Proposed by:

Sunho Kim

GSOC Proposal

Re-optimization using JITLink

Contact Details

Name: Sunho Kim

Github username: sunho

Email: ksunhokim123@gmail.com

LLVM discord username: sunho

LLVM discourse username: sunho

Project Info

Project Name: Re-optimization using JITLink

Mentors: Vassil Vassilev, Lang Hames

Size of Project: Large

Expected Deliverables: Reoptimization support in LLVM JIT with various polishing in internals of LLVM ORC engine

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.

Motivation

One of the most exciting advantages of JIT compilers is the ability to compile “hot” functions on demand utilizing various runtime profiling metrics gathered by slow versions of functions. ORC API, LLVM’s JIT API, currently only offers a clean way of just in time compiling of functions only once–it doesn’t really let clients make use of the profiling metrics and “reoptimize” the function. Even though ORC API is generic enough to allow implementation of such features out-of-tree, many infrastructure must have been reinvented since ORC API currently doesn’t provide them.

The purpose of this proposal is to extend ORC API so that the user can supply the new IR definition of function and trigger asynchronous recompilation of that function while ensuring the function swap happens safely. This way, the ORC user can supply arbitrary reoptimization, opening up various possible applications: swapping double into float based on numerical stability profile (which CERN engineers have been looking into), applying path guided optimization based on branch profile data (which Julia team have been experimenting with), or switching from CPU version to GPU version of function with profiling code removed just in time.

Project Description

In Just-In-Time compilers we often choose a low optimization level to minimize compile time and improve launch times and latencies, however some functions (which we call hot functions) are used very frequently and for these functions it is worth optimizing more heavily. In general hot functions can only be identified at runtime (different inputs will cause different functions to become hot), so the aim of the reoptimization project is to build infrastructure to (1) detect hot functions at runtime and (2) compile them a second time at a higher optimization level, hence the name "re-optimization".

There are many possible approaches to both parts of this problem. E.g. hot functions could be identified by sampling, or using existing profiling infrastructure, or by implementing custom instrumentation. Reoptimization could be applied to whole functions, or outlining could be used to enable optimization of portions of functions. Re-entry into the JIT infrastructure from JIT'd code might be implemented on top of existing lazy compilation, or via a custom path.

Whatever design is adopted, the goal is that the infrastructure should be generic so that it can be used by other LLVM API clients, and should support out-of-process JIT-compilation (so some of the solution will be implemented in the ORC runtime).

Personal Details

GSOC participation:

I'm a second year undergraduate student in De Anza College and this is my second time registering for the GSOC program. 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)?

Last year I did Google Summer of Code project with LLVM foundations to add new target support in the new generation JIT linker JITLink. My contribution has fixed notorious random crashes and hangs that have blocked linux aarch64 target to be in a higher support tier and drastically improved the status of windows JIT support (COFF/x86_64) to the point it can just-in-time link microsoft STL library to memory without any modification. I presented details and usage of my work in the LLVM dev meeting 2022 as a tutorial speaker. (https://www.youtube.com/watch?v=UwHgCqQ2DDA)

Because of this background I'm very familiar with the JIT infrastructure of LLVM and also know some parts that can be improved on including some unpolished parts of my JITLink work last year.

I'm somewhat familiar with clang codebase as well. I've pitched a patch that makes clang's lexer able to handle incremental source code input in order for it to be usable in incremental c++ executors such as cling or clang-repl (https://reviews.llvm.org/D143142)

I also 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, it was able to surpass the fastest known implementation of aheui language that was using rpython by an order of magnitude.

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

I have over around 50 patches totalling 5000+ lines of change to the JIT infrastructure of LLVM and clang-repl that landed in my last google summer of code. Here are some major ones:

https://reviews.llvm.org/D128968

https://reviews.llvm.org/D130479

https://reviews.llvm.org/D128601

https://reviews.llvm.org/D126286

https://reviews.llvm.org/D130456

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 5 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'm experienced in competitive programming enough to advance to the national round of ACM-ICPC. In these contests, I have to write efficient and debuggable c++ algorithmic code quickly. I'm familiar with graph theoretic topics such as graph isomorphism, matching and 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 during the summer vacation which starts from June, I can work on llvm full time.

