WEB BASED LLVM COMPILER STUDIO USING C/C++

Project Reference No.: *49S_BE_2476*

College : J.S.S. Academy of Technical Education, Bengaluru
Branch : Department Of Computer Science and Engineering
Guide(S) : Dr. Pradeep H K
Student(S) : Mr. Ajith Kumar Ms
Mr. Gowtham U
Mr. Harsh Anand
Mr. Jocelyn Adrian Parsons

Objectives:

1. Design and develop a fully browser-based compiler platform supporting C and C++ programs without requiring any local installation of compiler toolchains.
2. Enable multi-stage compilation output access including executable results, LLVM Intermediate Representation (IR), and target-specific assembly code (x86-64 / AArch64).
3. Facilitate direct side-by-side comparison of LLVM (Clang) and GCC compilers across multiple optimization levels (–O0, –O1, –O2, –O3, –Os) for benchmarking purposes.
4. Integrate LLVM pass-based granular analysis with quantifiable instruction-count metrics, enabling users to observe the individual impact of specific optimization passes.
5. Provide automated Control Flow Graph (CFG) generation and visualization to support structural understanding of program execution paths, loops, and branching logic.
6. Incorporate an AI-powered coaching module (Gemini 1.5 Pro) that translates low-level compiler outputs into human-readable, contextual explanations and recommendations.
7. Ensure secure, isolated, and resource-constrained code execution using Docker containerization to protect platform integrity in a multi-user environment.

Methodology:

A. System Architecture & Technology Stack

- The frontend is built as a component-driven React application featuring the Monaco Code Editor for professional-grade syntax highlighting and LSP-based real-time semantic validation.
- The backend is implemented in Python Flask, exposing two primary RESTful API endpoints: /api/compile and /api/llvm-pass.
- All user-submitted code is executed within ephemeral Docker containers that enforce strict CPU, memory, and time limits.

B. Main Compilation Workflow

- A user writes C/C++ source code in the Monaco editor, selects the compiler (LLVM or GCC), optimization level, and desired output formats.

- The backend validates the input, provisions a Docker container, invokes the appropriate compiler toolchain, and captures all outputs — standard output, LLVM IR, assembly, and error diagnostics.
- Results are rendered in a tab-based interface on the frontend.

C. AI Coach Analysis Workflow

C1. Optimization Advisor

- Triggered based on user selection, the system analyzes the source code and evaluates it across multiple optimization levels (-O1, -O2, -O3, -O4, -O5) using a Multi-Pass Benchmarking Engine.
- The backend processes these results and determines the most efficient optimization level based on execution speed and memory usage.
- The final recommendation is displayed in the Best Optimizer section.

D. LLVM Pass Analysis & CFG Generation

- The /api/llvm-pass endpoint enables surgical, pass-level optimization.
- Users select specific LLVM transformation passes (e.g., –mem2reg, –dce, –simplifycfg, –constfold) applied sequentially.
- After each pass, the system generates an updated IR snapshot and calculates a Metric Delta — a quantitative, instruction-level comparison of before-and-after states.
- CFGs are rendered using a high-performance graphing engine.

Key Experimental Results:

- Matrix Multiplication Benchmark: At –O0 baseline, the innermost loop block generated ~142 instructions per iteration. After applying –O3, the instruction count fell to just 38 — a reduction of nearly 73% — achieved through Loop Unrolling, Function Inlining, and SIMD Vectorization (vmovups / vmulpd instructions on x86-64 targets).
- LLVM vs. GCC Cross-Compiler Analysis: Cross-compiler analysis revealed systematic behavioral differences. GCC consistently prioritized code compactness (smaller binary footprint) while LLVM generated wider assembly optimized for Instruction-Level Parallelism. At identical –O2 flags, generated assembly diverged by 10–15% in instruction count across all benchmarks.
- LLVM demonstrated more aggressive SIMD vectorization in matrix multiplication benchmarks, while GCC favored compact addressing schemes for smaller binary sizes — reflecting each compiler's distinct engineering philosophy.
- Platform Validation: Successfully validated across a tiered test suite — from simple recursive algorithms (Fibonacci, Factorial) to memory-intensive data structures (linked lists, binary search trees) and high-cyclomatic-complexity control flow patterns.
- Security & Integrity: Docker-based sandboxed execution model ensures platform security and resource fairness. Plagiarism check returned a similarity score of only 2%, affirming originality of the work.

Key Project Outcomes:

- Fully functional browser-based compiler platform integrating LLVM (Clang) and GCC with no local installation.
- Demonstrated 73% reduction in instruction count via –O0 to –O3 optimization (Matrix Multiplication benchmark).
- Cross-compiler instruction count differences of 10–15% between LLVM and GCC at identical optimization levels.
- Secure Docker-containerized execution with strict CPU, memory, and timeout enforcement.
- Plagiarism score of 2% (DrillBit), affirming originality and research integrity.

Key Learnings:

- Deep practical understanding of LLVM architecture — LLVM IR, SSA form, optimization passes and Pass Manager.
- Docker orchestration, resource isolation, cold-start latency mitigation in shared cloud environments.
- Full-stack development integrating React, Monaco Editor, Flask REST APIs, and WebSocket real-time communication.
- Integration of Gemini 1.5 Pro LLM API for technical domain-specific explanation generation.

Industry Relevance:

- Academic: Functions as a virtual compiler laboratory — students directly observe how high-level C/C++ constructs descend through compilation stages, closing the theory-practice gap.
- Security Research: CFG visualization and assembly-level analysis provide a glass-box environment for auditing compiler-generated code, validating security flags, and detecting vulnerability surfaces.
- Embedded/IoT Engineering: LLVM pass-level analysis and instruction budgeting assist engineers in maintaining tight code-size and power-consumption constraints for production systems.

Future Scope:

- Extended Language Support: Add Rust, Swift, and Zig via LLVM backend.
- Runtime Profiling & Dynamic Analysis: Execution-time measurement, cache behavior analysis, hardware performance counters.
- Advanced CFG Visualization: Zoomable interactive CFG views with node-click AI explanations.
- Distributed Compilation & Scalability: Kubernetes-based pod pre-warming and load balancing.
- Multi-file Project Support and Collaborative Features: Git integration, version history, annotation tools.