Advanced symbol resolution and reoptimization for clang-repl

Name: Sahil Patidar
Email: sahilpatidar60@gmail.com
Github: https://github.com/SahilPatidar
Mentors:

- Vassil Vassilev

Size of Project: Large

1. Project Description:

Clang-REPL is an incremental compiler for C and C++ that extends the traditional static compilation model, enabling dynamic execution while retaining the performance benefits of C and C++. It allows interactive programming similar to Python but with the power of C and C++.

Currently, in Clang-REPL, users must manually load dynamic libraries when external symbols are required. This project aims to enhance Clang-REPL by implementing an **advanced symbol resolution mechanism** that **automatically loads the necessary dynamic libraries** when external symbols cannot be resolved.

Additionally, the project will introduce **runtime profile-guided optimizations (PGOs)** to **re-optimize executable binaries dynamically** based on profiling data, ultimately improving performance.

This project will enhance Clang-REPL's usability and performance, making it more efficient for interactive and dynamic C/C++ execution.

Project Objective:

- Implement an ***Auto-Load mechanism*** for Clang-REPL to automatically load missing dynamic libraries when resolving external symbols.

- Introduce Re-Optimization support to enhance runtime performance using ***Profile-Guided Optimizations (PGOs)***.

Expected Outcomes:

- A ***robust Auto-Loading mechanism*** for seamless symbol resolution in Clang-REPL.
- Re-Optimization support leveraging runtime profiling to improve execution performance.

Technical Skills: C/C++, LLDB, Familiar with LLVM ORC-JIT, LLVM IR, Git

2. Implementation Plan

2.1. Roadmap

Since out-of-process execution environments introduce the most overhead, optimizing them first ensures that the approach will also work efficiently for in-process execution. This project enhances **symbol lookups in ORC-JIT** by integrating **Bloom filters**, reducing redundant RPC calls, and improving execution efficiency in **both per-library and auto-loading modes**.

Phase 1: Controller Side

On the controller side, each dynamic library has its own orc::DefinitionGenerator attached to a JITDylib, which is responsible for resolving definitions for unresolved symbols in the dylib. To enable auto-loading, we need to either extend or create a new DefinitionGenerator specifically designed for auto-loading use cases. Additionally, to

support efficient symbol lookup, we need to integrate Bloom filter support by introducing a new **FilterListener API**that can work alongside the DefinitionGenerator. This outlines the main work that needs to be done.

- **Auto-Loading Mechanism**
  - Integrate auto-loading with `DefinitionGenerator` to avoid redundant complexity.
  - Introduce a **unique handle** for auto-loading recognition in lookup APIs.

- **Bloom Filter API & Request Strategy**
  - Develop a **FilterListener API** that integrates with **DefinitionGenerator** and **Lookup API** to manage **filter requests**.
  - Define a **request strategy** based on lookup mode.
  - Ensure seamless compatibility with ORC's **search mechanisms**.

Phase 2: Executor Side

On the executor side, symbol lookup is handled by `orc::SimpleExecutorDylibManager`, which is responsible for resolving unresolved symbols in a dylib using its dynamic library handle. To support auto-loading, we need to either enable or create a new `ExecutorResolver` API that can handle auto-loading seamlessly. Additionally, we need to design a **FilterBuilder** API that can build Bloom filters for both lookup modes — **per-library** and **global-library filtering**— and efficiently report them to the controller. This will be an important part of making lookups faster and more scalable.

1. Implement Auto-Loading Mechanism

- Dynamically scan **shared libraries** to collect and store symbol information.
- Enhance Orc's **symbol resolution** to track and register dynamically loaded libraries.
- Manage **LoadedLibraries** set to optimize future lookups on the executor side.

2. Develop Bloom Filter API and Filter Construction

- Design a **Bloom Filter API** that allows the executor to report filters to the controller, supporting both **per-dylib filters** and a **global Bloom filter**.

- Ensure the design routes filter updates to the corresponding `FilterListener` associated with each `DefinitionGenerator`.
- Construct **per-library** and **global Bloom filters** based on the request strategy.
- Build **global filters** that aggregate symbols across multiple libraries.
- Optimize **filter updates** using a combination of on-demand and proactive approaches.

2.2. Controller Side

Bloom Filter API

To optimize symbol lookups in both **auto-loading** and **per-dylib** modes, we need a **FilterListener API** in the controller. This API will handle **filter requests** from both modes and send requests to build filters. To build a filter, we must pass a **dylib handle** to identify which specific dylib's filter is required, as well as an identifier for the **global filter**.

