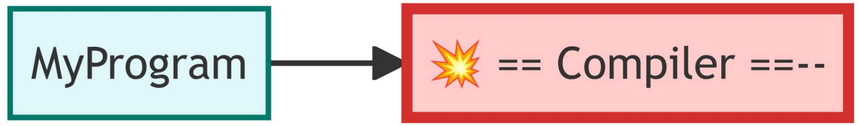
Towards Automatic Reduction of Module Bugs

Improving C-Vise

Maksim Ivanov, Google

The Problem: Test-Case Reduction

- **Goal:** Reduce a large, failing program to a minimal, self-contained reproducer.
- **Why:** Critical for debugging compiler issues (speeds up debugging), reporting downstream issues, creating focused regression tests.



- **Challenge:** Existing tools (C-Reduce, C-Vise) are often slow. Reductions can take hours, days, or even weeks on large, real-world test cases.
- **Core Limitation:** They are fundamentally designed for single-file inputs. Multiple files are only reduced separately and the same number of files remains.

The Specific Challenge: C++ Header Modules

Test-case reduction for C++ modules is significantly harder. A reproducer is not one file; it's a complex bundle:

- **Multiple Files:** Can involve thousands of source files, headers, and .cppmap files.
- **Multiple Commands:** Requires multiple, ordered compilation commands (to build PCMs) that must **succeed** before the final, **failing** command is run.
- **Complex Dependencies:** A bug may only manifest when a specific PCM is built and imported in a specific way.

Our Approach: Evolving C-Vise

We chose to improve C-Vise (a Python-based C-Reduce successor) to tackle this.

- **Goal 1: Speed**

  Radically improve core reduction performance, even on single-file inputs.

- **Goal 2: Versatility**

  Extend the tool to natively support multi-file, multi-command reductions for C++ modules.

- **Key Insight:** Achieving Goal 1 with a new architecture directly unblocked our path to achieving Goal 2.

Core Improvement: Hint-Based Architecture

Old C-Vise: In-Place Modification

Heuristics (e.g., "remove function body") directly modified files. This was rigid, sequential, and hard to parallelize effectively.

```
[Heuristic] ---> [Modifies File In-Place]
```

New C-Vise: Hint Emission

Heuristics emit "hints" (JSONs describing patches). A generic, parallel scheduler collects hints and decides how to test them.

```
[Heuristic] ---> [JSON "Hint"] --->
[Scheduler]
```

This decouples "what to try" from "how to try it" (+binary search, etc.).

Performance Win 1: Interleaved Execution

Old: Sequential Passes

Runs one full heuristic to completion before starting the next. It can get "stuck" on a low-yield pass for hours.

```
Run 'lines' pass (2 hours)
...stall...
Run 'remove-function' pass (1 hour)
...
```

New: Interleaved Hints

Mixes hints from **all** heuristics in a round-robin fashion. The scheduler constantly makes progress using the best reduction available from any pass.

```
Try 'line' hint
Try 'function' hint
Try 'comment' hint
Try 'line' hint (SUCCESS)
...
```

Performance Win 2: Folding Reductions

Old: Wasted Parallelism

If 5 parallel jobs find 5 different successful reductions, the tool picks **one**, discards the other 4, and restarts all workers.

```
Job 1: Success (Remove line 10)
Job 2: Success (Remove line 20)
Job 3: Fail
Job 4: Success (Remove line 30)
=> Keep Job 1, Discard 2 & 4.
```

New: Folded Reductions

We "fold" all successful, non-conflicting hints from a batch of parallel jobs into a **single** combined patch and test that.

```
Job 1: Success (Remove line 10)
Job 2: Success (Remove line 20)
Job 3: Fail
Job 4: Success (Remove line 30)
=> Fold [1, 2, 4] -> Test 1 patch.
```

This achieves a massive reduction in one step.

Miscellaneous improvements

- Robustness:
  - hung child process termination;
  - temporary files leaks;

- Performance:
  - improved ad-hoc parsers;
  - new heuristics based on Tree-sitter parsers.

Performance Results (Single-File)

These architectural changes resulted in a ~10x-80x speedup on our benchmark suite.

Test Case	C-Reduce 2.11.0	C-Vise 2.11.0 (Old)	C-Vise (New)	Speedup
clang-363816643	15 hours	16 hours	**12 min**	**75x**
clang-383027690	18 min	2.5 hours	**2 min**	**9x**
clang-321217557	?	30 days (est.)	**8.5 hours**	**85x**
clang-329180703	∞ (hung)	44 hours	**45 min**	**60x**
clang-410818184	31 hours	85 hours	**1.5 hours**	**20x**
gcc-94937	24 hours	∞ (hung)	**40 min**	**35x**
gcc-92516	4.1 hours	3.2 hours	**15 min**	**13x**

Applying to C++ Header Modules

Streamlined multi-file handling

Before

- Files were processed sequentially by each pass.
- Binary search was limited to instances in a single file.
- Number of files remained constant.

