Enhanced Dead Block Analysis with Dynamic Resource Allocation and Parallel Execution

Mohammad Nuruddin, Rakesh Das
dept. of Computer Science, *Texas State University*
San Marcos, Texas
ysx4@txstate.edu, atf58@txstate.edu

***Abstract*—Optimizing compilers are pivotal in translating high-level programming code into efficient machine-executable instructions. However, detecting overlooked optimizations automatically remains a formidable challenge, often constrained by program analysis complexity or limited applicability. This project presents a methodology for systematically identifying a broad spectrum of missed optimizations, centering on dead code elimination (DCE) as a core technique. Our approach introduces "optimization markers" strategically placed within program basic blocks, facilitating analysis of live and dead code segments. By evaluating the compiler's efficacy in eliminating dead code, our method provides a systematic framework to pinpoint and rectify overlooked optimization opportunities. Unlike existing methods, our approach offers a comprehensive means to assess compiler optimization efficacy, enhancing program performance across diverse computational tasks and programming languages.**

I. Introduction

Both academia and industry have dedicated substantial efforts to enhancing compiler optimizations over decades. However, compilers often encounter performance bugs, termed missed optimization opportunities, where they generate inefficient code despite potential for improvement. Pinpointing these bugs is challenging and typically relies on manual analysis, which is time-consuming and error-prone. Existing approaches like benchmarking and automatic testing have limitations in detecting these bugs comprehensively. As a result, addressing missed optimization opportunities remains a manual and intricate process, combining measurement and code analysis.

This project work introduces a method for automatically detecting missed optimization opportunities in compilers. By strategically inserting markers into program code, the technique probes compiler analyses and optimizations, leveraging dead code elimination (DCE) as a benchmark. Differential testing between compilers or optimization levels reveals missed opportunities, pinpointing their locations within code blocks. Empirical analysis on GCC and LLVM reveals significant numbers of missed opportunities and performance bugs, demonstrating the effectiveness of the approach. Key contributions include a widely applicable approach, implementation, and empirical evaluation.

II. Theoretical Explanation

This work presents a effective approach for detecting missed optimization opportunities in compilers. By leveraging dead code elimination (DCE), the approach identifies instances where a compiler fails to optimize code unexpectedly. Optimization markers are introduced to each basic block of a test case, facilitating comparison between differently optimized versions compiled with different compilers or optimization levels. The method offers a systematic way to identify and address missed opportunities, leading to bug-inducing test cases for further analysis. Detailed steps are outlined in the report, highlighting its efficacy in detecting and addressing optimization shortcomings.

A. Compiler

A compiler is a specialized software tool that facilitates the translation of high-level programming code, such as C, C++, Java, or Python, into machine code, enabling execution by a computer's processor Fig:1. It serves as an intermediary, bridging the gap between human-readable source code and machine-executable binary code.

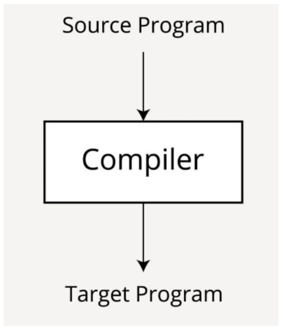


Fig. 1. Compiler Design.

GCC (GNU Compiler Collection) and LLVM (Low Level Virtual Machine) stand out as prominent open-source compiler infrastructures, each offering a comprehensive suite of compilers catering to diverse programming languages.

GCC, known as one of the oldest and most prevalent compiler collections globally, encompasses a range of compilers tailored for specific languages. For instance, it includes gcc for C, g++ for C++, and gfortran for Fortran.

On the other hand, LLVM emerges as a newer compiler infrastructure, originating from the University of Illinois at Urbana-Champaign and subsequently embraced by Apple Inc. It operates with an intermediate representation (IR) termed LLVM IR Fig:2, crafted to be platform-agnostic, facilitating a spectrum of high-level optimizations.

