Can a Local LLM Beat -O3?

A Small Experiment in C++ and LLVM IR Optimization

CppCon 2026
Tijl Vercaemer

Introduction

During his keynote at ACCU on Sea in June, Andrei Alexandrescu made some predictions. One of those was the integration of LLMs and compilers. Compilers can generate optimization remarks for LLMs to process, while LLMs can propose optimizations that are difficult for conventional compilers to discover. This poster investigates how useful small, local LLMs are today as performance optimizers.

Questions

This experiment sets out to answer three questions:

- How much can a small, local LLM improve performance as part of the compilation process?
- How likely is the LLM to produce code that is functionally correct?
- Is the IR optimization stage the right place to introduce LLM-based optimizations, or is it better to optimize the C++ code before it goes to the compiler?

Test code

The testing library consists of five exported functions, each implemented in a separate C++ file. The test cases contain deliberately inefficient algorithms, data structures, and implementation choices. This gives both the compiler and the LLM opportunities for optimization.

These are the five test functions:

- **fibonacci**: Recursive Fibonacci calculation
- **format_list**: Vector/string formatting
- **repeated_sort**: Repeated range sorting
- **count_matches**: Membership/counting lookup
- **top_words_from_file**: File parsing & word frequencies

Benchmarking

Each optimized version of the test code is compiled into a shared library. The benchmark runner is compiled only once and reused for every tested library. It loads the optimized library, runs tests to validate correctness, calibrates the number of iterations to run for approximately 30 seconds, and measures throughput. Final comparisons use repeated, randomized runs and compare median throughput. A difference of <=2% is treated as benchmark noise.

Optimization strategies

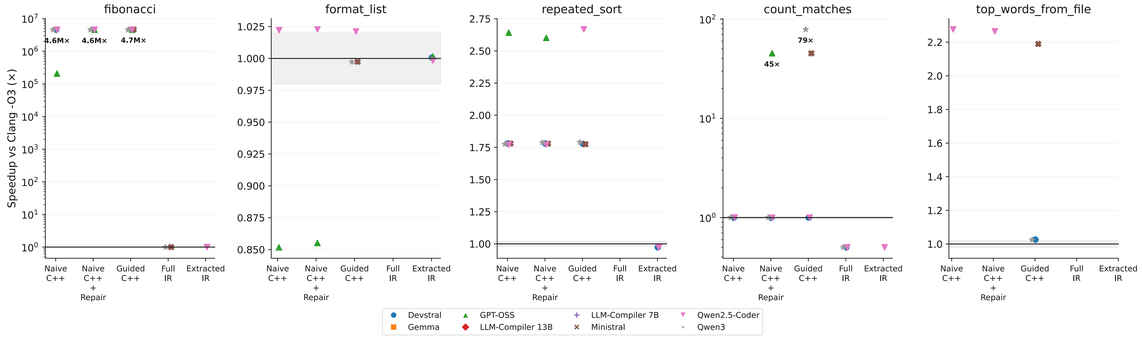
- **Naive C++**: Optimize C++ once, no retries
- **Naive C++ + repair**: Retry invalid C++ candidates
- **Guided C++**: Optimize C++ using compiler remarks
- **Full LLVM IR**: Optimize complete LLVM IR
- **Extracted LLVM IR**: Optimize individual LLVM IR functions

