LLVM JITLink

The Next Generation LLVM JIT Linker

Required Actions for PowerPC
V0.8

Jan. 2021 - abebeos@lazaridis.com
(parts are based on emails with lhames@gmail.com)
(context: https://github.com/abebeos/PowerPC/issues/4, Stage 2)

Summary

No immediate action needed.

Possible Pre-Tasks:

- extract base-classes (refactoring) to enable PowerPC implementation
- Provide public ppc64le cross-project CI (llvm/llvmlite/numba)

R: will be created by refactoring
N: newly to implement

[LLVM Compiler]

[ORC] - LLVM's newer JIT APIs. These are the primary client for JITLink
  [ORCv1] - deprecated, will be removed in llvm-13
  [ORCv2] - works with RuntimeDyld, too

[llvm-jitlink] - jitlink cli. Primarily for testing purposes

  [JITLink](1) - Next Generation LLVM JIT Linker API
    R:**[arm64 relocation edges]** Generic reloc-types for arm64 architecture.
    R:**[x86-46 relocation edges]** Generic reloc-types for x86-64 architecture.
    N:**[PPC relocation edges]** Generic relocation types for PPC architecture.
    N:[...]

    [LinkGraph] Core data structure, used by JITLInkerBase
    [JITLinkerBase] - Generic JIT linker algorithm. Subclasses handle reloc.
      R:**[JITLinker_arm64]** Implement arm64-specific relocations.
      R:**[JITLinker_x86_64]** Implement x86-64 specific relocations.
      N:**[JITLinker_PPC]** Implement PPC-specific relocations.
      N:**[...]**

    [MachOLinkGraphBuilder] (Generic. Subclasses handle relocation parsing)
      [MachOLinkGraphBuilder_arm64]
      [MachOLinkGraphBuilder_x86]

    R:**[ELFLinkGraphBuilder](2)** - Generic. Subclasses handle reloc parsing)
      [ELFLinkGraphBuilder_PPC]
      [ELFLinkGraphBuilder_x86_64]

LingGraphBuilder Entry points

      [createLinkGraphFromMachOObject_x86_64] - x86_64.
      [createLinkGraphFromMachOObject_arm64] - arm64.
      [createLinkGraphFromELFObject_x86_64] - x86_64.
    N:[createLinkGraphFromELFObject_PPC] - PPC.
    N:[createLinkGraphFromELFObject_ARCH] - ARCH.

      [link_MachO_arm64] Config/run JITLinker_arm64 for MachO/arm64 input.
      [link_MachO_x86_64] Config/run JITLinker_x86_64 for MachO/x86-64 input.
      [link_ELF_x86_64] Config/run JITLinker_x86_64 for ELF/x86-64 input.
    N:[link_ELF_PPC] Configures and runs JITLinker_PPC for ELF/PPC input.
    N:[link_ELF_ARCH] Configures and runs JITLinker_ARCH for ELF/ARCH input.

(1) There are no "architecture relocation edges", instead the relocation edges are currently specified for (format, architecture) pairs, e.g. MachO/x86-64. I have decided that this artificially limits code sharing, so we plan to switch to the scheme above as soon as possible, and as early as late next week.

(2) There is no ELFLinkGraphBuilder yet, instead there is a specific ELFLinkGraphBuilder_x86_64 implementation. This is just a consequence of it being Jared's first attempt at the design. He and I will work together to fix this in the next couple of weeks.

Basic JITLink Flow:

The core data type is the LinkGraph, which contains a name, a target triple, and a list of Sections and external Symbols. Each Section contains a list of Blocks and Symbols (Symbols are (Name, Block-offset) pairs). Each Block contains Content (a blob of bytes) and a list of Edges. Each Edge has a Kind (relocation type), Target (a Symbol), and Addend.

The LinkGraph represents a parsed relocatable object file. Blocks are slabs of bytes that can be moved relative to one another, but in general can not be broken up (unless you have some special insight into the content). In ELF you typically have one Section per function / variable, and one Block per Section. In MachO you typically have a fixed handful of Sections (.e.g. .text, .data), and multiple blocks per Section (usually one Block per function / variable).