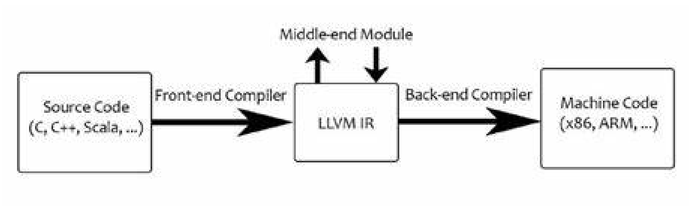

Middle-end Module
Source Code (C, C++, Scala, ...)
Front-end Compiler
LLVM IR
Back-end Compiler
Machine Code (x86, ARM, ...)

Fig. 2. Low Level Virtual Machine.

GCC and LLVM differ significantly in their architectural designs.GCC adopts a monolithic architecture, where the entire compiler suite is tightly integrated into a single executable. This centralized design simplifies management but may limit flexibility.

In contrast, LLVM employs a modular architecture, separating components like frontends, optimizers, and backends. These components are decoupled and communicate through a common intermediate representation (LLVM IR). This modular approach offers enhanced flexibility and facilitates easier integration into various projects.

B. GNU Parallel

GNU Parallel is a command-line utility tailored for Unix-like operating systems. It facilitates parallel execution of shell commands and scripts, enabling efficient distribution of tasks across multiple CPU cores or systems. Its versatility makes it invaluable for tasks like data processing, file manipulation, and computational tasks that benefit from parallelization.

C. Dynamic Load Balancing

Dynamic load balancing is a vital technique employed in distributed computing systems to ensure equitable distribution of computational tasks or workload across multiple processing units like CPUs, GPUs, or servers. It operates in real-time, dynamically adjusting resource allocation to optimize utilization, minimize processing time, and maintain fairness across the system, even amidst dynamically changing workload characteristics.

The Producer-Consumer model provides flexibility in task generation and consumption, facilitating asynchronous processing and decoupling of production and consumption rates. This model is advantageous for scenarios with variable computational loads or dynamic task generation patterns. However, it introduces additional complexity in managing the shared task queue and coordinating producer-consumer interactions.

III. Experimental Process

The workflow for parallel compilation with dynamic load balancing involves several essential steps to optimize the compilation process while ensuring efficient resource usage and system responsiveness.

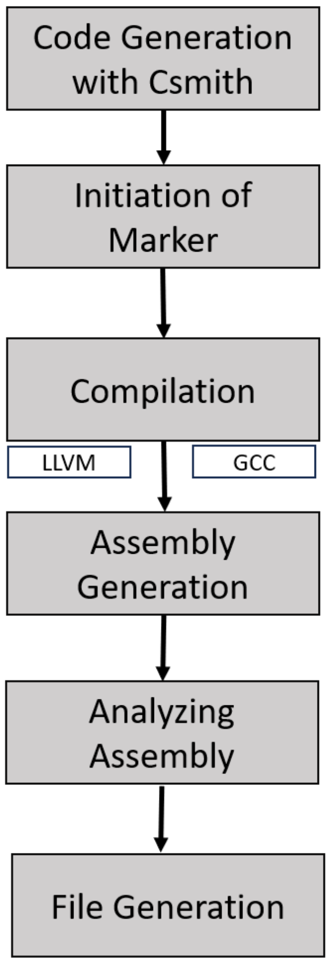


Fig. 3. Compilation and Analysis.

Initially, the process begins by concurrently generating and applying markers to multiple C source files. Utilizing GNU Parallel, the system runs multiple instances of the csmith command simultaneously, creating unique C source files. Following this, program markers are promptly applied to each file, ensuring they are ready for subsequent compilation. This parallel execution minimizes wait times and expedites file preparation.

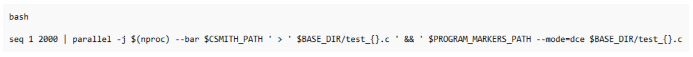
Next, during the compilation and analysis phase, the generated files undergo compilation with various GCC compilers and optimization flags. Each file is compiled with different GCC configurations, resulting in .s assembly files. These assembly files are then scanned for specific markers like DCEMarker, with the count of markers recorded for further analysis. Parallelization of compilation tasks accelerates the process significantly, allowing for swift analysis of multiple files.

