Lecture 1: Optimization

CESE4085 Modern Computer Architecture Course
Part 2, L7.1
Carlo Galuzzi

TUDelft

LECTURE CONTENTS

- Motivation for compiler optimizations
- Role of LLVM IR in optimization
- Common and advanced optimization passes
- LLVM tools and usage examples
- Concept and benefit of re-optimization
- Optimization Goals and Trade-offs
- Cost Models and Heuristics
- Optimization Correctness and Semantics Preservation

LEARNING GOALS:

Understanding optimizations is crucial for mastering compiler design and optimization techniques. By the end of this lecture, among other goals, you will be able to:

- Explain the motivation behind compiler optimizations and their importance in modern software development.
- Describe the structure and purpose of LLVM's intermediate representation (IR) and its role in the optimization pipeline.
- Identify and explain common optimization techniques, such as constant folding, dead code elimination, and loop-invariant code motion.
- Analyse LLVM IR code and recognize optimization opportunities.
- Understand how optimizations affect generated machine code, with a focus on the RISC-V architecture.
- Use LLVM tools to explore, apply, and inspect optimizations in practice.

EXTRA READING MATERIAL IN BRIGHSPACE

LLVM-Based Optimizations in Compiler Backends: Theory and Practice with C and RISC-V

1. The Role of Optimization in Compilation

Modern software must run efficiently across a variety of platforms, from embedded systems to data centers. When developers write programs in C, the code is typically straightforward and focused on readability or correctness, not performance. However, naïve compilation of this code often leads to binaries that perform redundant computations, overuse memory, or miss opportunities to take advantage of the hardware.

Compiler optimizations transform a program's intermediate representation into a more efficient version while preserving its behavior. These optimizations can reduce execution time, lower memory consumption, or decrease energy usage. Optimization is a core responsibility of the compiler backend, where intermediate code is lowered to a machine-specific form. This is particularly important for target architectures like RISC-V, where efficient use of registers and instructions can make a significant difference.

Optimizations are often categorized by their location in the compiler pipeline:

- **Frontend optimizations**: These occur early, often on the abstract syntax tree (AST) or high-level IR. They include basic transformations such as constant folding and syntax simplifications. For example, a frontend might simplify `3 + 4` directly to `7` during parsing.
- **Middle-end optimizations**: These operate on a target-independent intermediate representation (IR), which enables powerful analyses such as control flow, data flow, and alias analysis. Examples include loop-invariant code motion, dead code elimination, and common subexpression elimination. These optimizations are crucial for preparing IR that is both clean and semantically equivalent to the source.
- **Backend optimizations**: These are specific to the target machine and aim to improve final code generation. Examples include register allocation, instruction scheduling, and peephole optimizations. On a RISC-V backend, this might involve selecting instructions that minimize pipeline stalls or making efficient use of a small number of registers.

For instance, consider the expression:

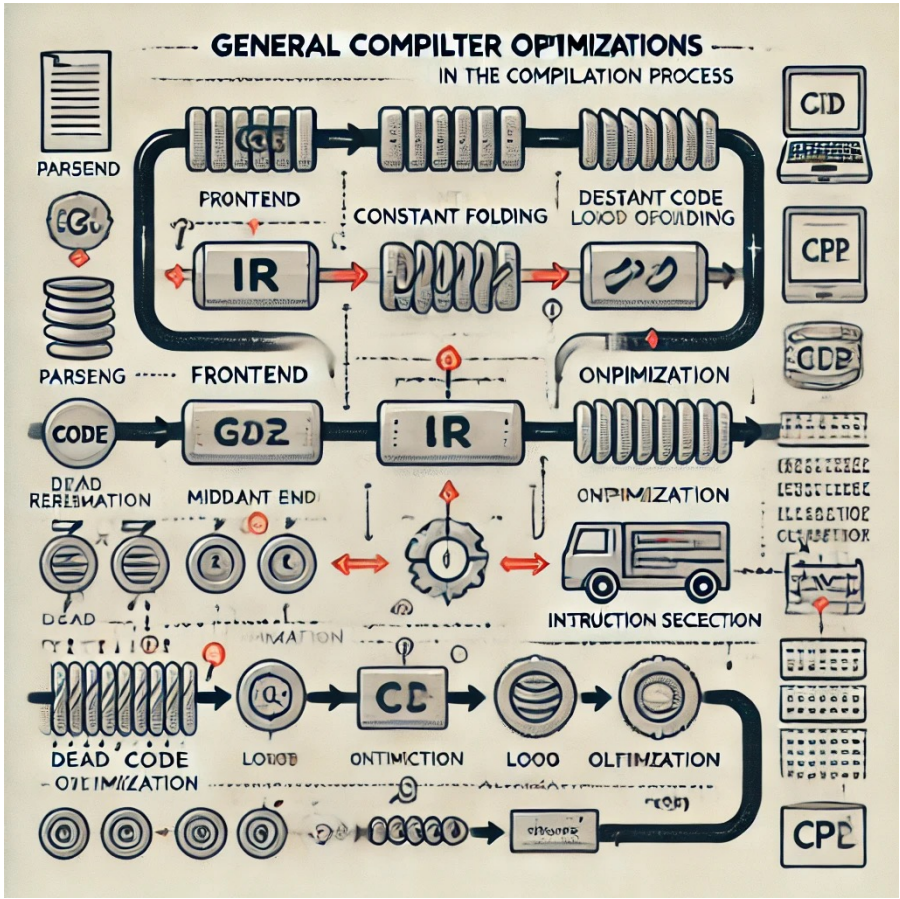
```
int x = (a + b) * (a + b);
```

A frontend might leave this as-is, but a middle-end optimization like common subexpression elimination will recognize that `(a + b)` is computed twice and store it in a temporary value. A backend optimizer might then schedule the instructions to avoid pipeline hazards on the target architecture.

LLVM performs many optimizations at the IR level using SSA (Static Single Assignment) form, where each variable is assigned exactly once. This representation makes dataflow analysis more straightforward and enables sophisticated optimizations that are difficult in other forms.

This report is structured to provide both theoretical foundations and practical insights into LLVM-based compiler optimizations. In the next section, we introduce the LLVM optimization pipeline, detailing how transformations are organized and executed. This is followed by a complete walk-through of a simple but illustrative example in C, which serves as a running example throughout the document to demonstrate the impact of various optimizations.

Optimization



INTRODUCTION TO COMPILER OPTIMIZATIONS

- Compiler optimizations are critical for transforming high-level code into efficient machine-level instructions. While programs written in languages like C are often easy for humans to read and understand, they are not always optimized for speed or size.
- This lecture explores how the LLVM compiler infrastructure performs transformations on intermediate representations to generate better-performing code.
- Understanding compiler optimizations gives insight into how source code is interpreted, analyzed, and improved by modern compilers.

Key Points:

- Modern software must run efficiently on many architectures.
- Human-readable code is not necessarily efficient for execution.
- LLVM provides structured and modular optimizations.

WHY OPTIMIZATION MATTERS

- Software runs on diverse platforms, from smartphones to high-performance servers. **The same source code may behave very differently depending on how efficiently it is compiled**.
- **Without optimization**, compilers may emit code that consumes **excessive** memory, executes more slowly, or drains battery life faster.
- Compiler optimizations help bridge this gap by **transforming** the internal representation of code into a more efficient version.

Benefits of Optimization:

- Reduce the number of instructions executed at runtime.
- Minimize memory access and improve cache locality.
- Exploit features of the underlying hardware architecture (e.g., pipelining, vector units).
- Increase battery life on mobile devices or reduce energy costs in large data centers.

OPTIMIZATION IN THE COMPILATION PIPELINE (1)

- Compiler pipelines are typically structured into multiple stages. Optimizations can occur at different points, each with distinct goals and methods.
- LLVM organizes these into **frontend, middle-end, and backend** stages.

Stages of Optimization:

- **Frontend Optimizations:**
  - These operate on high-level structures like the abstract syntax tree (AST).
  - **Examples**: constant folding, algebraic simplifications.

OPTIMIZATION IN THE COMPILATION PIPELINE (2)

- Compiler pipelines are typically structured into multiple stages. Optimizations can occur at different points, each with distinct goals and methods.
- LLVM organizes these into **frontend, middle-end, and backend** stages.

Stages of Optimization:

- **Middle-end Optimizations:**
  - These work on the intermediate representation (IR).
  - **Examples**: loop-invariant code motion, dead code elimination, common subexpression elimination.

OPTIMIZATION IN THE COMPILATION PIPELINE (3)

- Compiler pipelines are typically structured into multiple stages. Optimizations can occur at different points, each with distinct goals and methods.
- LLVM organizes these into **frontend, middle-end, and backend** stages.