Key Components of Bloom Filter API

1. **Filter Requests:**
    - The controller must be able to request a filter for:
        - **A specific dynamic library (per-dylib mode)** → Requires the **dylib handle** for identification.
        - **Global filtering (auto-loading mode)** → Requires identification for a **global filter** across all libraries.
2. **Challenges in Request Timing:**
   Determining the optimal time to request a filter from the executor depends on how the **FilterBuilder** manages filter creation.

In **per-dylib mode**, the approach is more straightforward: a filter should be requested **immediately** when searching a single library. Without using a filter in this mode, resolving symbols would require **N separate RPC calls for N libraries**, leading to significant inefficiency. Promptly requesting the filter avoids this overhead and streamlines symbol resolution.

Global Filter in Auto-Loading Mode:
Requesting a global filter in auto-loading mode is more complex because it must process a large set of library symbols. There are two possible strategies to handle this.

The first is **preemptive filter creation**, where all libraries are scanned at initialization, symbols are collected, and a request is sent to the executor to build the filter in advance. This approach ensures fast symbol lookups during execution but incurs **high**

upfront costs. To improve this, an **asynchronous approach** could be considered—scanning libraries in the background without blocking execution.

The second strategy is **on-demand filter creation**, where the filter is requested **only when the executor fails** to resolve a symbol in auto-loading mode. This failure indicates that **all libraries have already been searched**, making it the right time to build and return a global filter for future lookups. Since **only one RPC call is made per symbol lookup** in auto-loading mode, this method ensures that the filter is built **only when absolutely necessary**, thereby minimizing unnecessary RPC communication.

Possible Request Strategy Based on Mode

- **Per-Dylib Mode:**
  - The filter is needed **from the beginning**, as each lookup would otherwise require multiple RPC calls.
- **Auto-Loading Mode:**
  - The filter is only required **after an unresolved lookup** since only one RPC call is made per lookup.

By integrating these approaches, the **Bloom Filter API** will enhance performance by reducing redundant RPC calls and ensuring efficient symbol resolution.

Auto-Load calling mechanism:

Enabling Auto-Loading with a Unique Handler

To support **Auto-Loading**, the mechanism must integrate smoothly with the **existing DefinitionGenerator-based approach**, rather than introducing a completely new search method.

Key Challenges:

- **DefinitionGenerators** handle single-library lookups, but **Auto-Loading** must efficiently manage a **large set of libraries**.
- Creating a new DefinitionGenerator for Auto-Loading would add unnecessary complexity.

Proposed Solution by Lang Hames:
Introduce a Unique Handle for Auto-Loading

- Define a **dedicated unique handle** distinct from standard dynamic library handles.
- This handle will be recognized by **Search and Lookup APIs** to **trigger Auto-Loading**.

2.3. Executor Side

Auto-Loading Mechanism

The **Auto-Loading mechanism** dynamically loads shared libraries when the system encounters unresolved symbols during execution. This enhances usability by **automating symbol resolution** without requiring manual intervention.

Current Symbol Resolution in Clang-REPL (ORC-JIT)

Clang-REPL leverages **ORC-JIT** for incremental compilation and execution. Symbol resolution in **JITDylib** is managed by **DefinitionGenerators**, where:

- Each **loaded library** has its own **DefinitionGenerator**.
- **DefinitionGenerators** interact with **DylibManager API** to:
    - Load libraries.
    - Retrieve dynamic library (dylib) handles.
    - Pass the handle to the **DylibManager API** for symbol resolution.

Cling's Auto-Loading Search Mechanism Approach

Our approach will leverage **Cling's fallback search mechanism** for efficient symbol resolution.

How Cling's Auto-Load Search Works

1. **Library Scanning & Registration**
   Cling starts by scanning all available libraries and registering them into a container. This gives it a list of libraries it can search through when looking for symbols.
2. **Symbol Lookup Process**
   When Cling needs to find a symbol, it goes through the registered library information to check if the symbol exists. As it searches, it creates **Bloom filters** on-the-fly for each library. These filters help speed up future lookups by quickly telling whether a library might contain a symbol or not.
3. **Efficient Caching with QueriedLibraries**
   If Cling finds the symbol, it returns the path to the library and saves the library

info into a separate container called **QueriedLibraries**. The next time it needs to find a symbol, it checks **QueriedLibraries** first, so it doesn't waste time searching the same libraries again. It also removes any libraries that have already been loaded from the main scanned container, making future searches even faster.

To improve efficiency and better support global symbol lookups, we introduce a few refinements. First, we maintain a separate **LoadedLibraries Info set** for libraries that have been queried and successfully loaded. This allows us to use individual **Bloom filters** for faster lookups within already loaded libraries. It also enables the construction of a **global Bloom filter** that includes symbols from both **LoadedLibraries** and **unloaded scanned libraries**, covering both system and user libraries.

