Proposed by:

Sunho Kim

GSOC Proposal

Write JITLink support for a new format/architecture (ELF/AARCH64)

Contact Details

Name: Sunho Kim

Github username: sunho

Email: ksunhokim123@gmail.com

LLVM discord username: sunho

LLVM discourse username: sunho

Project Info

Project Name: Write JITLink support for a new format/architecture (ELF/AARCH64)

Mentors: Vassil Vassilev, Lang Hames, Stefan Gränitz

Size of Project: Large

Expected Deliverables: JITLink specialization for ELF/AARCH64 format.

Introduction to LLVM and JIT

LLVM is an open source framework for building compiler toolchains. It is a powerhouse behind various modern languages such as swift, rust and julia which use LLVM for generating efficient machine code. There are two major approaches in the compiler world: AOT (ahead of time) and JIT (just in time). In AOT compilation, the compiler transforms the entire source code to machine code before the runtime. Whereas in JIT compilation, the compiler runs “just in time” during the runtime, transforming some part of source code to machine code on demand. Since AOT compilation and JIT compilation differ in many characteristics, LLVM supports separate APIs and toolchains dedicated for JIT compilers.

JITLink API and its Motivation

JITLink is LLVM's new JIT linker API--the low-level API that transforms compiler output (relocatable object files) into ready-to-execute bytes in memory. It was developed to eliminate the restrictions of previous JIT linker API RuntimeDyld. Among those restrictions, the most critical ones are lack of support for the small code model and prohibition of certain features such as thread local storage. The Julia language team switched to JITLink API recently because of support for the small code model. When adding support for aarch64-darwin target, Julia team suffered from bugs within the large code model implementation of aarch64 backend, which caused numerous random hangs and segmentation faults. After switching to JITLink API with a small code model, these segmentation faults disappeared.

Project Description

JITLink's generic linker algorithm needs to be specialized to support the target object format (COFF, ELF, MachO), and architecture (arm, arm64, i386, x86-64). LLVM already has mature implementations of JITLink for MachO/arm64 and MachO/x86-64, and a relatively new implementation for ELF/x86-64. In this project, I will write JITLink specialization for ELF/aarch64 target by reusing/refactoring the already existing specialization for ELF/x86.

Roadmap

- Set up basic linker implementation that loads elf object file and iterate over relocation sections
    - Would be preferable if we can separate out more of the common logic from ELF/X86 and MachO/ARM64 format.
    - Most of the plumbing work is already done in the upstream.
- Implement basic fixup edges that are needed for running hello world c program
    - These are the relocation entries found in hello world c object file: R_AARCH64_ABS64, R_AARCH64_PREL32, R_AARCH64_ADD_ABS_LO12_NC, R_AARCH64_ADR_PREL_PG_HI21, R_AARCH64_CALL26
    - I would need to add code to applyFixup function in ELF_aarch64.cpp.
    - Adding test cases that are similar to ELF_x86-64_common.s would be preferable, utilizing llvm filecheck and jitlink-check infrastructure.
- Implement global offset table relocation to run c program with literals and global variables
    - These are the additional relocation entries found in a bit more complicated hello world c object file that is using global variable and string literal: R_AARCH64_ADR_GOT_PAGE, R_AARCH64_LD64_GOTOFF_LO15, R_AARCH64_LD64_GOT_LO12_NC, R_AARCH64_ADR_PREL_PG_HI21, R_AARCH64_ADD_ABS_LO12_NC
    - I would need to add code to applyFixup function and create a new TableManager that reserves the entries for GOT.

- Implement thread local storage relocation edges
  - These are the additional relocation entries found in a c++ program that includes thread, iostream, and uses thread_local variable: R_AARCH64_TLSDESC_ADR_PAGE21,R_AARCH64_TLSDESC_LD64_LO12, R_AARCH64_TLSDESC_ADD_LO12, R_AARCH64_TLSDESC_CALL
  - I might have to account for aarch64 linux abi which uses the TPIDR_EL0 system register for doing TLS related operations.
- Test julia lang on aarch64 linux with added ELF/AARCH64 jitlink support and fix bugs discovered accordingly
  - Related issue: https://github.com/JuliaLang/julia/issues/42295
  - Would need to turn on small code model

Works needed to be done:

ELF format library and generic JITLink implementation is already written in the llvm upstream. The generic ELFLinkGraphBuilder class implementation receives the ELFFile class instance and emits JITLink graphs containing external/internal symbols. In order to add support for ELF/AARCH64 format, we need to implement specialization dedicated to ELF/AARCH64 format and implement “fixup edges” that account for each target specific relocation type of ELF/AARCH64. Fixup edges implement the patching of stub addresses to the relocated symbol as requested by relocation entry of elf file. For example, in the ELF/X86 backend, applyFixup function patches the code like this:

```
case Pointer32: {
    uint64_t Value =
E.getTarget().getAddress().getValue() +
E.getAddend();
    if (LLVM_LIKELY(isInRangeForImmU32(Value)))
      *(ulittle32_t *)FixupPtr = Value;
    else
      return makeTargetOutOfRangeError(G, B, E);
    break;
  }
```

This corresponds to ELF/x86 relocation entry R_X86_64_32S. Numerous relocation entry types are used by LLVM codegen, so in order to link in-memory form object files generated by LLVM toolchain, we must have fixup edge implementation for each of these. An interesting possibility to look at would be to separate out shared aarch64 patch logic from MachO/arm64 target. As an example, ARM64_RELOC_BRANCH26 of MachO/ARM64 and R_AARCH64_CALL26 of ELF/AARCH64 do a very similar kind of modification. One of the important relocation entries to look into is those related to the global offset table. Global offset table is used to achieve position independent code–the code that can be executed regardless of where it is placed within memory. It is heavily utilized in llvm codegen for implementing global variables, literal constants, and more.