Stages of Optimization:

- **Backend Optimizations:**
  - These target machine-specific transformations (e.g., RISC V).
  - **Examples**: instruction scheduling, register allocation, peephole optimizations.

Why This Matters: Knowing where an optimization occurs helps developers understand **its scope, impact, and limitations**.

EXAMPLE OF STAGE INTERACTION

Let's consider a simple expression and examine how it is optimized at different stages of the compiler pipeline.

Example Code: `int x = (a + b) * (a + b);`

How Different Stages Contribute:

- **Frontend:** Parses the code and generates the corresponding abstract syntax tree and then the IR.
- **Middle-end:** Detects that the expression `(a + b)` is computed twice and can be replaced with a temporary variable to avoid recomputation.
- **Backend:** Once simplified, the backend schedules the final instructions efficiently to avoid pipeline stalls and optimize register usage.

Takeaway: Optimizations work together across compiler stages. **Even a simple arithmetic expression can benefit from multiple layers of transformation.**

LLVM INTERMEDIATE REPRESENTATION (IR) (1)

- **LLVM IR** is a low-level, strongly-typed programming language used internally by LLVM for code analysis and transformation. **It serves as the main vehicle for optimizations.**
- The IR is **platform-independent** and provides a consistent foundation for implementing powerful analyses and transformations.

Key Properties:

- **Static Single Assignment (SSA):** Each variable is assigned exactly once, making dataflow analysis simpler.
- **Three-address code:** Instructions usually have a single operation and two operands.
- **Typed system:** All variables and operations are strongly typed, which helps with correctness and optimization.

LLVM INTERMEDIATE REPRESENTATION (IR) (2)

Benefits for Optimization:

- SSA form enables efficient and precise dataflow tracking.
- **Platform independence** allows optimizations to be **reused** across architectures.
- IR can be transformed, verified, and reoptimized multiple times before generating machine code.

Example:

```
%1 = add i32 %a, %b
%2 = mul i32 %1, %1
```

This IR snippet shows a simple arithmetic computation in SSA form.

LLVM OPTIMIZATION PIPELINE

- LLVM uses a modular pass-based system to apply optimizations.
- Each pass performs a specific analysis or transformation over the IR.
- These passes are scheduled in a pipeline that can be customized or automatically configured by optimization levels.

Key Interfaces for Optimization:

- **clang**: Compile C/C++ source code and invoke the optimization pipeline with flags like `-O1`, `-O2`, or `-O3`.
- **opt**: Apply individual passes explicitly for analysis and debugging.
- **Optimizations** happen mostly at the IR level (**middle-end**), but backend optimizations are also applied later.

Why This Matters: Fine-grained control over passes helps in both **understanding and debugging performance issues**.

MOTIVATING EXAMPLE — THE `sum` FUNCTION

- Consider a simple C function that accumulates the values in an array:

```
int sum(int *A, int n) {
    int total = 0;
    for (int i = 0; i < n; i++)
        total += A[i];
    return total;
}
```

- Although this code is short and readable, the unoptimized machine code can be far from efficient.

We use this function as a running example throughout the lecture.

Takeaway: Even simple code can benefit from multiple optimizations.

UNOPTIMIZED LLVM IR (EXCERPT)

At `-O0`, LLVM generates IR that maps closely to the C source:

```
%total = alloca i32, align 4
store i32 0, i32* %total
...
%0 = load i32, i32* %total
%1 = load i32, i32* %arrayidx
%add = add i32 %0, %1
store i32 %add, i32* %total
```

- Variables are stored on the stack using `alloca`, and values are accessed via explicit loads and stores.

Drawbacks:

- Stack-based operations are slower than register-based.
- Redundant memory traffic limits performance.

OPTIMIZED LLVM IR AFTER `mem2reg`

After applying `mem2reg`, stack memory is replaced by SSA variables:

```
%total = phi i32 [0, %entry], [%add, %loop]
%add = add i32 %total, %val
```

- The use of `phi` nodes [1] ensures proper value selection at control-flow joins.
- No memory operations are needed within the loop body.

Advantages:

- Fewer instructions and better register usage.
- Enables further optimizations like loop unrolling or CSE.

[1] Phi nodes are a key feature of the **Static Single Assignment (SSA)** form used in LLVM IR. They allow the IR to represent variables that can take on different values depending on which **control flow path** was taken to reach a certain point in the program.

