CppAlliance Fellowship Proposal

Optimize Usage of Source Locations in Clang Modules

Applicant: Ayokunle Amodu
Mentor: Vassil Vassilev, Princeton University / CERN
Organization: CppAlliance, Compiler Research Group.
Duration: May 4 – August 31, 2026
Contact: ayokunle321@gmail.com | github.com/ayokunle321

Contents

---

1. Abstract

Clang represents every source location as a 32-bit offset managed by `SourceManager`. When a module is loaded, it allocates a new region of this space without checking whether identical files have already been seen. As a result, shared headers included across multiple modules are assigned redundant regions, causing the loaded portion of the address space to grow with the number of modules rather than the number of unique files. This proposal introduces deduplication at the allocation site so identical files reuse existing source location ranges instead of consuming new ones. The approach is first validated with a `DenseMap`-based prototype, then extended to an `llvm::IntervalMap` design for scalable range-based reuse. The result is a system that scales with unique files rather than module count, reducing unnecessary address space consumption in large modular builds.

2. Motivation

Loading the standard library in a typical libc++ compilation consumes roughly 96MB of source location address space before a single line of user code is processed. In a build that imports std.pcm, std_core.pcm, and Darwin.pcm, the last allocated entry sits at offset 2,147,453,751, just 29,897 bytes below the 2^{31} limit. The program being compiled does little more than call `std::cout` once. That space does not recover between translation units, and each additional module import reduces what remains. This leaves very little margin.

The community looked at this in 2025. Two RFC threads on LLVM Discourse, Revisiting 64-bit Source Locations and RFC: Opt-in CMake Option for 64-bit SourceLocation, explored expanding `SourceLocation` to 64 bits. Neither converged on adoption. The core objection was that widening the offset raises the memory footprint of common AST nodes like `DeclRefExpr` across every build, not just the ones actually hitting the limit. It adds cost everywhere to solve a problem that originates in one place.

That place is the allocation site. The space runs out not because 31 bits is too small, but because the same files are allocated repeatedly. When modules are not built bottom-up, the same headers appear independently across multiple modules, and each import repeats the allocation regardless of whether the file has already been mapped. The fix belongs here.

---

3. Benefits to the Community

To the Clang and LLVM Ecosystem

- Directly addresses source location exhaustion in modular builds without expanding `SourceLocation` to 64 bits, preserving the memory and cache efficiency of the 32-bit design for all users.
- `SourceManager`, `ASTReader`, and the module serialization pipeline all receive upstream-ready patches, with LIT test coverage covering key scenarios including shared headers across modules, symlinked files resolving to the same inode, and chained PCH imports.

- Implements the recycling strategy the LLVM community identified before considering wider `SourceLocation` expansion, closing an open design question with a concrete implementation.

To Users of C++ Modules

- Large projects that import many modules sharing common headers such as the standard library will see the loaded `SLoc` address space scale with unique content rather than module count.
- Projects approaching the 2^{31} limit receive a structural fix rather than a limit increase that could expose declaration merging issues.

---

4. Background: How Source Locations Work

`SourceLocation` is a 32-bit value (`SourceLocation::UIntTy`) used by Clang to represent positions in source code. The highest bit is reserved to distinguish macro locations, leaving 31 bits for offsets into a global address space managed by `SourceManager`.

This address space is divided into two regions. Local `SLocEntries` represent the current translation unit and grow upward from zero. Loaded `SLocEntries` represent entries deserialized from precompiled modules and grow downward from 2^{31}. This layout allows both regions to coexist within the same address space without overlapping.

When a module is loaded, `SourceManager` allocates space for its source location entries using `AllocateLoadedSLocEntries`. The function takes the number of entries and the total byte size, reserves a contiguous region in the loaded portion of the address space, and returns a (`BaseID`, `BaseOffset`) pair. This pair is then used to translate the module's local source location offsets into global ones:

```
LoadedSLocEntryTable.resize(LoadedSLocEntryTable.size() + NumSLocEntries);
NextLoadedOffset -= TotalSize;
return {-(int)LoadedSLocEntryTable.size(), NextLoadedOffset};
```

The allocation is unconditional and each module receives a distinct region in the loaded address space, regardless of whether the same files have already been loaded. This behavior originates from the call site in `ASTReader`, where `AllocateLoadedSLocEntries` is invoked during module deserialization for every module:

```
std::tie(F.SLocEntryBaseID, F.SLocEntryBaseOffset) =
SourceMgr.AllocateLoadedSLocEntries(F.LocalNumSLocEntries, SLocSpaceSize);
```

As a result, if multiple modules include the same header, each module is assigned a separate, non-overlapping region for identical files. This is where duplication occurs, and any solution must intervene at this allocation point.