Timeline

April 20 – May 23: (before coding)

- Familiarize myself with the llvm jit codebase and infrastructure.

June 13 – June 20 (Official coding period starts):

- Start writing the implementation.
- Prepare test cases extracting from a simple hello world c program.
- Deliverable: test cases required in the next week and basic structure needed before implementing fixup edges.

June 20 – June 27:

- Support following relocation types
    - R_AARCH64_ABS64
    - R_AARCH64_PREL32
    - R_AARCH64_ADD_ABS_LO12_NC
    - R_AARCH64_ADR_PREL_PG_HI21
    - R_AARCH64_CALL26

- Make sure external symbols are resolved correctly
- Deliverable: ability to run a c program that prints one integer using llvm-jitlink tool in aarch64 linux.

June 27 – July 3:

- Support following relocation types
    - R_AARCH64_ADR_PREL_PG_HI21
    - R_AARCH64_ADD_ABS_LO12_NC
- Handle multiple relocation sections
- Fix bugs and tidy up the code
- Mostly prepare for the next major step
- Deliverable: ability to run a simple c program which have multiple '.rela.text' sections

July 4 – July 11:

- Add test cases with global offset table
- Add the support for global offset table
- Support following relocation types
    - R_AARCH64_ADR_GOT_PAGE
    - R_AARCH64_LD64_GOT_LO12_NC
    - R_AARCH64_LD64_GOTOFF_LO15
- Deliverable: ability to run a hello world c program with global variable and string literals

July 12 – July 18:

- Attempt running a simple c++ program that includes iostream
- Add thread local storage TableManager
- Implement following relocation types:
    - R_AARCH64_TLSDESC_ADR_PAGE21
    - R_AARCH64_TLSDESC_LD64_LO12
    - R_AARCH64_TLSDESC_ADD_LO12
    - R_AARCH64_TLSDESC_CALL
- Deliverable: ability to run a simple c++ program which include iostream

July 19 – July 26:

- Attempt running a c++ program that create threads and uses thread_local variable
- Fix bugs that are discovered during this process
- Deliverable: ability to run a somewhat complex c++ program using multiple threads and thread_local variables

JULY 29th MID TERM EVALUATION

August 1 – August 8:

- Finish the basic implementation and test the implementation against typical use cases.
- Add simple link optimization passes if a significant bottleneck is discovered.
- Write documentation
- Deliverable: Concise documentations

August 9 – August 16:

- Test the finished implementation against julia lang codegen on aarch64
- Fix bugs that are discovered during the process
- Deliverable: Measure of the quality of the product in a real world use cases

August 17 – September 5:

- Continued polishing and bug fixes.
- Deliverable: Mature ELF/AARCH64 format support

Personal Details

GSOC participation:

This is my first time registering for the GSOC program and I didn't apply to any other gsoc project other than this.

What are your prior compiler and compiler-related experience, if any (e.g. studies at the University, prior contributions)?

I have taken compiler courses in my college, but most of my compiler experience was gained by tweaking open source compiler projects (e.g. llvm, qemu) and working on hobby compiler projects in my free time. I learned the concept of IR language and JIT compilation by modifying the dynamic recompiler of qemu in order to use it for my emulator project. I have written a JIT compiler from scratch for a language called aheui. It features a simple IR language and multiple backends targeting x86, wasm, and aarch64. After writing an optimization pass that folds memory load-store pairs to one SSA value, I was able to surpass the fastest known implementation of aheui language that was using rpython by an order of magnitude.

Most recently, I have contributed two minor patches to llvm aarch64 backend last month. One of which was adding a DAGCombiner pass that reduces the machine instruction count of the pattern generated by the negative absolute value function. Another patch was a fix to sink operands of a certain simd instruction so that they can be matched by tablegen pattern that generates a more fitting instruction even if some of the operands are in another basic block.

Have you had any prior contributions to LLVM? If yes, please provide links to these contributions.

https://reviews.llvm.org/D117944

https://reviews.llvm.org/D118595

What programming languages do you have experience with and please estimate your level of experience? (e.g. C, C++, Python, Rust, etc.)

I'm very familiar with c++ as I used it for more than 4 years on a regular basis. These are some of the major projects I have written in c++:

- GPU accelerated vulkan ray tracing renderer:

  https://github.com/sunho/GPUSpectral

- JIT compiler for aheui language:

  https://github.com/sunho/AheuiJIT

In addition to that, I frequently participate in competitive algorithm contests using c++ where I have to write efficient and debuggable c++ code quickly. I'm familiar with some graph theoretic topics such as bipartite matching and graph coloring because of my competitive programming experience.

What other commitments do you have that might affect your ability to work during the GSOC period (exams, classes, holidays, other jobs, weddings, etc.)? Please list each and when they occur.

I have courseworks in the early period of gsoc, but I have no special commitment other than college. During the summer vacation which starts from June, I can work on llvm full time.