Roadmap

Unify all indirect stub logic into one place “stub table manager”

- Refactor JITLink stub creation
- Create JITLink API for creating stub outside of JITLink realm.
- Design and define the interface for stub table manager. It must be out-of-process aware. (e.g. remote executor)
- Expose a way to swap the stub pointer atomically
- Create a thorough test cases covering all use cases of stub table manager
- Try to still not break RuntimeDyLD if possible

Implement redefinable materialization units

- Materialization units that allows redefinition of symbols using pointer swap of stub table manager
- Should use the same async query infrastructure in ORC
- Must consistently handle multiple redefinition requests in flight problem (resolve them by FIFO order; can we cancel the some compilations in the middle)
- Must be generic enough to be usable by typical ORC user who want “redefine” functionality in ORC
- Examples that demonstrate usages and ORC test cases to cover new additions

Implement ReOptimizeLayer

- Layer that uses redefinable materialization units to implement reoptimization
- Use ORC runtime infrastructure in order to handle all register restoring issues and out-of-process RPC issue
- Handle reentry properly
- LLJIT example that demonstrates full reoptimization usage or clang-repl example that optimizes O0 code to O3 code just in time.
- If time allows, write a tutorial article on how to use it

Timeline

May 4 - 28: (before coding)

- Draft on designs of stub table manager and redefinable materialization units
- Discuss high-level design of how all components should work together with mentors

May 29 – June 14 (Official coding period starts):

- Start writing the implementation.
- Prepare test cases for stub table manager
- Write basic stub table manager that manages active stubs and memory allocations of stubs (or offload responsibility to JITLinkMemoryManager somehow)

  Deliverable: test cases required in the next week and basic structure needed before refactoring JITLink to use stub table manager

June 14 – June 27:

- Refactor JITLink to use global stub table manager provided through JITLInkContext
- Investigate possible cleanup on COFF dllimport table logic using global stub table manager
- Expose JITLink API to create stub from external world
- Make sure they don't break all JITLink test cases

  Deliverable: JITLink API for stub generation and refactored JITLink stub management

June 27 – July 3:

- Investigate possible refactor in ORC side to use new table manager instead of old ones
- Make sure they don't break runtime dyld world; new stub table manager requires JITLink which has somewhat limited target support compared to runtime dyld
- Prepare test cases and figure out use cases for redefinable materialization units

  Deliverable: ORC side refactor that replaces old stub table manager and test cases required for next week

July 4 – July 11:

- Implement redefinable materialization units
- Write documentations and examples on how to use redefinable materialization units

  Deliverable: redefinable units that allow ORC user to redefine symbols and documentation of how to use it

JULY 14th MID TERM EVALUATION

July 12 – July 18:

- Finish writing of redefinable materialization units
- Create a test case that demonstrate how it resolves multiple redefinition requests properly

Deliverable: Finished redefinable materialization units and verification of its functionality

July 19 – July 26:

- Implement ReoptLayer
- Figure out how to fit it cleanly into LLJIT in API sense
- Start to design reoptimization demo that put together all new components added

Deliverable: Finished ReoptLayer and design of public LLJIT API that exposes reoptimization functionality

August 1 – August 8:

- Write reoptimization demo example code that changes LLVM optimization level
- Write documentation on all new APIs

Deliverable: Concise documentations and LLJIT demo

August 9 – August 16:

- Write POC patch to clang-repl that changes c++ optimization level just-in-time

Deliverable: Real world demo that measure quality and usage of the new features

August 17 – September 5:

- Continued polishing and bug fixes.