After

- Passes now operate simultaneously on all files.
- Binary search operates across file boundaries.
- Detecting file cross-references, deleting unused files/dirs.

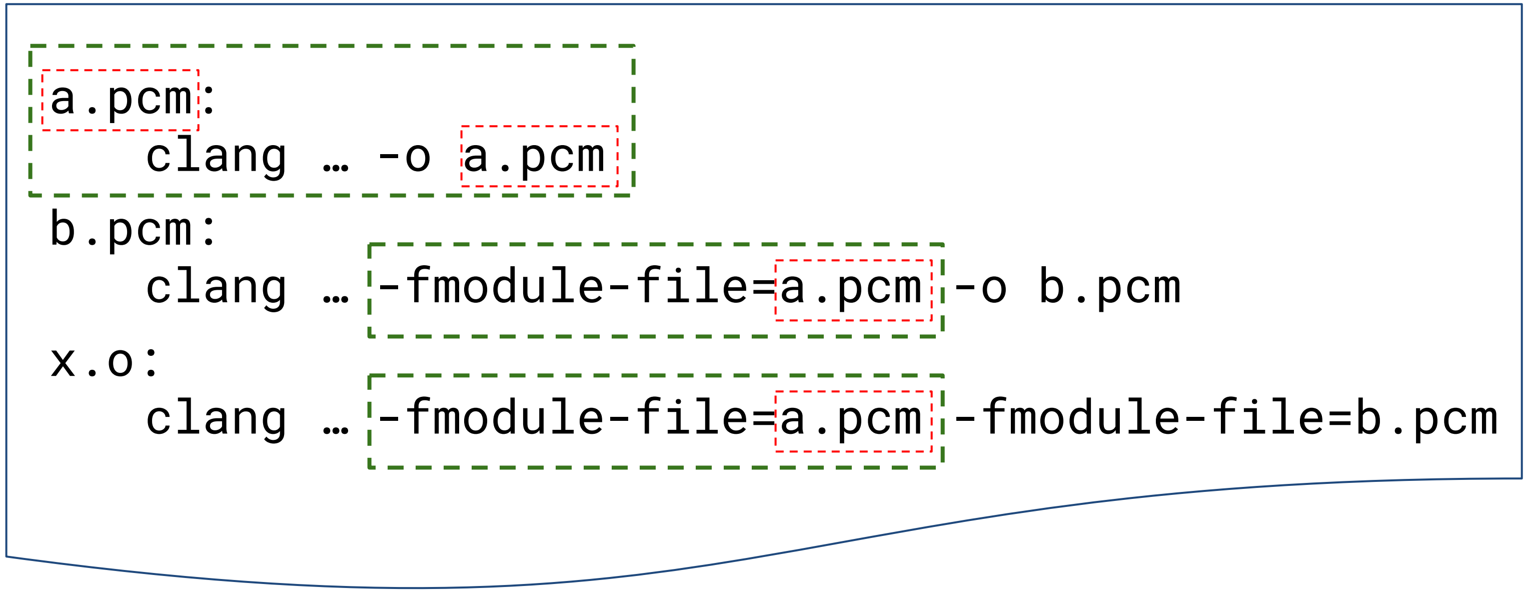
Compilation command reductions

Our approach - use Makefiles:

- ➢ stores compilation commands and dependencies between them;
- ➢ can also be executed by the interestingness test.

New heuristics: cmd parameter removal, target removal.

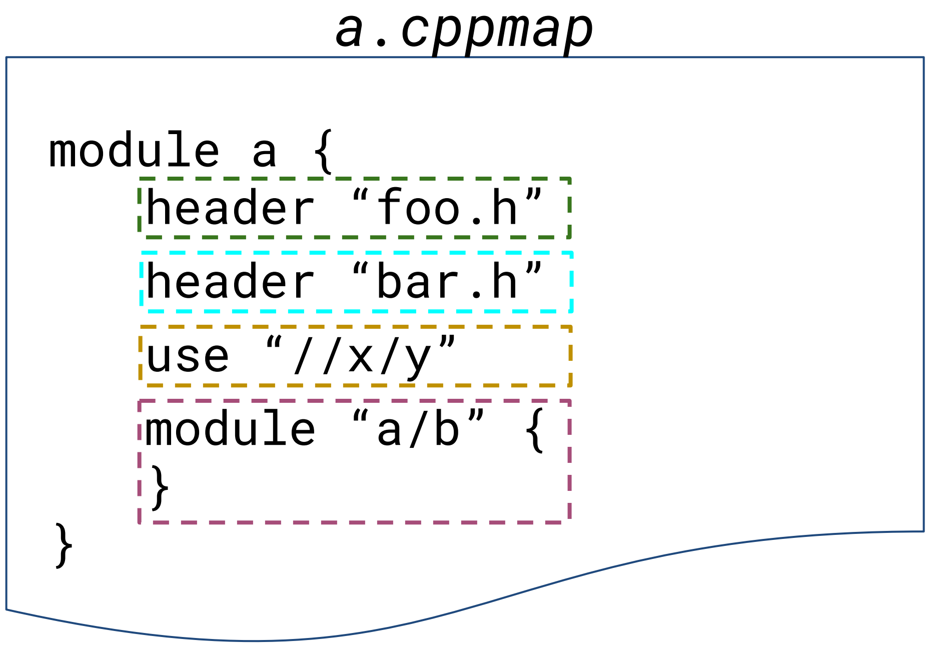
Makefile



```
a.pcm:
    clang … -o a.pcm
b.pcm:
    clang … -fmodule-file=a.pcm -o b.pcm
x.o:
    clang … -fmodule-file=a.pcm -fmodule-file=b.pcm
```

Module map reductions

New heuristics for structured removal of contents from module map files.



