CODE OPTIMIZATION

Consideration for Optimization, Scope of Optimization, Optimization Techniques, Flow graph

S AL Ameen
192111075
M Shanmuk
192111080
G Bhanu Prakash
192210366

DESIGN OF A COMPILER

Compiler front-end: lexical analysis, syntax analysis, semantic analysis

Tasks: understanding the source code, making sure the source code is written correctly

Compiler back-end: Intermediate code generation/improvement, and Machine code generation/improvement

Tasks: translating the program to a semantically the same program (in a different language).

What is Code Optimization?

Optimization is a program transformation technique, which tries to improve the code that consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

A code optimizing process must follow the three rules given below:

1. The output code must not change the meaning of the program in any way.Should not change the output produced for any input

Should not introduce an error

2. Optimization should increase the speed of the program and if possible, the program should demand less resources.

3. Optimization should itself be fast and should not delay the overall compiling process

I**mprovements can be made at various phases:**

Source Code:

-Algorithms transformation can produce spectacular improvements

Intermediate Code:

-Compiler can improve loops, procedure calls and address calculations

-Typically only optimizing compilers include this phase

• **Target Code:**

- Compilers can use registers efficiently

Optimized code's features:

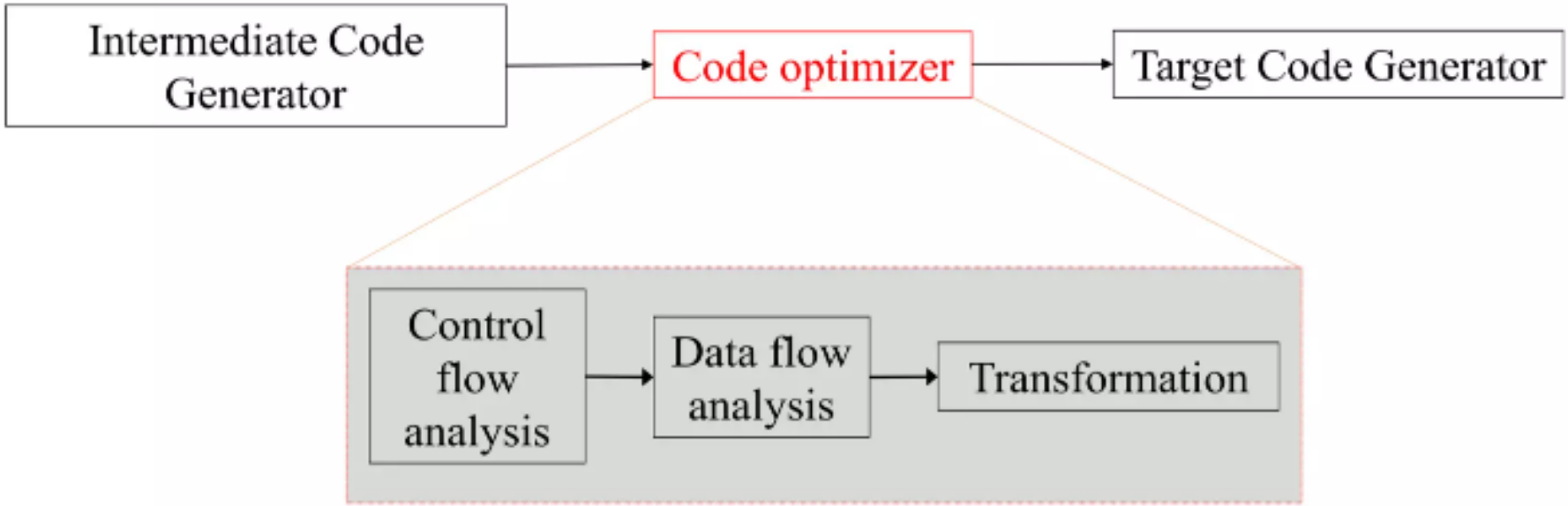
Executes faster

Code size get reduced

Efficient memory usage

Yielding better performance, Reduces the time and space complexity

Organization of an optimizing compiler



Flow analysis

- Organization of an optimizing compiler

Flow analysis is a fundamental prerequisite for many important types of code improvement

Generally control flow analysis precedes data flow analysis.

Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as

control flow graph-graphical representation of control flow or computation during the execution.

Call graph represents calling relationships between subroutines.

Data flow analysis (DFA) is the process of asserting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

A basic block begins in one of the following ways:

the entry point into the function.

the target of a branch (can be a label)

the instruction immediately following a branch or a return

A basic block ends in any of the following ways:

a jump statement

a conditional or unconditional branch

a return statement

Conclusion:

In the realm of compiler design, optimizing code is paramount for improving program efficiency, reducing resource consumption, and enhancing overall performance. Through the course of this project, we've explored various optimization techniques aimed at transforming source code into more efficient forms while preserving its functionality. These optimizations range from simple peephole optimizations to sophisticated global optimizations, each targeting specific aspects of code improvement.

Our investigation revealed that optimization is not a one-size-fits-all endeavor. Different programs exhibit distinct characteristics, and thus require tailored optimization strategies. Furthermore, the optimization process must strike a balance between execution time and the complexity of the optimization itself. While aggressive optimizations may yield significant performance gains, they often come at the cost of increased compilation time and code complexity.

Key findings from our exploration include:

Local Optimizations: These optimizations focus on small code segments and aim to eliminate redundant computations, utilize machine instructions more effectively, and reduce memory accesses. Techniques such as constant folding, common subexpression elimination, and strength reduction fall under this category.

Global Optimizations: Operating on a larger scale, global optimizations analyze inter-procedural dependencies and program-wide behavior to identify opportunities for optimization. Examples include loop optimizations, inlining, and inter-procedural analysis.

Machine-Dependent vs. Machine-Independent Optimizations: Depending on the target architecture, optimizations can be tailored to exploit specific features of the underlying hardware. However, care must be taken to ensure portability across different platforms.

Trade-offs: Optimization decisions often involve trade-offs between code size, execution speed, and compile time. Striking the right balance requires careful consideration of the target environment and the performance requirements of the application.

Compiler Flags and Options: Compiler flags and options play a crucial role in controlling the optimization process. Understanding these flags and their implications is essential for achieving desired performance goals.

THANK YOU