On Demand Parsing in Clang

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

- Vassil Vassilev
- Matheus Izvekov

Size of Project: Large

1. Project Description:

This proposal outlines a project to implement on-demand compilation within Clang, a widely used C++ compiler. Traditional compilation in Clang processes the entire codebase at once, regardless of actual usage. This approach leads to high peak memory usage and slow compilation times, especially for large projects and interactive development.

Project Objective:

This project seeks to develop a mechanism within Clang that allows for the parsing of C++ entities (functions, classes) only when they are used in the code. This approach aims to significantly improve the efficiency of the compilation process by:

- **Reducing Peak Memory Usage:** By deferring parsing of unused entities, the memory burden during compilation will be considerably lower.
- **Enhancing Compile Times:** Focusing compilation efforts on utilized elements will lead to faster overall compilation speeds.
- **Optimizing Interactive C++ Development:** Header inclusions will have minimal impact as parsing of entities will occur only when required, fostering a more responsive development experience.

Expected Outcomes:

- **On-demand parsing for non-templated functions & structs/classes:** Parse only what's used, reducing memory usage and compile times.
- **Performance benchmarks & report:** Quantify improvements on various codebases.
- **Community RFC document:** Introduce on-demand parsing to the Clang community.

- **[Stretch Goal] Template support:** Enable on-demand parsing for templates (ambitious).

Technical Skills: C/C++, LLDB, Familiar with LLVM IR, x86, Git

2. Implementation Plan:

- **Plan and challenges:**
  1. **Token Stream Management**: If every lazy parsed entity records its token stream, managing these token sequences efficiently could be challenging. This includes ensuring that tokens are correctly re-positioned or referred to when needed, and handling cases where modifications to the token stream are required (such as appending an "eof" token).

    - To implement lazy entity definition, we propose attaching the token stream of the definition to its corresponding Declaration using a Map. This approach allows us to efficiently retrieve the token stream associated with a Declaration via a LookupResult. Once we obtain both the Declaration and its token stream, we can append a `tok::eof` token to mark the stream's end. Finally, the complete stream is passed to the Preprocessor class using the `EnterTokenStream` method.

  2. Function Handling with Lazy Definition Parsing: We propose a mechanism for lazy parsing of function definitions, similar to the approach used by `LateTemplateParser`. Here's how it would work:

  The proposed approach focuses on two key stages:

  1. **Skipping Function Definitions During Parsing:**
     - We will modify the ParseFunctionDefinition method to efficiently identify functions eligible for lazy parsing.
     - The HandleDeclarator method will be leveraged to create a preliminary declaration for the function.
     - Redefinition checks will be implemented to ensure the function is not being redefined.
     - If the function qualifies for lazy parsing, ActOnSkippedFunctionBody will notify Sema (Semantic Analysis)

and store the function's body token stream attached to the declaration (Decl).

2. **Handling Function Usage with Lazy Parsing:**
    - Clang currently annotates names with declarations during expression parsing using SemaLookup.
    - A filter will be applied to the LookupResult during lazy definition mode to identify the appropriate declaration for a function identifier based on reachability using Source Location.
    - ActOnCallExpr will build the call expression with the relevant declaration using argument type information.
    - Upon encountering a call to a function with a previously stored lazy definition, Clang will enter lazy parsing mode.
    - ActOnStartOfFunctionDef will signal Sema to parse the definition, and ActOnFinishFunctionBody will notify Sema after parsing is complete.
    - Once parsed, the lazy-parsed information (e.g., function body token stream) associated with the declaration will be removed.

3. Lazy Parsing for Classes and Structs:

Similar to our approach with functions, we propose implementing lazy parsing for class and struct definitions.