5. Reproducing the Problem

The experiment setup is based on a minimal configuration described in the initial design discussions with Vassil Vassilev. I reproduced it locally to validate the observed behavior and establish a baseline. The setup consists of four files that isolate duplication:

File Setup

`a.h` -- Module A header, includes `iostream`:

```
#ifndef A_H
#define A_H
#include <iostream>

int mod_A = 42;
#endif // A_H
```

`b.h` -- Module B header, also includes `iostream`. This is where the duplication originates:

```
#ifndef B_H
#define B_H
#include <iostream>

int mod_B = 21;
#endif // B_H
```

`use.cpp` -- the translation unit under test:

```
#include "a.h"
#ifdef TWO
#include "b.h"
#endif

extern "C" int printf(const char*, ...);
int main() {
  printf("mod_A=%d\n", mod_A);
#ifdef TWO
     printf("mod_B=%d\n", mod_B);
#endif
}
```

`module.modulemap`:

```
module A { header "a.h" export * }
module B { header "b.h" export * }
```

Experiment Commands

One-module baseline (module A only):

```
gtime -v clang++ -fmodules -fimplicit-module-maps \
     -fmodules-cache-path=. use.cpp \
     -Xclang -fdisable-module-hash \
     -Xclang -print-stats >stats-one-modules.txt 2>&1
```

Two-module case (A and B, `iostream` duplicated across both):

```
gtime -v clang++ -DTWO -fmodules -fimplicit-module-maps \
     -fmodules-cache-path=. use.cpp \
     -Xclang -fdisable-module-hash \
     -Xclang -print-stats >stats-two-modules.txt 2>&1
```

The `-fdisable-module-hash` flag forces both modules to be rebuilt without a content-based cache key, so duplication is always visible. Without it, the module cache reuses a prior build and masks the problem entirely.

Results: The Doubling

One-module case:

```
AST File Statistics:

278/5827 source location entries read (4.77%)
20/30960 types read (0.064599%)
125/29037 declarations read (0.430485%)
1485/5199 identifiers read (28.563187%)
0/1465 macros read (0.000000%)
2/47438 statements read (0.004216%)
1/3293 lexical declcontexts read (0.030367%)
11/1666 visible declcontexts read (0.660264%)
```

Two-module case (module B added, `iostream` now duplicated):

```
AST File Statistics:

574/11650 source location entries read (4.92%)
37/61916 types read (0.059758%)
245/58068 declarations read (0.421919%)
2965/10386 identifiers read (28.548044%)
0/2929 macros read (0.000000%)
3/94872 statements read (0.003162%)
1/6584 lexical declcontexts read (0.015188%)
19/3331 visible declcontexts read (0.570399%)
```

Source location entries increase from 278/5827 to 574/11650. The denominator doubles exactly, from 5827 to 11650. Types, declarations, and identifiers all scale proportionally, since every `iostream` entry is allocated again when module B is loaded. Add a third module that includes `iostream` and it triples. The pattern is linear and entirely avoidable.

---

6. Root Cause and Proposed Fix

`AllocateLoadedSLocEntries` does not retain any information about prior allocations. Each module is assigned a new region of the address space, even when the same files have already been loaded.

For example, if two modules both include `iostream`:

```
Module A loads iostream  ->  allocates SLoc range [X .. X+N]
Module B loads iostream  ->  allocates SLoc range [X+N+1 .. X+2N] (duplicate,
wasted)
```

Both ranges represent identical files, but are allocated independently. The proposed fix is to introduce reuse at the allocation point. Instead of allocating a new region, previously allocated ranges are reused when the same file is encountered:

```
iostream allocated once at [X .. X+N]
Module A and Module B both rewired to reuse that range
```

The challenge is that the SLoc address space serves two purposes at once: lookup and identity. It supports efficient lookup by mapping a raw offset back to a file, line, and column, and it encodes identity by ensuring that locations within the same file occupy a contiguous range.

