July 26th 2023

Re-optimization using JITLink

Mentors: Vassil Vassilev, Lang Hames
Student: Sunho Kim

Google
Summer of Code

Reoptimization

- Choose low optimization level to minimize compile time, identify few “hot” functions, then recompile it with higher optimization level.
- One of most exciting feature of JIT compilers since it offers trade-off between flexibility and performance.
- My project is to extend LLVM’s ORC API to support flexible reoptimization that fulfills needs of various projects

Motivation for reoptimization

- CERN folks wants to do real time “double to float” optimizations to balance the power consumption and numerical stability.
- Long-term profile guided optimization for clang-repl or cling.
- Opens door for many runtime optimizations for LLVM based languages.

JITLink

- 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
- Designed to support full features of AOT compiled object files
- Have linker object abstraction called LinkGraph and plugin system that supports adding “LinkGraph transform” passes.
- In general, more stable and robust than previous JIT linker RuntimeDyld.

Requirements

- When to run optimization or optimization itself must be user-defined.
  - Many different use cases exist and they are all some sort of engineering problems that need their own approaches.
  - Make sure that API is easy to use so that it causes minimal hassle.
- Multi-threading, remote executor process, and laziness support.
  - These are the selling points of ORC API; we don’t want to lose them because of reoptimization.
- Must be compatible with the latest JIT infrastructure.
  - A lot of progress happening around new JITLink infrastructure; the future is there.

Roadmap

- Part 1: Refactor the JIT infrastructure to better support “redefinition” of symbols using JITLink.
- Part 2: Implement reoptimization infrastructure that processes reoptimization requests.
- Part 3: Create a cool clang-repl demo that showcases the reoptimization capability.

Part 1: Refactor the JIT infrastructure

- Many JIT features including reoptimization rely on the “symbol redefinition” problem.
  - Lazy JIT is essentially redefining symbol with the actual definition of function when the lazy compilation was triggered.
  - Speculative compilation is redefining symbol eagerly.
  - Reoptimization is redefining the symbol with the “reoptimized” definition.
- Right now multiple internal implementations exist to solve the same issue.
- The most established method live in IndirectionUtils.h
  - Pretty capable but it doesn’t utilize new JIT infrastructure and tricky to use it for reoptimization.
  - Only supports the stubs approach where it create a “stub” symbol with the actual symbol name that jumps using function pointer that can be atomically rewritten.

Part 1: Refactor the JIT infrastructure

- **Proposal:** Create RedirectionManager abstraction.
- It has two methods: createRedirectableSymbol and redirect.
- createRedirectableSymbol will define the symbol that can be redirected and redirect will change the “dummy” symbol to direct the new “impl” symbol.
- Support both JITLink and RuntimeDyLD.

Part 2: Implement reoptimization

- **Proposal:** Create ReoptLayer that processes the reoptimization request.
- ReoptLayer can be used to add reoptimizable materialization unit which will create redirectable symbol.
- When user wants to trigger reoptimization, call reoptimize function to request reoptimization.
- Possibly, support a handy way to keep track of common profiles such as how many times the function has been ran.
- Many design decisions to make: How to make it thread-safe? Can we batch the reoptimization request? Should we just receive the defined symbol or IR module? How do we clean up the “old” impl symbols?

Part 3: Create a cool clang-repl demo

- Brainstorming right now. Currently have two ideas:

1. Standard reoptimization demo that optimizes function from -O0 to -O3 when the function gets ran a lot.

2. Advanced profile guided optimization demo that does PGO real time while trying to suppress the profiling overhead by turning off profiling at certain interval.

Roadmap

- Part 1: Refactor the JIT infrastructure to better support “redefinition” of symbols using JITLink. (submitted patches)
- Part 2: Implement reoptimization infrastructure that processes reoptimization requests. (~mid-August)
- Part 3: Create a cool clang-repl demo that showcases the reoptimization capability. (~early-September)