1. **Skipping Class/Struct Definitions:**
    - In the parseClassAndStruct method of the parser, the ParseClassSpecifier function will be modified to incorporate a mechanism for efficiently skipping class or struct definitions.
    - A helper method will be introduced to determine if a definition can be skipped based on specific criteria.
    - If a definition is skippable, a declaration will be constructed using ActOnTag.
    - Redefinition checks will be implemented to ensure the constructed declaration isn't a duplicate.
    - Before skipping the definition, Sema will be notified via SkipTagDefinition.
    - The definition's token stream will be attached to its declaration for potential future use.
2. **Lazy Parsing of Class/Struct Definitions:**

- During expression parsing, when SemaLookup is used to annotate names, a filter will be applied to the LookupResult if lazy definition mode is active. This filter will ensure only relevant declarations are considered.
- Once the appropriate declaration(s) are identified, a search will be conducted within previously stored lazy definition information.
- If a declaration has a lazy definition, Clang will be instructed to enter lazy parsing mode.
- Sema will be configured to parse the definition using ParseCXXMemberSpecification.
- After parsing the lazy definition for the declaration, the lazy-parsed information (e.g., class body token stream) will no longer be required and will be removed from the corresponding storage map.

4. Efficient Lookup Tables: When parsing lazy definitions in Clang, one of the key challenges is efficiently obtaining the appropriate declaration within the current parsing scope. This is particularly important for cases where there are multiple declarations with the same or different name but different scopes or visibility.

The Scope Issue and Potential Consequences

Imagine a scenario where you have several functions named calculateArea scattered throughout your codebase. When parsing a lazy definition that references calculateArea, the parser needs to determine the exact calculateArea declaration intended within the current context. Selecting the wrong one could lead to errors or unexpected behavior.

Two Effective Approaches for Efficient Lookup

To tackle this challenge, Clang offers two effective approaches that leverage source locations:

1. **Post-Lookup Filtering:**
    - Clang first performs a lookup for the DeclarationName (e.g., calculateArea). This yields a LookupResult containing all declarations that match the name, potentially including those outside the current scope.
    - Clang then iterates through the LookupResult and employs SourceLocation information associated with each declaration. By comparing the SourceLocation of each declaration with the current

parsing scope's SourceLocation, Clang can effectively discard declarations that fall outside the relevant scope. This ensures that only declarations within the current context are considered, promoting accuracy.

2. **Modified** CppNameLookup **for In-Line Filtering (Alternative Approach):**
    - This approach proposes modifying the CppNameLookup class itself to integrate source location filtering directly during the lookup process.
    - By incorporating this filtering logic within CppNameLookup, Clang could potentially return a more focused set of declarations that are already scoped appropriately. This could streamline the parsing process and potentially improve performance by reducing the need for post-processing the LookupResult.

5. Handling enclosing declaration: Declaration into the enclosing context when a struct or class declaration appears inside the class.

- We could introduce a new mode called `LazyParsingMode` . If enabled , this mode would only skip the parsing of lazy entity definitions. However, this mode would only be active when parsing scopes that are either a Translation Unit, a File, or a Namespace.

6.Handling Scope Switching for Lazy Parsing: One challenge arises when parsing a local expression that uses a class with lazy parsing entities. To parse the definition of such an expression, we must return to the file or namespace scope. However, once this parsing is complete, we need to reset it.

Solution: RAII-based Scope Management
To address this challenge, we propose introducing a new RAII class for managing scope changes during lazy parsing. This class will encapsulate the logic for entering and exiting the file or namespace scope.

- Enter Method: This method will be responsible for switching the scope from the current context to the file or namespace scope.
- Exit Method: This method will ensure that the scope is reset back to the previous context after parsing is complete.

By employing an RAII class, we can guarantee proper scope management and avoid potential issues arising from lingering scope changes.

- **Timeline:** The expected timeline of the project is as follows:

  Week 1:

  - **Task:** Implement a lazy entity parsing interface. This includes:
    - Storing token stream methods for Declaration.
    - Retrieving token stream methods for Declaration.
    - Initiating the preprocessor using `EnterTokenStream` for the lazy token stream.
  - **Deliverable:** Interface methods for storing and retrieving token streams for Declaration.

  Week 2:

  - Continue work from Week 1 if necessary, and buffer time for debugging any problems
  - Begin implementing the lookup filter. It is responsible for eliminating unreachable declarations in the LookupResult that are not reachable from the current parsing scope using SourceLocation.
  - **Deliverable:** The lookup filter functionality should be successfully incorporated within the LookupResult component.

