GSOC report

Information

Project: Re-optimization using JITLink

Organization: LLVM compiler infrastructure

Contributor: Sunho Kim

Background

For initial spec and background you can see the original proposal pdf here:
https://compiler-research.org/assets/docs/Sunho_Kim_Proposal_2023.pdf

Goals

- To extend JITLink API and ORC runtime to support re-optimization use cases
- To design and implement ORC API for customizable re-optimization
- To enable re-optimization in clang-repl, LLVM's C++ JIT interpreter
- To implement a profile guided optimization using re-optimization API
- To write a clang-repl demo that showcases the re-optimization feature

Report

Extending JITLink API

In order to implement re-optimization, we need an ability to "redirect" call to certain symbol into another. We decided that one of the most straightforward way to achieve this is through creating "stubs" which will jump to certain code address by using its modifiable function pointer. (this function pointer is atomically swapped to new address when the redirection occurs)

I extended the JITLink API by adding cross-architecture stub creation API. This API works in all platforms and architectures that JITLink supports and through this we can create the redirectable stubs through by using JITLink.

Designing and implementing ORC API for re-optimization part 1: Redirection API

We introduced new "`RedirectionManager`" abstraction that redirects symbol to another. This `RedirectionManager` is used to replace the call to old version of function into new version of function. We realized that when stubs are used to implement redirection, we actually need to materialize stub symbols with the same name as the original symbol.

Thus, the `RedirectableSymbolManager` was introduced to also abstract the creation of "redirectable" symbols. Theoretically, we only need `RedirectionManager` to implement re-optimization if we actually rewrite the call instructions to point at the new version of function. But, for now, we decided to keep it simple since instruction rewriting is a lot more complicated than the stubs approach. Below is the interfaces of `RedirectionManager` and `RedirectableSymbolManager`.

```
/// Base class for performing redirection of call to symbol to another symbol in
/// runtime.
class RedirectionManager {
public:
  /// Symbol name to symbol definition map.
  using SymbolAddrMap = DenseMap<SymbolStringPtr, ExecutorSymbolDef>;

  virtual ~RedirectionManager() = default;
  /// Change the redirection destination of given symbols to new destination
  /// symbols.
  virtual Error redirect(JITDylib &JD, const SymbolAddrMap &NewDests) = 0;

  /// Change the redirection destination of given symbol to new destination
  /// symbol.
  virtual Error redirect(JITDylib &JD, SymbolStringPtr Symbol,
                         ExecutorSymbolDef NewDest) {
    return redirect(JD, {{Symbol, NewDest}});
  }

private:
  virtual void anchor();
};

/// Base class for managing redirectable symbols in which a call
/// gets redirected to another symbol in runtime.
class RedirectableSymbolManager : public RedirectionManager {
public:
  /// Create redirectable symbols with given symbol names and initial
  /// desitnation symbol addresses.
  Error createRedirectableSymbols(ResourceTrackerSP RT,
                                  const SymbolMap &InitialDests);

  /// Create a single redirectable symbol with given symbol name and initial
  /// desitnation symbol address.
  Error createRedirectableSymbol(ResourceTrackerSP RT, SymbolStringPtr Symbol,
                                 ExecutorSymbolDef InitialDest) {
    return createRedirectableSymbols(RT, {{Symbol, InitialDest}});
  }

  /// Emit redirectable symbol
  virtual void
  emitRedirectableSymbols(std::unique_ptr<MaterializationResponsibility> MR,
                          const SymbolMap &InitialDests) = 0;
};
```

Something to note here is that we have a "`emitRedirectableSymbol`" method. This was needed since when we are "replacing" the original symbols with redirectable stub symbols, the `MaterializationUnit` is already emitting the original symbol that we can't simply let "`RedirectableSymbolManager`" to redefine the symbol that MU is responsible for. In order to support this kind of use case, we decided to expose raw

emission process to the public and `createRedirectableSymbol` is simply a wrapper function that uses `emitRedirectableSymbol` method to define new stub symbols.

Then, I wrote `JITLinkRedirectableSymbolManager` class which implements `RedirectableSymbolManager` using JITLink API. It actually pools stubs and create stubs in batch since the overhead of create `LinkGraph` for each stub might be costly.

Designing and implementing ORC API for re-optimization part 2: ReOptimizeLayer

After we have done redirection API, actual implementation of re-optimization began. A new `IRLayer` `ReOptimizeLayer` was introduced to support re-optimization of IR modules. There were many abstraction levels where redirection could be implemented, but I ended up doing it at IR level since that brings a lot of re-optimization techniques to be implemented easily by transforming IR directly. From API perspective, the most flexible abstraction level to do this would be at FrontEnd AST level. In the future, we might extend the API to support custom `MaterializationUnit`--not just `IRMaterializationUnit`--in order to give clients power to do any arbitrary re-optimization using their favorite intermediate level.