Reusing an existing range therefore requires remapping a module's local offsets into the shared global range on a per-file basis. The `F.SLocEntryBaseOffset` field in `ModuleFile` provides the base for this remapping.

This approach is safe because SLoc entries are lazily deserialized and only accessed when needed. If two modules load identical file content, their underlying `SrcMgr::SLocEntry` representations are consistent. As a result, diagnostics from either module that reference the shared range resolve correctly.

---

7. Implementation Plan

The work splits into two phases around the midterm. Phase 1 proves the concept with a `DenseMap` prototype. Phase 2 replaces that prototype with an `IntervalMap` and brings the implementation to an upstream-ready state. Testing runs throughout both phases rather than being treated as a separate step.

7.1 Phase 1: DenseMap Prototype

The fix is conceptually simple: before allocating a new SLoc region for a module's input files, check if those files have already been allocated. If they have, reuse the existing range; if not, allocate a new one and record it.

The prototype implements this with a `DenseMap<FileEntryRef, std::pair<int, SourceLocation::UIntTy>>` added to `SourceManager`. The changes are contained to three places:

- `SourceManager.h` introduces the map
- `AllocateLoadedSLocEntries` in `SourceManager.cpp` performs the lookup, returning an existing `(BaseID, BaseOffset)` when available or allocating and recording otherwise
- `ASTReader.cpp` is updated in the `SOURCE_LOCATION_OFFSETS` case to pass file identity through so allocation decisions are made with full context

The success condition: in the two-module experiment, the denominator in `-print-stats` should drop from 11650 back toward the one-module baseline of 5827. If it does not, the issue lies in how file identity is propagated, and that is resolved before anything else moves.

Once the prototype behaves correctly and the numbers confirm it, a focused test suite is added. Three scenarios must pass: two modules sharing the same header directly, the same file appearing under different paths with the same inode, and a module being imported through multiple transitive paths. A `-print-stats` counter for redirected allocations is added so the deduplication behavior is directly observable. The full LLVM test suite is then run with the prototype in place, and any regressions are resolved before the midterm.

Midterm deliverable: a working prototype with reduced duplication, all targeted tests passing, and a clean full test suite.

7.2 Phase 2: IntervalMap Integration

The DenseMap prototype establishes correctness while the production implementation uses `llvm::IntervalMap<SourceLocation::UIntTy, FileEntryRef>`, keyed by global `SLoc` offset ranges. This aligns with the structure of the address space and handles more complex reuse scenarios.

The `IntervalMap::Allocator` is introduced in `SourceManager` first. The `DenseMap` is then replaced with the `IntervalMap` for the non-overlapping case, with all existing tests kept passing. This is followed by profiling on a real libc++ build to verify lookup cost stays within one to two cache lines. Only then is the partial-overlap case addressed.

The `IntervalMap` generalizes beyond exact matches. While the `DenseMap` handles identical file identity, the `IntervalMap` detects overlap in allocated ranges for the same file. This supports the partial-overlap case, where different modules serialize different numbers of SLocEntries for the same file, which does not appear in the simple experiment but does occur in real builds.

This case is the main challenge. `ASTReader` currently assumes a single contiguous base offset per module file, and supporting partial reuse requires revisiting that assumption. The preferred approach is to introduce local remapping within `ASTReader`, keeping changes contained and avoiding disruption to

`SourceManager` invariants. If that proves insufficient, broader changes are considered, with scope confirmed before implementation.

After handling partial overlap, edge case tests are added: chained PCH with shared includes and modules loaded through multiple transitive paths. The full LLVM test suite is then run again to ensure stability.

The final step is upstream preparation. The work is split into three patches: the deduplication mechanism, the test suite, and the `IntervalMap` integration. Each patch is independently testable and reviewable. They are shared on LLVM Discourse, with time allocated for feedback and iteration.

---

8. Detailed Timeline

The bonding period focuses on two things: building a deep understanding of the codebase and engaging with the existing design discussions. This includes reading through `SourceManager` and `ASTReader`, as well as commenting on the relevant RFC threads and reaching out to contributors involved in those discussions. That context matters later when patches go up for review, and it takes time to do properly.