Moreover, dynamic job adjustment is pivotal for adapting the workload to the current system load. By dynamically calculating the number of concurrent jobs based on the system's load, the workflow ensures optimal resource utilization while preventing overloading. This mechanism maintains system responsiveness by continuously monitoring the system load and adjusting the number of parallel jobs accordingly. The workflow of the project is pictured in Fig:3

Parallelization: The process involves breaking down a task that can be executed concurrently into multiple smaller tasks and then executing those tasks simultaneously across multiple processors. This is particularly effective for CPU-intensive tasks like compiling code, where each compilation process can run independently.

Dynamic Load Balancing: This concept refers to the system's ability to monitor and distribute workload dynamically across different processors based on their current load. The goal is to optimize resource utilization, avoid overloading any single processor, and reduce the overall execution time. Detailed Code Walkthrough

Generating and applying markers: The script uses GNU Parallel to execute multiple instances of the csmith command, each generating a unique C source file. For each file generated, the program markers are applied immediately in the same command chain, ensuring that the file is ready for compilation as soon as it is created Fig:4. This use of parallelism minimizes waiting times that would occur if each file had to be processed sequentially.

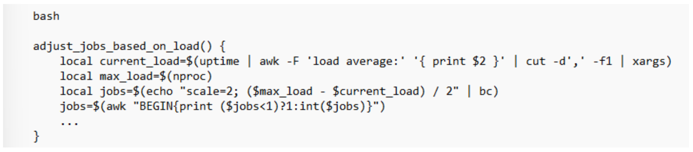


```
bash

seq 1 2000 | parallel -j $(nproc) --bar $CSMITH_PATH ' > ' $BASE_DIR/test_{}.c ' && ' $PROGRAM_MARKERS_PATH --mode=dce $BASE_DIR/test_{}.c
```

Fig. 4. Generating and applying markers.

Adjusting jobs based on system load: The script calculates the number of jobs to run simultaneously based on the current system load. This helps in adapting the workload according to the system's capability at any given moment, avoiding overloading and ensuring smooth execution Fig:5. The calculation ensures at least one job is always running while capping the maximum number of jobs to prevent overloading.



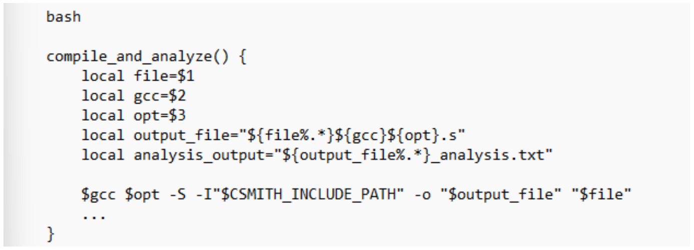
```
bash

adjust_jobs_based_on_load() {
    local current_load=$(uptime | awk -F 'load average:' '{ print $2 }' | cut -d',' -f1 | xargs)
    local max_load=$(nproc)
    local jobs=$(echo "scale=2; ($max_load - $current_load) / 2" | bc)
    jobs=$(awk "BEGIN{print ($jobs<1)?1:int($jobs)}")
    ...
}
```

Fig. 5. Dynamic Job Adjustment.

Compiling and analyzing: Each generated file is compiled with different GCC compilers and optimization flags. The compilation outputs are .s assembly files, which are then searched for specific markers (DCEMarker) Fig:6.The count of these markers is recorded, providing a metric for analysis.

In summary, the workflow orchestrates parallel compilation with meticulous attention to efficient resource allocation and workload balancing. By leveraging parallelism and dynamic load balancing techniques, the process optimizes compilation



```
bash

compile_and_analyze() {
    local file=$1
    local gcc=$2
    local opt=$3
    local output_file="${file%.*}${gcc}${opt}.s"
    local analysis_output="${output_file%.*}_analysis.txt"

    $gcc $opt -S -I"$CSMITH_INCLUDE_PATH" -o "$output_file" "$file"
    ...
}
```

Fig. 6. Compilation and Analysis.

times, enhances system performance, and ensures seamless execution under varying computational loads.