There is a class, JITLinkGeneric, in the lib folder which implements the core linker algorithm. This is comprised of a number of phases: (1) dead stripping, (2) allocation, (3) external symbol resolution, (4) fixups, (5) finalization. You do not need to reimplement any of this.

[section “The primary tasks when supporting a new architecture are” moved to end]

Clarify who is doing currently what

Lang Hames (@lhames) is the llvm code owner for JITLink. He designed it, wrote the MachO implementations (x86-64 and arm64), and continues to maintain and develop it.

An early ELF implementation for x86-64 is being developed by Jared Wyles and Stefan Granitz, with some support from lhames.

Generic ELFLinkGraphBuilder: lhames, Jared, possible assistance from implementers of architectures.

No dedicated PowerPC developers is currently active.

Relevant Sources and Docs

Sources are in **llvm/include/ExecutionEngine/JITLink** and **llvm/lib/ExecutionEngine/JITLink**. There are few documents at the moment, other than the code and comments.

https://llvm.org/docs/ORCv2.html#transitioning-from-orcv1-to-orcv2

Most target-code will end up within::

- `llvm/include/llvm/ExecutionEngine/JITLink/ELF_PPC.h`
  - `-- createLinkGraphFromELFObject_PPC, link_ELF_PPC.`
- `llvm/include/llvm/ExecutionEngine/JITLink/PPC.h`
  - `-- PPC edge kind definitions,.`
- `llvm/lib/ExecutionEngine/JITLink/ELF_PPC.cpp`
  - `-- ELF_PPC.h implementation, plus ELFLinkGraphBuilder_PPC.`
- `llvm/lib/ExecutionEngine/JITLink/PPC.{h, cpp}`
  - `-- ELFLinker_PPC.`
- `llvm/test/ExecutionEngine/JITLink/PPC`
  - `-- Regression test files.`

Reference-Systems

- RuntimeDyld(ELF)
- The current JITLink MachO Implementation
- Upcoming JITLink x86-64 Implementation

Estimate completeness-grade of base-implementations

The MachO implementations are largely complete. The current ELF implementation is an x86-64 specific prototype, hopefully soon to be generic.

Assess quality of base-implementations (e.g. inheritance/inheritance-like reuse of base sources)

(...)

Deliverables

High Level:

A ELF/PPC JITLink implementation that supports at least the PIC relocation model, and default code model. This would represent a minimum viable solution to enable a switch from RuntimeDyld to JITLink. It is unlikely to regress any performance, and may improve it.

Lower level deliverables:

- PPC-specific edge kind definitions, and fixup implementations.
- ELF/PPC LinkGraphBuilder.
- ELF/PPC GOT/TOC and PLT support.

Stretch goals / bonuses:

GOT / PLT bypass -- GOT/PLT entries are used conservatively when a target *might* be out of range. On x86-64 we remove these after the external resolution phase if we can prove that the target is within range of the caller/user. This can improve performance of JIT'd code on some benchmarks.

eh-frame support (maybe) -- If PPC uses dwarf-style eh-frames you can add JITLink's existing eh-frame support passes to process them, which would enable exception handling support in JIT'd code.

thread local support (probably not) -- If PPC uses thread locals you can look into adding support for these, however no other architecture has this yet, and it usually requires runtime support.

Why Early Migration to JITLink?

How much "pressure" is there to migrate to JITLink (e.g., will RtDyldELF be deprecated at some point) ?

Lang Hames (Apple) is no longer maintaining RuntimeDyldELF. It will not receive any new feature work, and will only receive bug-fixes in critical cases. I consider the codebase to be in poor shape for a number of reasons:

1. It commingles code for multiple architectures and formats. A fix for one architecture has an unreasonable risk of breaking another. People working on one architecture are unlikely to test on others thoroughly.
2. Error checking is poor. Both programming bugs and JIT errors (e.g. unresolved symbols) are easy to miss and difficult to track down.
3. The RuntimeDyld linker model is limited (and poorly described). It is difficult to, for example, implement both GOT and PLT support for an architecture (RuntimeDyld says a symbol can have one "stub", and that's it). This leads to hacks to try to work around the problem, and limits the code models that can be used. This can in turn hurt performance (e.g. forcing many things to be indirected through registers using inefficient code patterns).
4. Support for advanced features (exception handling, static initializers, thread locals) is essentially non-existent.