Auto-Loading Process

The auto-loading process begins with an initial lookup where the system searches for the symbol across all currently loaded dynamic libraries. If the symbol is not found, a fallback mechanism triggers Auto-Loading. At this point, the system scans the available search paths for libraries that have not yet been loaded, including standard system library paths like `/usr/lib` and `/lib`, as well as any user-defined library directories. Once discovered, these dynamic libraries are registered into a container for efficient future access. The system then iterates through the registered libraries, checking each one for the required symbol. If the symbol is found, the corresponding library is loaded dynamically, the symbol is retrieved, and the library is added to the **LoadedLibraries** set. This not only speeds up future lookups but also contribute to the construction of the **global Bloom filter**. for even more efficient symbol resolution going forward.

BloomFilter API

Need for Bloom Filter in Symbol Lookup APIs

In an out-of-process execution environment, resolving unresolved symbols in dynamic libraries can quickly become inefficient, especially in **per-library lookup mode**. Here, Each `ORC::DefinitionGenerator` attached to a JITDylib must make an RPC call for symbol lookup. If there are **N** loaded libraries and the required symbol happens to be in the **(N-1)th** library, the system will end up making **N-1** RPC calls before successfully resolving it. This overhead stems from the fact that there is currently no way to predict whether a symbol exists in a particular library without first attempting a lookup.

Similarly, in **auto-loading (global-library lookup) mode**, a single RPC call is made, but the system must search through all libraries. If the symbol is missing in all libraries, this results in unnecessary overhead, significantly impacting runtime performance.

To address these issues, we propose introducing a **Bloom filter into** both **auto-loading mode** and **single-library lookup mode**. The goal is to provide a lightweight, probabilistic way to quickly check if a symbol might exist in a library before making a costly lookup attempt. The main challenges are designing a solution that works efficiently across both lookup modes, supports both **in-process** and **out-of-process** execution environments, and figures out the right timing and strategy for building Bloom filters at both the **global** and **per-library levels**.

Design Considerations

1. When to Create and Request the Bloom Filter

The main challenge is determining **when** to construct the Bloom filter and **when** the controller should request it from the executor.

- **Per-library Bloom filter:**
  - Straightforward to implement but must support both execution modes.
- **Global Bloom filter (Auto-loading mode):**
  - Needs to aggregate symbols from multiple libraries.
  - Can be expensive when dealing with a large number of libraries.
  - Requires a mechanism to efficiently collect and process symbols.

FilterBuilder API

The **FilterBuilder API** is responsible for processing filter requests from the controller and constructing Bloom filters for both **per-dylib** and **global-dylib (auto-loading mode)** lookups.

The controller may pass a **dylib handle** to indicate whether a filter is needed for a **specific library (per-dylib)** or for **all libraries (global-dylib)**.

Key Responsibilities

1. **Per-Dylib Mode:**

- Maintain a mapping of **loaded dylib handles** to **library information**.
- Use this mapping to efficiently build a filter for **individual libraries** when requested.

2. **Global-Dylib Mode (Auto-Loading):**

    - This is more complex as it involves managing **all scanned and loaded libraries**.
    - Leverage the **DynamicLoader API** used in the **auto-loading mechanism** to track library metadata.
    - Ensure that the filter can be efficiently built while minimizing performance overhead.

By integrating with the **existing dynamic loading infrastructure**, the FilterBuilder API will enable efficient symbol lookup optimizations.

Approaches for Building the Bloom Filter

1. **After a Failed Symbol Resolution:**

    - If symbol resolution fails and control returns to the controller, it indicates that all libraries have been searched.
    - At this point, we can request the executor to build the **global Bloom filter**, ensuring that subsequent lookups are optimized.

2. **On-Demand Request from Controller:**

    - The controller can request the Bloom filter from the executor.
    - If the filter is **not yet available**, the executor adds the request to a **pending queue**.
    - If a lookup failure occurs, the system would have already collected all symbols, making it an ideal time to construct and return the filter.

3. **Proactively After Library Scanning:**

    - During initialization, libraries are scanned, and their metadata is collected.
    - We iterate over library information (initially containing only paths) and extract symbols into a **global symbol set**.
    - This symbol set is then used to construct the **global Bloom filter**, reducing overhead during actual lookups.

Integrating Bloom filters in symbol lookup APIs will **reduce unnecessary RPC calls**, improving lookup efficiency and runtime performance. The key to success lies in carefully balancing the cost of constructing the filter with the performance gains it provides. The final design should support both **per-library** and **global** filtering strategies while being adaptable for both in-process and out-of-process execution environments.