I am committing 40 hours per week for the full three months, and I am prepared to extend up to six months if needed to see the work through.

Phase	Dates	Deliverables
Bonding week 1	May 4-10	Read `SourceManager.h/.cpp`, `SourceLocation.h`, `ASTReader.cpp`. Document every call site of `AllocateLoadedSLocEntries`. Reproduce experiment and record baseline observations.
Bonding week 2	May 11-17	Read `ASTWriter.cpp` and `IntervalMap.h`. Comment on both RFC threads. Reach out to contributors active in those discussions. Align with Vassil on implementation strategy.
Bonding week 3	May 18-24	Buffer and final prep. Development environment confirmed. Any open questions from reading resolved before coding starts.
Week 1	May 25-31	Add `LoadedFileEntryAllocations DenseMap` to `SourceManager.h`. Add dedup check to `AllocateLoadedSLocEntries`. Run existing module LIT tests after each change.
Week 2	Jun 1-7	Wire `FileEntry` identity through `SOURCE_LOCATION_OFFSETS` in `ASTReader.cpp`. Confirm two-module denominator shrinking toward baseline. Debug identity pass-through if not.

Phase	Dates	Deliverables
Week 3	Jun 8-14	Write LIT test for two-module shared header scenario. Write symlink scenario test. Both passing before moving on.
Week 4	Jun 15-21	Write transitive import LIT test. Add `-print-stats` counter for redirected allocations. All three scenarios passing.
Week 5	Jun 22-28	Run full LLVM test suite with prototype in place. Fix any failures. Record experiment observations with prototype.
Week 6	Jun 29-Jul 5	Buffer week before midterm. Clean up remaining issues. Prepare midterm report.
Midterm	Jul 6-10	Midterm evaluation. Denominator reduction confirmed, three test scenarios passing, full test suite clean, experiment observations recorded.
Week 7	Jul 13-19	Add `IntervalMap::Allocator` to `SourceManager`. Replace `DenseMap` with `IntervalMap` for non-overlapping case. All existing tests passing.
Week 8	Jul 20-26	Profile non-overlapping `IntervalMap` on libc++ compilation. Verify lookup cost within expected cache line range.
Week 9	Jul 27-Aug 2	Confirm partial-overlap approach with Vassil. Begin investigating local `ASTReader` remapping path.
Week 10	Aug 3-9	Complete partial-overlap implementation and write tests confirming correct reuse when modules serialize different numbers of `SLocEntries` for the same file.
Week 11	Aug 10-16	Edge case tests: chained PCH with shared includes, module reload via multiple transitive paths. Full LLVM test suite run.
Week 12	Aug 17-23	Prepare upstream patch series (3 patches). Clean up, document, and submit to LLVM Discourse.
Final	Aug 24-31	Final evaluation. Respond to first round of review. Final report.

9. Expected Final Results

By the end of the fellowship, the following will be in place:

- A working `DenseMap` prototype in `SourceManager` that eliminates redundant `SLoc` allocation for identical input files, with the two-module experiment denominator returning to the one-module baseline
- A full `IntervalMap<SourceLocation::UIntTy, FileEntryRef>` implementation handling both non-overlapping and partial-overlap cases, with `IntervalMap::Allocator` properly integrated into SourceManager
- A `-print-stats` counter tracking redirected allocations so the deduplication behavior is visible and measurable in any compilation
- LIT test coverage for: two-module shared header, symlink scenario, transitive import, chained PCH with shared includes, and partial-overlap case
- A measurements dataset comparing address space consumed, peak RSS, and wall time across the canonical two-module experiment and a real-world libc++ compilation, before and after deduplication
- An upstream patch series (3 patches) submitted to LLVM Discourse with accompanying correctness documentation

9.1 Stretch Goals

- Full upstream landing of all three patches within a 3-month period
- Extension of the experiment harness to cover explicit module builds (`-fmodule-file=A=./A.pcm`), which Vassil noted in the initial design document as having even higher resource consumption than implicit builds
- Investigation of whether the same dedup mechanism applies to macro ID allocations, which show the same linear doubling in the `-print-stats` output

---

10. Performance Evaluation

The primary metric is source location address space consumed, read from `-print-stats` output. The target is that the two-module denominator of 11650 returns to the one-module value of 5827 after deduplication. For N modules sharing the same headers, the goal is a denominator that stays flat at the one-module value rather than growing linearly.