A. Comparison and Analysis

In the compilation process, we conducted optimization and dead block elimination on a varied set of programs, ranging from 200 to 5000 in number. For GCC, I employed versions 9, 10, 11, and 12, while for LLVM, versions 11, 12, 13, and 14 were utilized. Each program underwent compilation with different optimization flags:

-O0: This flag indicates no optimization, serving debugging purposes by generating un-optimized code.
-O1: Moderate optimization is enabled, incorporating most optimizations that do not compromise space-speed trade-offs.
-O2: Adopting a more aggressive optimization approach, this flag triggers additional optimizations that may augment compile time and code size.
-O3: At the highest optimization level, this flag implements even more aggressive optimizations like in-lining and loop unrolling.
-Os: This flag prioritizes code size optimization, aiming to reduce the size of the generated code while sacrificing some speed.
By employing this comprehensive compilation setup, we aimed to assess the impact of different optimization levels and dead block elimination techniques across a diverse range of programs and compiler versions. This detailed approach allowed for a thorough analysis of how varying optimization strategies and compiler versions influence the performance and size of compiled code.

From Fig:7 and and Fig:8 we observe why the dead block analysis for the modern compiler are important. For each of the compilers we adopted 4 versions of GCC and LLVM indicated earlier with 5 optimization flags with varying the number of codes from 200 to 5000. Now from Fig:7 we observe that for varying number of codes in different LLVM versions and optimization flags the number of dead blocks varied drastically. The LLVM version 11 and 12 with -O3 have the highest number of optimization opportunities missed and interestingly this issue is fixed in the later LLVM version which we can potentially observe in the version 13 and 14 that the number of optimization opportunities missed by flag -O3 has reduced down more than -O2. Interesting we found all the other optimization flags were consistent almost in every version we worked with. Among them the -O1 and -Os has the