`ReOptimizeLayer` simply takes `IRMaterializationUnit` and replace those with redirectable symbols using `RedirectableSymbolManager`. It sets the initial destination of redirectable stubs to point at function address that got emitted by next IR layer. (e.g. `IRCompilerLayer`)

It transforms the initial version of function to have a "re-optimization request" code that simply calls orc-runtime __orc_rt_reoptimize function. For instance, the original IR function below:

```
define dso_local noundef i32 @_Z1fv() #0 {
entry:
  ret i32 5
}
```

gets transformed to this:

```
define dso_local noundef i32 @_Z1fv() #0 {
entry:
  %0 = load i64, ptr @__orc_reopt_counter, align 8
  %1 = icmp eq i64 %0, 20
  %2 = add i64 %0, 1
  store i64 %2, ptr @__orc_reopt_counter, align 8
  br i1 %1, label %3, label %4

3:                                                ; preds = %entry
  call void @__orc_rt_reoptimize(i64 3, i32 0)
  br label %4

4:                                                ; preds = %entry, %3
  ret i32 5
}
```

Enabling re-optimization in clang-repl

It didn't take a lot of efforts to enable re-optimization in clang-repl. I first enabled orc runtime in clang-repl since it would be handy to implement advanced profile guided optimizations. There was a little mismatch what clang-repl expects from how runtime runs static initializers and how ELF orc runtime runs it. "dlopen" on library ran static newly installed initailizers before but new orc runtime decided to not run them. We're still discussing what to do about this. My mentor Lang suggested to add a new dl function which made sense to me but Lang also wants to look at alternatives as well.

Other than that, it was matter of just adding few layers to `LLJIT` instance. For enabling new layers, I proposed new `LLJIT` API to do it. Previously, we had `LLLazyJIT` which had lazy compilation enabled by adding `CompileOnDemandLayer` to the stack. But, now we got another layer to the party, it would be not a good design to introduce `LLReOptJIT` class since we might also want "`LLLazyReOptJIT`" and so on.

I proposed `LLLayerJI` API which allows something like below.

```
LLLayerJIT LayerJIT;
LayerJIT.addLayer(std::make_unique<LLReOptimizeLayer>()); // Add re-optimization
LayerJIT.addLayer(std::make_unique<LLCompileOnDemandLayer>()); // Add lazy-compilation
```

I haven't polished this API enough to land in tree yet but I think it's a really cool API design somewhat resembling tensorflow's Keras API.

Since clang-repl rely entirely on LLJIT to do JIT stuffs, getting re-optimization running for LLJIT was all we needed to enable it in calng-repl. Now, we got a nice real-world experiment environment where we can test new re-optimization techniques and do benchmarks to see if they are win.

Implement profile guided optimization

Now fun time begins :) Since I got all the infra needed for re-optimization techniques, I could implement any kind of profile guided optimization by transforming IR module freely.

First thing I tried was going from -O0 to -O2 when function call count exceeds 10. In some of the cases I tested, this offered a nice trade-off between time spent on compilation and runtime. For instance, one testcase ran in 0.6 seconds with -O2 and 0.4 seconds with -O0. With reopt on, it ran in 0.5 seconds--just in the middle of the two.

Second thing I tried was de-virtualization PGO which instruments indirect function calls to get which function addresses that indirect call jumps to and inline them when re-optimization happend. I haven't polished the code to be upstreamable yet but it implements instrumentation on orc-runtime side which simplifies implementation a lot. (here is the branch that implements this:) In some of the cases, this can get up to 10% gain compared to non reopt version. There are regressions cases since the instrumentation has huge cost though. So, when the correct function was non re-optimized it has lead to huge performance drop. We do need to look into them further.

Future goals

- Look into optimizing function with a huge loop up front. The penalty we get when we couldn't re-optimize certain function are substantial.

- Look into instruction rewriting redirection manager. The indirect calls from redirectable stubs can gives some performance drop as much as 5% from my observations.
- Land more codes -- this is something I'd do it soon even after GSOC ends officially.

Links to the patches and PRs

- https://reviews.llvm.org/D155557
- https://reviews.llvm.org/D157378
- https://github.com/llvm/llvm-project/pull/66802
- https://github.com/llvm/llvm-project/pull/66812
- https://github.com/llvm/llvm-project/pull/67050
- A lot of unpolished over 1000 lines of additions here: https://github.com/sunho/llvm-project/tree/karikari

Final Words

I'm really grateful for my GSOC mentors for being such flexible on schedule even to extend the final deadline one month. They also provided enormous help on getting the right design as well as suggesting various interesting ideas to try out that I was not aware of.