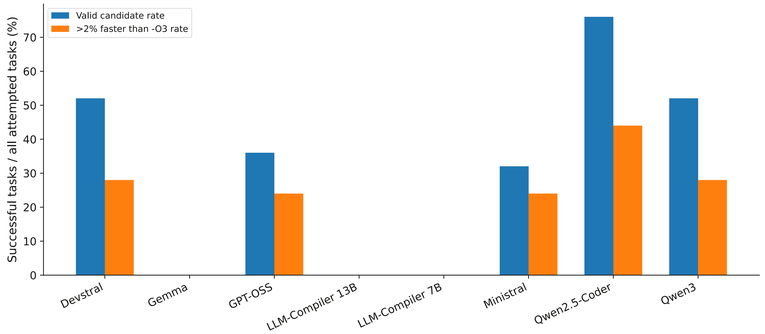
Models

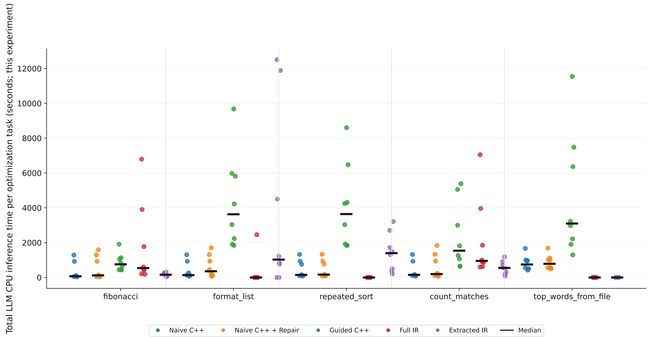
- Four general/coding models of similar size (12–14B): **Qwen3 14B**, **Qwen2.5-Coder 14B**, **Gemma 4 12B**, **Ministral 3 14B**.
- Two compiler-specialized models: **LLM Compiler 7B** and **13B**, trained on LLVM IR, assembly, and compiler optimization.
- Two larger coding/reasoning models: **gpt-oss-20b** and **Devstral Small 2 24B**.

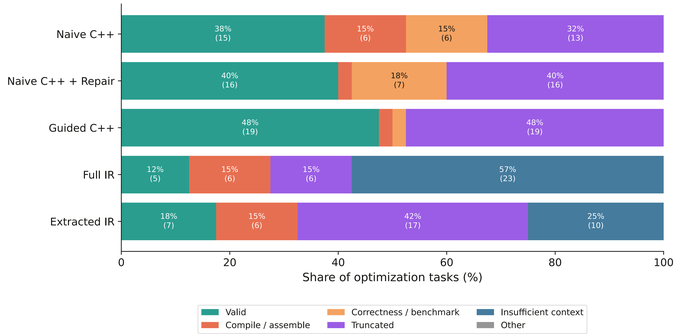
Hardware

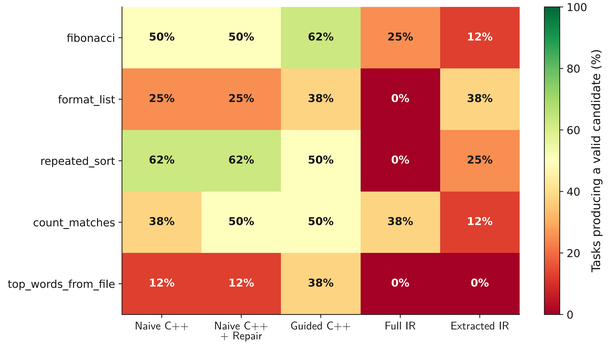
The full experiment ran on AlmaLinux on an AMD Ryzen 9 5950X with 128 GB of RAM. The compiler was Clang/LLVM 20.1.8. No GPU acceleration was used. LLM inference ran entirely on the CPU using llama.cpp.











Conclusions

C++ was a much more effective optimization target than LLVM IR. Across 40 model/benchmark combinations, C++ produced 15 valid candidates with the naive approach, 16 with corrections, and 19 with compiler guidance. Full LLVM IR produced only 5 valid candidates and extracted IR only 7. Neither IR approach produced a result more than 2% faster than -O3.

LLM-generated optimizations require strong validation. Candidates frequently failed to compile, were truncated, or failed correctness or benchmark validation. Some candidates contained undefined behavior. Unlike conventional compiler optimizations, LLM-generated code cannot be assumed to preserve functional behavior.

LLMs can substantially outperform -O3, but mainly by changing the program structure. Examples include replacing recursive Fibonacci with iteration, moving an invariant sort outside a loop, and replacing repeated linear searches with a hash set. These are transformations that conventional optimizing compilers generally don't perform.

LLM optimization is fundamentally different from traditional compiler optimization. The model may change algorithmic choices that the programmer deliberately made. Such transformations do not belong in an invisible compiler pass. They are more suitable as reviewable suggestions.

Compiler-guided C++ gave the best results overall. It produced 19/40 valid candidates and 14/40 candidates more than 2% faster than -O3. This compares with 16/40 valid and 12/40 faster for Naive C++ + repair, and 15/40 and 11/40 for one-shot Naive C++.

Reliability depends strongly on the model. Qwen2.5-Coder was by far the most successful model in this experiment. Some models produced no valid candidates at all. Similar parameter counts did not imply similar optimization ability.

Benchmark	Fastest LLVM	LLM advantage
fibonacci	-Os	+401,400,000%
format_list	-Ofast	+2.06%
repeated_sort	-O2	+166.2%
count_matches	-O1	+7,757%
top_words_from_file	-O2	+114.8%

Code

Some more context, together with a link to the code and all results, can be found at:
https://tijl.blog/posts/cppcon-local/

What about frontier models?

Invalid outputs still occur, but preliminary tests show more sophisticated IR transformations. I plan to publish some results at:
https://tijl.blog/posts/cppcon-cloud/

Can a Local LLM Beat -O3? Yes. But it does so less by being a better compiler and more by acting like a programmer.