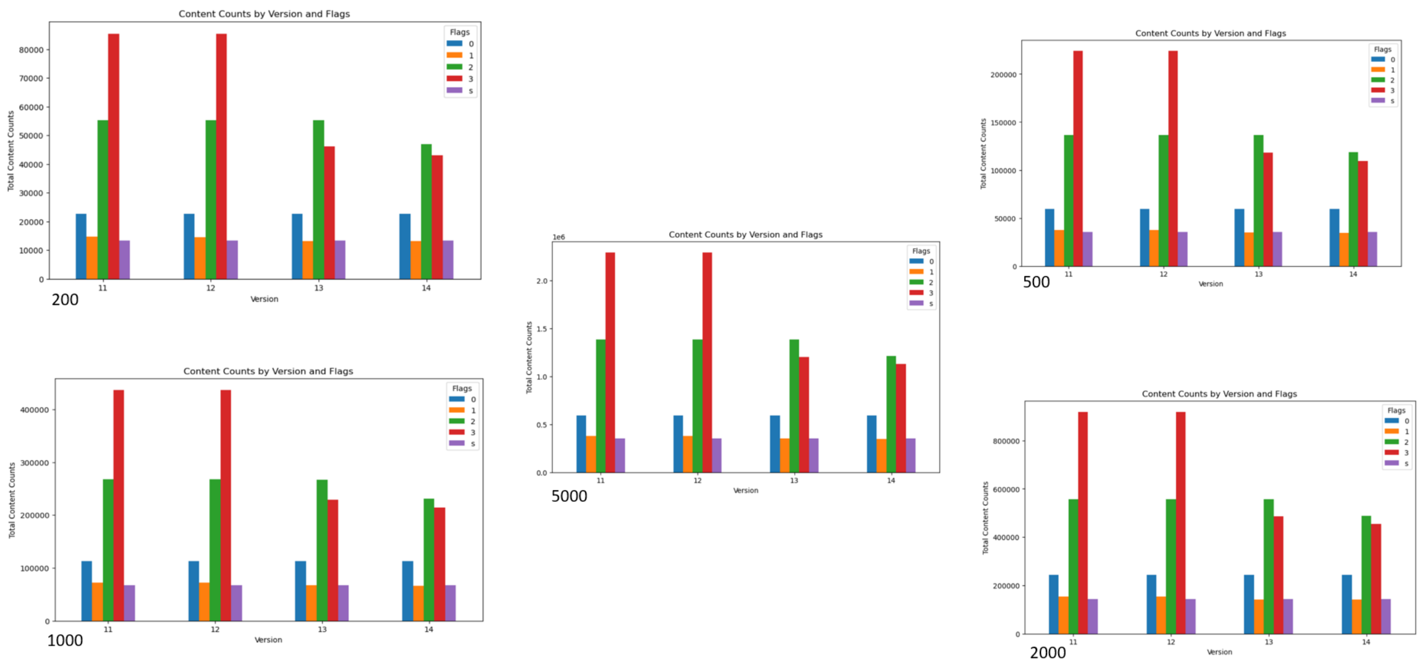


Fig. 7. Missed Marker for LLVM.

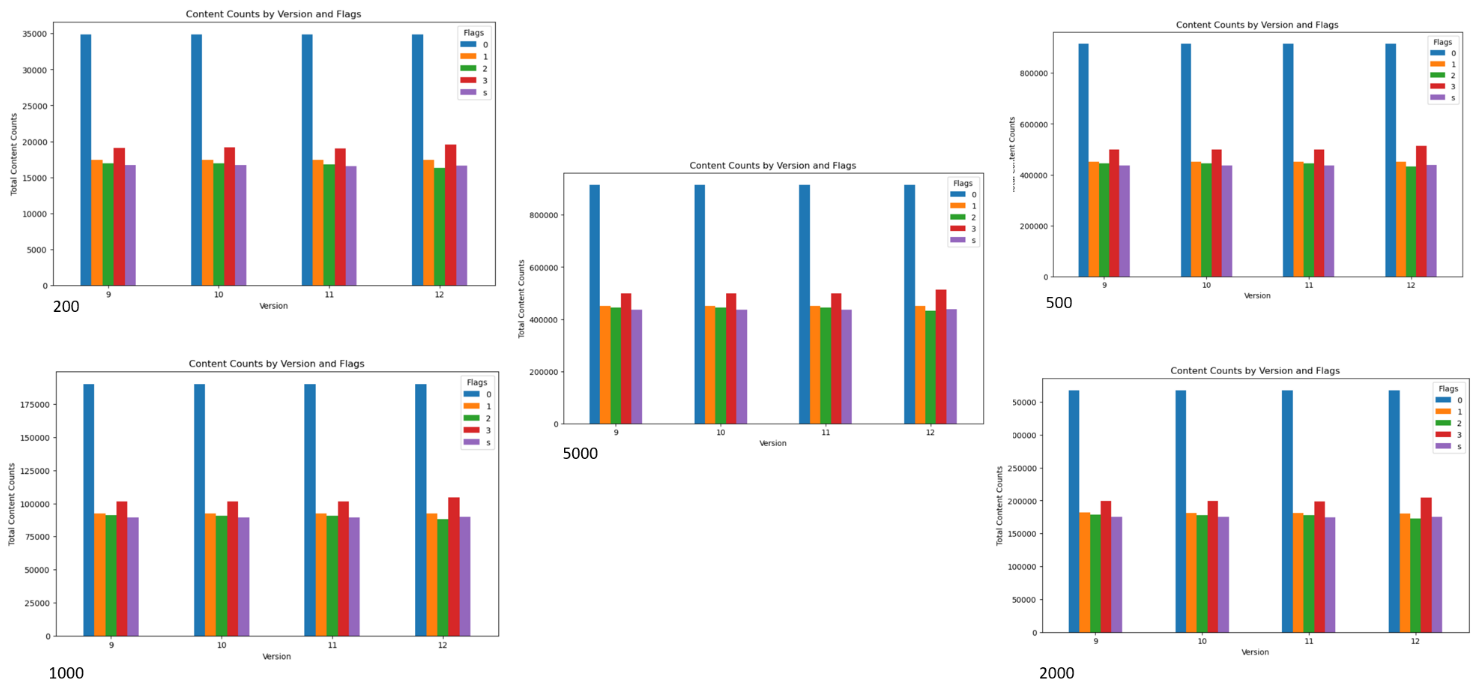


Fig. 8. Missed Marker for GCC.

lowest optimization opportunities missed indicating that they are most capable for eliminating the dead blocks in the final assembly file, still the number of optimization opportunity they miss is significant