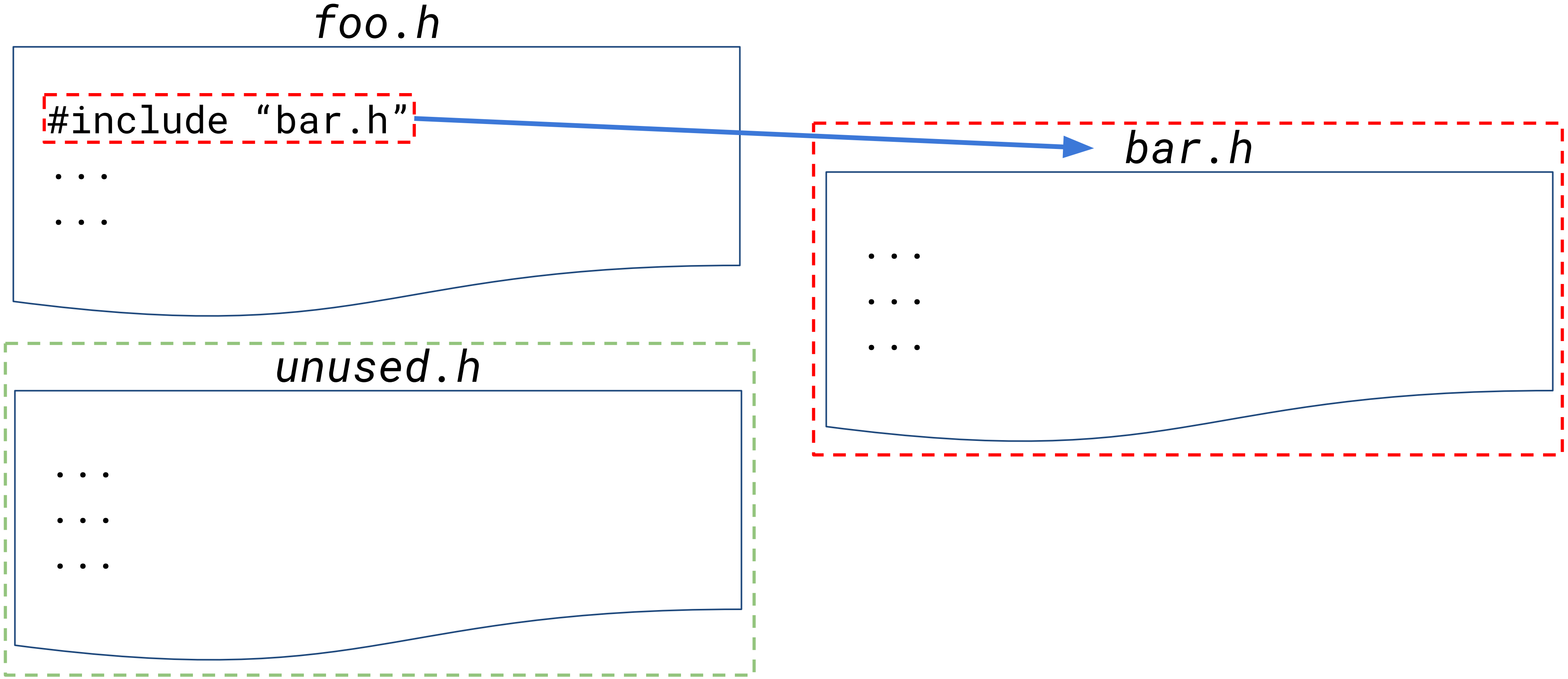
a.cppmap

```
module a {
    header “foo.h”
    header “bar.h”
    use “//x/y”
    module “a/b” {
    }
}
```

Detecting file references

Goal: new passes to delete unused files; to attempt deleting a file with all references.

How: Run the Clang preprocessor to build the graph of #include’s.



Performance results (Header Modules)

Test Case	input size	duration	output size
clang-355835505	37 MB, 2638 files	2.5 hours	47 KB, 13 files

Summary

Done

- Re-architected C-Vise with "hints" for flexibility and parallelism, achieving **10x-80x speedup** on single-file tests.
- Efficient multi-file reduction.
- Header module aware passes.

Future Work

- C++20 modules support.
- More heuristics.
- Improving parallelism bottlenecks.
- Reduction in the cloud.
- LLM-based heuristics and drivers.