  Week 3:

  - **Task:** Implement lazy parsing for class and struct definitions.
    - Review and understand the current implementation of `ParseClassSpecifier` and the existing definition skipping logic.
    - Implement the logic to check for skippable definitions and redefinitions after getting a declaration from ActOnTag.
    - Integrate the definition skipping logic into `ParseClassSpecifier` to handle skipping class and struct definitions.
    - Test the modifications extensively to ensure they work as expected and do not introduce any regressions.
    -
  - **Deliverable:** We can successfully skip class and struct definitions, with some demonstration using tests.

Week 4-5:

- Continue implementing lazy parsing for class and struct definitions.
- Develop logic to identify expressions that represent lazy parsing entities. This will prioritize recognition of class and struct identifiers used as types within expressions.
- Establish a mechanism to initiate the parsing process for a lazy entity when it's encountered. This includes verifying if the definition has not been parsed yet and triggering efficient parsing upon confirmation.

- **Deliverable:** Lazy class and struct parsing functionality.

Week 6:

- **Task:** Implement lazy definition handling. This involves modifying `ParseFunctionDefinition` to efficiently skip function definitions. it should also check redefinition.

- **Deliverable:** We can skip function definition successfully and with some demonstration using test's.

Week 7-8:

- **Task:** Continue implementing lazy definition handling.
    - Modify `ParsePostfixExpressionSuffix` for call expressions and function pointer expressions.
    - Allocate buffer time for debugging any problems.
- **Deliverable:** Lazy function parsing functionality.

Week 9:

- Use this week as a buffer for any issues or other delayed tasks.

Week 10-11:

- **Task:** Test and benchmark the implemented lazy entity parsing. Fix any issues and get the performance impact.
- **Deliverable:** Benchmark results comparing performance with traditional parsing on various code samples.

Week 12:

- **Task:** Document the work and its benefits.
- **Deliverable:** A blog post explaining the project, its implementation details, and the performance implications of lazy entity parsing.

3. GSoC Contribution:

- **Motivation:** I am highly passionate about compiler and system programming, driven by a deep curiosity to understand how computers translate high-level code into efficient machine instructions. Specifically, I'm fascinated by the art of optimization – squeezing the most performance out of code by minimizing memory usage and CPU cycles. The LLVM project stands as a cornerstone in this field, providing a powerful infrastructure for building compilers and optimizing code for a wide range of architectures. Contributing to the LLVM project would be an incredible opportunity. Working on the project would allow me to learn from their expertise and gain invaluable insights into this complex domain.

- **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:

  - [InstCombine] FP fold, cond ? x : -x == 0 into x == 0 #85250
  - [X86][AVX] Fix handling of out-of-bounds shift amounts in AVX2 vector logical shift nodes #83840
  - [InstCombine] Fold mul (sext bool X), Y into select X, -Y, 0
  - [X86][AVX] Fix handling of out-of-bounds shift amounts in AVX2 vector shift nodes
  - [InstCombine] Fix Failure to convert vector fp comparisons that can be represented as integers #82241
  - [DAG] Fix Failure to reassociate SMAX/SMIN/UMAX/UMIN

- [mlir][spirv] Replace hardcoded attribute strings with op methods Resolve #77627
- [InstCombine] Fix Failure to fold (and %x, (sext i1 %m)) -> (select %m, %x, 0) with multiple uses of %m
- [InstCombine] Fold fcmp ogt (x - y), 0 into fcmp ogt x, y #85245
- [InstCombine] Fix fail to fold (A >> C1) Pred C2 if shr is used multple times #83430
- Update amdgpu_gfx functions to use s0-s3 for inreg SGPR arguments on targets using scratch instructions for stack #78226

4. Commitment

Communication:

- Availability: I am available for regular communication with my mentors during Indian Standard Time (UTC+5.5) evenings. I am also happy to schedule additional meetings as needed.

Collaboration:

- **Weekly Check-ins:** I plan to maintain regular contact with my mentors through weekly check-in meetings. These meetings will ensure we're aligned on progress, identify any roadblocks early on, and keep communication flowing smoothly.
- **Proactive Problem Solving:** In the event of unexpected delays, I will immediately notify my mentors. We can then collaboratively adjust the work schedule to minimize disruption and ensure project success.

Dedication:

I am fully committed to this project and eager to dedicate all my time to it throughout the summer. Without any conflicting work obligations, I will have the bandwidth to focus intensely on the project's goals and deliver high-quality results.