RuntimeDyld is not officially deprecated (JITLink is too new), but it's very much in maintenance mode with a view to deprecation in the future. PowerPC does not need to move yet, but will be strongly encouraged to at some point. Moving earlier may offer performance benefits and access to new features.

OrcV1 will be removed in llvm-13 (https://github.com/llvm/llvm-project/commit/6154c4115cd4b78d0171892aac21e340e72e32bd)

Migrating to **OrcV2** can be done based on RuntimeDyldELF, in is not necessary to implement JITLink first.

Further Plans

This topic may be worth visiting towards the end of the year. JITLink is part of a grander plan (including ORC and the ORC Runtime), that has started to reach some of it's larger milestones. I hope to have a significant demo ready for the LLVM Developer's Meeting towards the end of this year, and it will probably include an announcement of full JITLink support for at least ELF/x86-64, and possibly ELF/arm64 (MachO/x86-64 and MachO/arm64 are already essentially complete). This may in turn inspire more contributors to the area, and increase the pressure on others to get on board.

Major Benefit of Early Involvement

- Functionality and clarity can be shaped / influenced

Supporting a new Target Architecture

Primary Tasks:

(1) If not already available, define generic edge kinds to represent the relocations available on the target architecture, and write relocation implementations for those kinds.

Nobody has done this for PPC yet, so you would have to do it. Some relocation types are quite straightforward. E.g. you usually have Pointer64 ("write the target pointer plus added here"), Delta32 ("write 32-bit delta from here to target, plus added"), Delta64 ("write 64-bit delta from here to target"). , and then a few more complex relocations (e.g. for the TOC and PLT).

(2) Write a LinkGraphBuilder implementation for your Format/Architecture. Your implementation will be responsible for building the LinkGraph from the input relocatable object file. Usually you will have a <Format>LinkGraphBuilder base class that you can subclass, and the only real task for the implementer is to handle relocation parsing to map from the format's relocations to the edge kinds defined in (1). This mapping is not 1-1, but should generally be straightforward.

(3) Implement TOC / PLT handling. On x86-64 this is GOT/PLT handling, which consists of building a pointer-sized entry for each GOT relocation target, and a jump stub that references the GOT entry for each PLT target. I suspect PPC's requirements are similar, but it seems like there might be some more significant range limitations (e.g. 8,192 GOT entries) that would force you to do more clever things than are required on x86-64.

(4) Wire up (add your newly supported triple to a few switch statements to indicate JITLink support), and write tests using the llvm-jitlink tool (See test/ExecutionEngine/JITLink for examples).

Testing and Implementation:

I would be inclined to start by writing a handful of very simple programs containing calls/uses representative of standard usage, then run readelf -r over them to get a sense of what relocations will be required.

(1) A PPC tester will be extremely useful, as we definitely want to see this actually execute code.

(2) Regression tests are actually possible on all architectures. JITLink is a cross-linker: You can link any supported target from any LLVM host platform. The llvm-jitlink utility has a '-noexec' option which tells it to load and link the target files, but not jump to main. It also has a '-check' option which you can use to specify a file containing text lines of the form:
jitlink-check: <expr>

These checks are run against the linked memory to verify that relocations were applied correctly. You can check out llvm/test/ExecutionEngine/JITLink/X86 for examples.

I think a rough ordering of possible goals for a JITLink/PPC project would be

A. Parity with RuntimeDyldELF for PPC.

B. Support for C code for PIC relocation model and default code model. This makes migrating from MCJIT to ORC attractive from a maintenance (and possibly performance) perspective.
C. Full relocation support. Use any relocation / code model.
D. Support for exception handling. JITLink supports improved exception handling on x86-64. It is likely (but not certain) that you could re-use substantial portions for PPC.
E. Support static initializers. Enables full C++ support.
F. Performance optimizations. Elide TOC / PLT uses for in-range targets. Further improves performance.

Dev System Requirements

Ideally

- **Testability**: all supported architectures must be covered by a publicly available test-system (ci), which has its full source-code available, thus it can be reproduced in remote forks and locally.
- **Convenience**: make migration painless, ideally drop-in replacement (which can be later extended)

Resources

https://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi.html#MOD-OVER