2.4. Re-Optimization in Clang-Repl

Re-optimization is a technique in **ORC-JIT** that allows optimizing hot functions at runtime based on profiling data, producing a more optimized version of the function. ORC already provides this capability through the ReOptimizeLayer, but **Clang-Repl** currently lacks support for it. Our goal is to integrate **Re-Optimization** into **Clang-Repl** to leverage its performance benefits.

Tasks to be Completed

- **Add `--re-opt` flag**: Introduce a command-line option in clang-repl to enable re-optimization.
- **Implement `ReOptFunc`**: Define a custom **ReOpt function** that will be passed to ReOptimizeLayer. This function will use runtime profiling data to apply targeted optimizations and LLVM passes for improving performance.
- **Utilize `RedirectableManager`**: We can initially use JITLinkRedirectableManager for handling function redirection during re-optimization.

This integration will enable **Clang-Repl** to dynamically re-optimize frequently executed functions, improving execution efficiency over time.

3. Timeline: The expected timeline of the project is as follows:

Time	Task	Deliverable
2 June – 8 June	• Implement the Controller side API for auto-loading. • Introduce a unique handle for managing auto-loading. • Implement new	`ExecutorResolutionGenerator` Implementation.

	`DefinitionGenerator` for auto-loading	
9 June – 22 June	• Implement the `RemoteResolver` API for executor-side auto-loading. • Add Auto-Loading Search, inspired by Cling's approach. • Implement library scanning to extract symbol information from dynamic libraries. • Develop a caching mechanism to store previously queried libraries for future lookups. • Introduce a LoadedLibraries set to facilitate global Bloom filter construction. • Efficiently aggregate symbols to construct a global Bloom filterfor auto-loading. • Begin testing symbol collection efficiency.	`RemoteResolver` implementation.

23 June – 6 July	• Develop the `FilterListener` API in the controller to handle filter requests.
	• Implement the FilterBuilder API for constructing per-dylib and global Bloom filters.
	• Integrate Bloom filtering into the Lookup API for efficient symbol resolution.
	• Enable the executor to transmit Bloom filters to the controller for future lookups.
	• Optimize lookup efficiency:
	  ◦ Single-library search: Avoid redundant RPC calls if a symbol is absent.
	  ◦ Auto-loading mode: Minimize exhaustive searches across all libraries.
	• Conduct unit tests to validate API functionality.
		BloomFilter API and initial integration.

7 July – 13 July	• Complete integration of the Bloom Filter mechanism. • Resolve any issues identified during initial testing.	Fully integrated BloomFilter API.
14 July – 20 July	• Begin implementing re-optimization support in clang-repl. • Introduce the `ReOptimizeLayer` in LLJIT to support dynamic re-optimization.	—
21 July – 27 July	• Implement **command-line support** for re-optimization, e.g., --re-opt in clang-repl. • Develop a **handler** for the --re-opt flag to configure LLJIT accordingly. • Implement **custom** `ReOptFunc`, which will be passed to the `ReOptimizeLayer`.	—
28 July – 3 Aug	• Finalize the implementation of **re-optimization support** in clang-repl. • Address any integration issues found during testing.	Fully functional re-optimization support in clang-repl.
4 Aug – 17 Aug	• Conduct **testing and benchmarking** of the new **auto-loading and Bloom Filter-based symbol resolution**. • Compare performance against traditional symbol resolution methods on various code samples.	Benchmark results showcasing performance improvements.
18 Aug – 25 Aug	• **Document** the project implementation, API usage, and benefits. • Write a **detailed blog post** explaining the work, design choices, and performance improvements.	Project documentation and blog post.

4. About me:

- **Background and Motivation:** I am interested in compiler development and system programming. I have contributed to the LLVM project, mainly working on optimization passes and small improvements related to ORC-JIT. Recently, I've been exploring Clang-REPL and ORC-JIT in more depth. I'm still learning, but I enjoy reading the codebase, experimenting with small patches, and understanding how different parts fit together.

  I am applying to GSoC because I want to learn by working on a real project. The project on ORC auto-loading seems like a great opportunity for me to dive deeper into runtime symbol loading and JIT internals.

  Through this experience, I hope to improve my coding, debugging, and open-source contribution skills.

- **Open Source Experience:** I already have hands-on experience with the LLVM compiler infrastructure, having actively participated in the project by submitting several patches. These patches demonstrate my understanding of LLVM's codebase and my ability to contribute meaningful improvements.

  Specific Contributions:
  https://github.com/llvm/llvm-project/pulls/SahilPatidar

5. Availability

I have no prior commitments this summer, so I will be available full-time.