Deliverable: Mature reoptimization support in LLVM JIT

General spec:

- Old version of function can still be usable while new version of function is compiling
    - We shouldn't free old function when re-optimized until all of its "runtime" uses of old function is finished
        - Needs an atomic counter of current executions of old function which gets incremented when function is called and decremented when the call returns.
- Reoptimization shouldn't ever block the current execution of a function; it should support multi-threaded recompilations.
    - Recompile function that gets called inside the target function will just request the recompilation to the queue and move on.
    - Tricky case: if everything is supposed to be single-threaded -> probably must block but try to not have this as default option
- Priority of re-optimized definitions is defined as FIFO
    - Simplifies the design a lot; users don't have to provide custom metrics for each reoptimized result.
    - With this rule, for cases like "my O2 compilation finished after O3 compilation is done" will be resolved in a sane manner.
- Out-of-process support
    - Reoptimization requests can be sent through RPC calls
    - Use ORC runtime and SPS for this (using ORC runtime solves all register restore/RPC issue)
- The metrics collected for reoptimization / when to start reoptimization / what optimizations are done when reoptimizing are all user-defined
    - Different clients need different reoptimization metrics / contents
        - Cling folks want to do "double to float" optimization based on a collected numerical stability profile which is something entirely out-of-tree.
        - Julia folks once tried collecting PGO metrics in unoptimized functions and do PGO optimization with them.
        - Clang-repl might want to change not only LLVM optimizer level but also clang optimizer level.
        - Have a single function that chooses to run on CPU or GPU; instrument the function by selecting GPU randomly and once it collected enough data create a

new function with “hard coded” criteria for hardware selection with all perf related code removed.
  - We will provide some default implementation but timing and contents of reoptimization itself should be highly customizable
- Prohibit global variables with static initializers
  - We can “partition” them out from the IR module just like how we do in the CompileOnDemand layer, but still very tricky because global variable definition can change or it might be removed because of global variable optimization etc.
  - A lot of traps in general, make sense to prohibit them. Main purpose is function reoptimization anyways.

Technical Proposal:

The internal of reoptimization is always the same; atomically swap the indirect function stub pointer to the newly available function. But, in order to implement this elegantly, we need to solve two fundamental issues in ORC.

First: we need to unify the management of stubs and prevent “optimization” of PLT stubs. Currently, RuntimeDyLD, ORC, and JITLink all have their own implementations of stub creations. They all have some sort of switch on target such as here. It’d be nice to unify them this time while extending them to support what we need for redefinition. Especially, in JITLink, there is no guarantee that symbols will have unique PLT stub with GOT stub pointer, meaning it’s tricky to atomically swap stub pointer to point at redefined function address. Also, JITLink does PLT stub optimization which eliminates the indirect jump using stub pointer to direct jump. This is a very nice feature but it will break the reoptimization approaches as even if we swap stub pointers, some places just jumps to the old function.

Second: we must try hard to make clear what it means to “swap the function definition.” The general philosophy of ORC is no “duplicate symbol definitions” which has been enforced to the date. Swapping the function definition is, by definition, providing another definition into JITDyLIb. What we need to change is to extend the ORC infrastructure so that it allows redefining certain symbols which are explicitly marked as redefinable. Notice redefinition problem is in some sense generalization of all “lazy” functionalities. For instance:

Reoptimization is redefining symbols on arbitrary timing.

- Lazy compilation is “blocking redefinition” where the original definition is requesting recompilation.
- Speculative compilation is “speculative redefinition.”

We have two possible proposals for solving the redefinition problem. They are implemented on different abstraction levels but achieve the same purpose. Once we have done these two groundworks, implementing reoptimization support will become very straightforward.

Therefore, the proposal is divided into three parts as following:

- Part 1: Unified stub table manager
- Part 2: Redefinable symbols
- Part 3: ReOptimizeLayer

Part 1: Unified stub table manager

We currently have stub creations inside both JITLink and ORC sides. JITLink is a better place to handle stub logics since stub creation is the main responsibility of JITLink because it deals with PLT/GOT relocations. But, JITLink doesn’t really ensure the uniqueness of stub or provide a way to change the stub pointer

- Have a new class StubTableManager which manages a global stub table per JITDyLib and provides an interface to atomically write the GOT pointer to the actual function definition per symbol.
    - This class instance is provided through JITLinkContext and held by the ORC side.
    - Should be abstract since JITLink shouldn’t know a lot about ORC
    - Only knows the pointers of stub table by symbol, the actual creation of stub will entirely reside in JITLink backend
    - Be extremely careful to not jump to an out-of-reach stub inside JITLink
        - Even though MemoryManager should have ensured every block inside the jitdylib be not too far away from the start of jitdylib symbol, it’s now more likely for out-of-reach to happen as the global stub has more chance of getting far away.
        - We don’t want JITLink to regress by introducing a global stub table.
        - Explicitly check the address of stub and if it’s out of reach, we will create “anonymous stub” to real stub as a backup plan. (maybe have an option to disable this behavior as well)
- Stub definition inside StubTableManager will have a flag “optimizable.” The ORC side will create a stub for function beforehand and set this flag false.

