - The backends weren't properly separated. Relocation was basically a giant nest of switch statements. Lang separated it out into COFF, ELF, and MachO backends, but never managed to break the ELF backend up into per-arch backends.
  - Relocation coverage was very limited, and didn't handle the default code and relocation models. This means that you could only link code that was deliberately compiled in "MCJIT-friendly" mode.
  - The internal data structures were too simple to handle things like dead-stripping, or proper GOT and PLT handling.
  - Native TLV wasn't supported -- this needs runtime support, and Lang and maintainers didn't have good ideas about how to manage RuntimeDyld <--> runtime interactions.
  - The system was a black-box (with the exception of specific events reported via the JITEventListener API).
  - On debugging the RuntimeDyld asserts -- they basically needed to print the llvm ir and dump the object file and then compute offsets by hand.
  - More….

JITLink

- A new just in time linker in LLVM
- Development started in 2019 by Lang Hames
- Work-in-progress replacement for RuntimeDyLD, old JIT linker
- LinkGraph abstraction made from the lessons learned in RuntimeDyLD
- Asynchronous by design
- All features supported: thread locals, runtime, static initializer, small code model, weak and hidden symbols, etc…
- Small code model is one of major immediate gains

Small code model

- https://github.com/JuliaLang/julia/issues/42295
- As they added support for m1 mac, they experienced random hangs and seg faults because large code model is not native in macho/arm64.
  - Macho object format used in darwin just don't have relocation type to support large code model easily
- They switched to JITLink, and used small code model. Seg faults disappeard!
- It's not unique to darwin, even on aarch64 linux, unknown errors observed across multiple users (cling, mono, swift notably) when using large code model.
- Main reason: small code model is the native, default, performant choice
- Clasp (JIT common lisp) noticed 10x to 78x slowdown in exception handling in large code model.

LinkGraph

- Addressable “nodes” (which represents the memory block) has relocations “edges” to symbols
- Local symbol = addressable block + offset
- Several link passes that process the graph step by step
  - Steps are all asynchronous by default
- Somewhat similar to atom graph abstraction of lld (https://releases.llvm.org/11.1.0/tools/lld/docs/design.html)

Benefits of LinkGraph

- Code can be shared across backends
  - Generic EH frame handling pass is used in ELF/X86 (x86 *nix), MachO/X86 (intel mac), and MachO/ARM64 (m1 mac) with no specialization
- One can edit the graph within each pass
  - Add new relocation edges freely – actually made possible some pass to be shared across backend
  - Allocate and emit code blocks freely step by step – base for supporting small code model robustly
- Many optimization opportunities
  - Devirtualization, dead symbol stripping, got indirection optimization
- Personally, it’s been just pleasant to work with

But, not all architectures and platforms supported…

- Main reason why it still hasn't replaced RuntimeDyLD even if it's pretty stable on supported targets.

	Linux (ELF)	Mac (MachO)	Windows (COFF)
ARM64	X	O	X
X86	O	O	X
PPC64	X	X	X
RISCV	O	X	X

My project

1. Write ELF/AARCH64 backend for JITLink to support arm64 linux.
    a. Various related issues in real world projects:
    b. https://github.com/cms-sw/cmssw/issues/31123
    c. https://github.com/JuliaLang/julia/issues/42295
    d. https://github.com/dotnet/runtime/issues/46881
    e. https://github.com/apple/swift/issues/57535
2. Write COFF backend for JITLink to support windows.
    a. There is no code written to support windows in JITLink currently
    b. COFF support in RuntimeDyLD was not ideal either

Current progress

- Got most of the major features working in aarch64 linux and submitted patches over the last two weeks
- Can run complicated c++ object files with exceptions, externs, vtables, and static variables.
- Could reuse huge portion of code thanks to clear architecture of JITLink
- **Completed:** Typical branch/ldst/data relocations, Global offset table, Procedural linkage table, Eh frame handling, Static initializers
- **Incompleted:** Thread locals, Battle test on real softwares
- Likely dig into COFF backend that is required for windows after finishing aarch64

Timeline

Note that this is very rough estimate

June: Land all pending elf/aarch64 patches to upstream, Complete elf/aarch64 thread locals, Try JITLink in julia on aarch64 gnu linux

July: Polish elf/aarch64 implementation

August: Write a generic COFF link graph builder and fixup specializations for one architecture (TBD, likely x86)

September: Look into COFF ORC runtime support.