Secondary metrics collected on both the two-module experiment and the real libc++ compilation:

- Peak RSS to measure whether deduplication reduces memory consumption proportionally to the address space savings
- Wall clock time per compilation, to confirm the dedup mechanism adds no meaningful overhead to the build
- `IntervalMap` lookup cost profiled, targeting one to two cache lines per traversal given the key and value sizes
- Number of redirected allocations from the `-print-stats counter`, as a direct measure of how many duplicate allocations were eliminated

The dedup mechanism itself should add no meaningful overhead. `DenseMap` lookups are O(1) and `IntervalMap` lookups are O(log N) where N is the number of distinct input files across loaded

modules, typically a few thousand at most. Any wall time regression would indicate a bug rather than an inherent cost.

---

11. Risks and Mitigations

Diagnostic Regressions

Sharing `SLoc` ranges between modules could cause diagnostics from one module to resolve source text from another module's version of the same file. Vassil flagged this in the initial design document as the hardest risk to predict without a prototype, which is exactly why he recommended starting with one before making any broader changes.

The mitigation: deduplication only triggers when files are verifiably identical via `FileEntryRef` inode, and any mismatch falls back to fresh allocation. The diagnostic regression suite runs after every meaningful change throughout both phases.

Partial-Overlap Edge Cases

The partial-overlap case, where one module serialized more `SLocEntries` for a file than another, requires careful `BaseOffset` arithmetic. `ASTReader` currently assumes a single contiguous base per module file and supporting partial reuse means addressing that assumption.

There are two paths forward: a local remapping solution inside `ASTReader` that avoids touching broader `SourceManager` invariants, or a more invasive `SourceManager` change that is cleaner but brings performance unknowns. The local path gets investigated first. Either way the scope is confirmed with Vassil at the start of week 9 before any implementation begins, so there is no risk of going in the wrong direction for two weeks.

---

12. About Me

I am a final-year Computing Science student at the University of Alberta, finishing my degree this spring. My background is in compiler design and computer architecture, and I have spent the past year contributing actively to LLVM infrastructure.

I worked as a research assistant under Prof. J. Nelson Amaral from May to August 2024. Most of that time went into building a RISC-V dataflow analysis framework: control-flow graph construction, dead code elimination, live variable analysis on non-SSA assembly. That work is now part of the CMPUT 229 curriculum.

Here are the links to the labs: CFG & DCE

I also TA'd that course, teaching 200 students per semester. Working that deep in RISC-V assembly means you are constantly thinking about 32-bit representations, what fits in an offset, what the hardware actually sees. It is not abstract to me.

From there I moved into MLIR, which is where I really got hooked. I built a complete compiler for the Gazprea language as part of a four-person team, implementing code generation across the scf, memref, linalg, and llvm dialects.

LLVM Contributions

My upstream work spans ClangIR, MLIR, and Clang itself:

- In Clang I have been in the diagnostics infrastructure, refactoring sites to use `enum_select` for better maintainability.
- In ClangIR I have contributed codegen and CIR-to-LLVM lowering for atomic builtins including the full `__sync_fetch_and_*` family, `__builtin_isinf_sign`, `setjmp` and `longjmp`, `__builtin_bcopy`, and FP environment RAII options.
- In the ClangIR incubator I added elementwise math builtins with ThroughMLIR lowerings.
- In MLIR I fixed two crashes in the affine dialect.

Full contribution list: https://github.com/ayokunle321/open-source-portfolio/blob/main/README.md

Why I Am the Right Person for This

Clang is not new territory for me. The diagnostics work and the ClangIR contributions have put me in the codebase across multiple layers, and I have spent enough time there to know how to navigate it. But what I think matters most for this specific project is the lower-level background. RISC-V assembly is a 32-bit world where you are constantly reasoning about what fits where and what the hardware actually sees. MLIR work is similar in a different way: you are tracing how something expressed at a high level has to survive a chain of lowerings and still mean the same thing at the bottom. Both of those have made me comfortable sitting with low-level invariants and being careful about what I am allowed to assume.