    - JITLink has an ability to eliminate the PLT stub sequence to direct call to the actual function when the target symbol is close to the allocated address. This optimization will break the reoptimizable stubs.
    - JITLink will now not try to optimize the PLT stub when this flag is off.
    - Note that even though this stub ends up out-of-reach, JITLink will create an anonymous stub to “real stub” not real symbol, so backup plan B will not break reoptimization.
    - JITLink must have a public API to create target-specific stub and add it to global stub table manager

Part 2: Redefinable symbols

- New materialization unit RedefinableMaterializationUnit and new mediator class RedefinableMUManager
- When RedefinableMaterializationUnit is materialized, it will create a stub using global stub table manager to the original function which will be emitted as an anonymous symbol with random name.
- RedefinableMUManager has a method “redefine()” which will swap the stub pointer of the stub to a new symbol definition.
- RedefinableMUManager will schedule the compilation jobs and resolve the multiple redefinition of the same symbol conflict in FIFO manner.
    - One possible strategy we can look into: collect the “redefinitions to do” list and query them at the same time to reduce the redundant traversal of the JITDyLib dependency graph.

Part 2: Redefinable symbols (alternative approach)

- New JITSymbolFlags: Redefinable
    - Weak and redefinable symbols are not allowed.
- Symbol state is splitted into two orthogonal states
    - MaterializationState:
        - Unmaterialized
        - Materializing (materialization begun because it was queried)
        - Materialized (materialization is done)
    - SymbolState
        - NeverSearched
        - Resolved (address assigned)
        - Emitted (emitted to memory, but waiting on transitive dependencies)
        - Ready (ready to be called)
    - When the SymbolState is ready, the symbol queries will use that symbol address regardless of materialization state (which will be address to PLT stub)
    - Symbol queries will start materialization regardless of symbol state if materialization state is Unmaterialized.
- New JITDyLib method: redefine(MaterializationUnit)
    - Error if the old symbol flag doesn't have a redefinable bit.
    - If materialization state is Unmaterialized,
        - Replace the MaterializationUnit that has been added before, we make sure that the symbol list of that new MaterializationUnit is the same as the old MaterializationUnit
    - If materialization state is Materializing,
        - Set the current MaterializationUnit ignored. Move the state to Unmaterialized, then attach the new materializer to the symbol map.
    - If materialization state is Materialized,
        - Move the state to Unmaterialized, then attach the new materializer to the symbol map.

Redefinable symbol materialization process pseudocode:

```
If (current MaterializationUnit is ignored) {
    No state transition or address assignment takes effect.
    return
}
if (it's defining the symbol for the first time) {
    Create stub using stub table manager when resolve was called
} else {
    if (new symbol flag is with redefinable bit) {
        if (symbol state is Resolved or Emitted){  // first time definition not finished yet
            Don't move the symbol state until the new function is emitted and ready
            Update stub pointer to real function address when resolve was called
        }
        if (symbol state is Ready)  {
            Don't move the symbol state keep it Ready so that they can use old functions
            Update stub pointer to real function address when the new function is emitted and ready
        }
    }
}
```

Part 3: ReOptimizeLayer

This is actually the simplest part given part 1 and part 2 were done.

- Create a ReOptLayer that marks all symbols inside the IR module as "redefiniable."
- ORC runtime function "reoptimize" will be called inside target function
- Client callback "ReoptimizeModule" gets called and it will return the new IR module
- ReOptLayer does "redefine(IRMaterializationUnit(re-optimized IR module))" and query that function.
- ORC will internally swaps the symbol pointer using global stub table