In Fig:8 we observe a constant performance produced by each of the optimization flags in these four versions of GCC. Among them -O0 have most optimization missed and other optimization flags: -O1, -O2, -O3, and -Os has almost similar number of optimization opportunities missed. This indicates that the elimination capability of dead block in these later optimization flags after -O0 is better comparing to -O0

Now if we compare Fig:7 and Fig:8 we can observe a significant different in the capability of each compiler in eliminating the dead blocks and the potential winner here is the LLVM as three of the optimization flags: O0, O1 and Os showcased a great result in eliminating the dead blocks

Now as mentioned earlier we employed the dynamic resource allocation with task parallelism for both of the compilers from code generation to analysis of the assembly file for the counting of the dead block calls or the markers. In both of the figures Fig:9 and Fig:10 we observe that when we ran the program sequentially meaning without any further optimization the execution time was almost linear. For example in case of GCC we found out that for sequential analysis it was around 14769 seconds or 4.1 hours for 5000 programs to finish the analysis. But with the optimization policy we found out that for the same number of code the execution times was 3.48 hours 37.2 minutes less than the sequential version. In the beginning for small amount of code the dynamic load balancing and task parallelism had a little contribution in the execution time but as both of the relationship increased linearly the margin between them went higher and higher for each batch of codes which proofs our optimization attempts by utilizing the system load and parallelization efficiently we had a significant positive outcome on the analysis of the program markers.

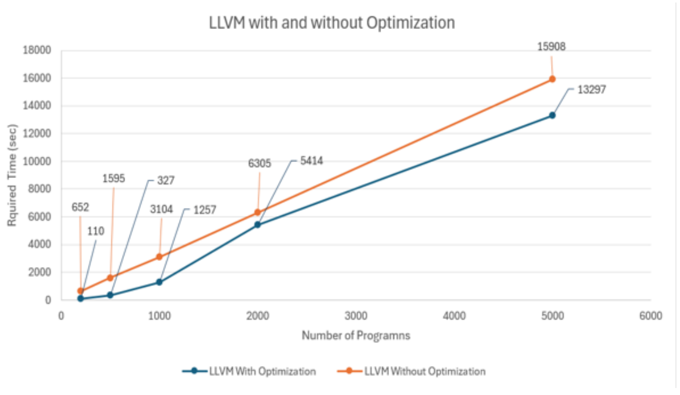


Fig. 9. LLVM optimization Num of Programs vs Required Time.

IV. Conclusion

In summary, the approach outlined in this project provides notable benefits in terms of efficiency, speed, scalability,

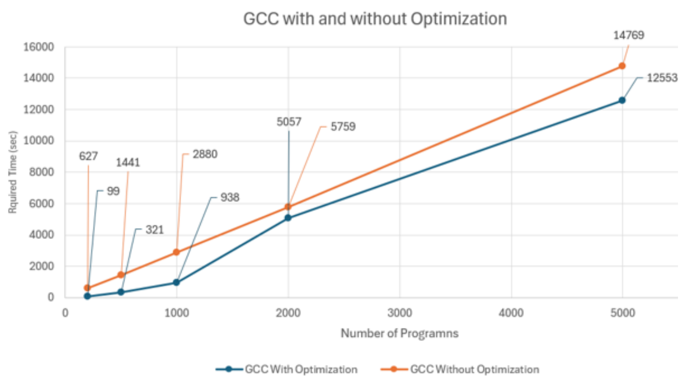


Fig. 10. GCC optimization Num of Programs vs Required Time.

and resource management. Through the utilization of parallel execution and dynamic load balancing techniques, the script effectively reduces idle CPU time and accelerates task completion compared to sequential execution. Furthermore, its scalability enables seamless adaptation to changes in system configuration or the addition of CPU cores, ensuring consistent performance across diverse environments. Additionally, dynamic load balancing facilitates the efficient utilization of system resources without causing system overload, thereby preserving stability and ensuring optimal performance. This method is particularly advantageous in environments requiring the efficient management of multiple intensive tasks without the need for manual intervention, thereby enabling the maximization of resource utilization at all times.