OPTIMIZED RISC-V CODE (SIMPLIFIED)

The final machine code, generated after optimizations and transforming code from a higher-level representation to a more concrete, lower-level one — closer to machine code:

```
li a2, 0
li a1, 0
loop:
  bge a1, a0, exit
  slli a3, a1, 2
  add a4, a3, a5
  lw a4, 0(a4)
  add a2, a2, a4
  addi a1, a1, 1
  j loop
exit:
  mv a0, a2
  ret
```

- Each instruction is useful — no redundant memory ops.

Takeaway: High-level optimizations translate directly into faster and cleaner machine code.

STRUCTURE OF OPTIMIZATION SLIDES

- In the following slides, we will analyze individual optimization techniques used by LLVM in detail.
- For each optimization, the structure will be consistent to help with clarity and comparison:

Each slide will include:

- **Explanation:** What the optimization does and why it's useful.
- **Example:** A simple C code snippet, with LLVM IR before and after the transformation.
- **Impact:** A short discussion of the benefits in terms of performance or simplification.

LLVM Tooling: In the Extra Document (see Brightspace), you can see how to trigger or observe the optimization using LLVM tools like `opt`.

This structure will help reinforce both **theoretical** understanding and **practical** relevance.

CONSTANT FOLDING

- Constant folding is an early optimization where constant expressions are evaluated at compile time.
- This reduces runtime computation and simplifies the IR.

Example: `int x = 3 * 4; // becomes: int x = 12;`

LLVM IR before:

```
%mul = mul i32 3, 4
store i32 %mul, i32* %x
```

LLVM IR after constant folding:

```
store i32 12, i32* %x
```

Benefits:

- Reduces runtime instructions.
- Exposes further optimization opportunities (e.g., **dead code elimination**).

DEAD CODE ELIMINATION (DCE)

- DCE removes instructions that compute values never used.
- Helps shrink the IR and reduce memory and register pressure.

Example:

```
int x = 5;
x = 6; // The value 5 is never used
```

LLVM IR before DCE:

```
store i32 5, i32* %x
store i32 6, i32* %x
```

LLVM IR after DCE:

```
store i32 6, i32* %x
```

Benefits:

- Reduces instruction count.
- Improves runtime efficiency and compile-time clarity.

COMMON SUBEXPRESSION ELIMINATION (CSE)

- CSE finds identical computations and reuses their results.
- Useful when expressions like `(a + b)` appear multiple times.

Example:

```
int x = (a + b) * (a + b);
```

Naïve IR:

```
%1 = add i32 %a, %b
%2 = add i32 %a, %b
%3 = mul i32 %1, %2
```

LLVM IR after CSE:

```
%1 = add i32 %a, %b
%2 = mul i32 %1, %1
```

Benefits:

- Reduces redundant ALU operations.
- Simplifies control and data dependencies.

LOOP-INVARIANT CODE MOTION (LICM)

- LICM moves expressions that do not change inside a loop to outside the loop.

Example:

```
for (int i = 0; i < n; i++) {
    int y = x * 2;
    A[i] = y + i;
}
```

Optimized Version

```
int y = x * 2;
for (int i = 0; i < n; i++) {
    A[i] = y + i;
}
```

Why it works:

- `x * 2` is computed only once instead of every iteration.
- Reduces work in the loop body.

STRENGTH REDUCTION

- Strength reduction replaces expensive operations (like multiplication) with cheaper ones (like addition or shifts).

Example:

```
for (int i = 0; i < n; i++) {
    A[i] = i * 2;
}
```

LLVM IR before:

```
%mul = mul i32 %i, 2
```

LLVM IR after Strength reduction:

```
%shl = shl i32 %i, 1
```

Benefits:

- Shifts are faster than multiplies on most architectures.
- Especially helpful in tight loops.

REGISTER PROMOTION (`mem2reg`)

- Many optimizations are unlocked when variables are moved from memory (stack) to virtual registers in SSA form.
- mem2reg eliminates explicit alloca, load, and store instructions.

Example:

```
%a = alloca i32
store i32 5, i32* %a
%1 = load i32, i32* %
```

After `mem2reg`:

```
%a = 5
```

Impact:

- Simplifies IR, improves performance.
- Enables further optimizations like DCE, LICM, CSE.

LOOP UNROLLING

- Loop unrolling duplicates the loop body to reduce branching overhead and expose parallelism.
- It is particularly effective for loops with small, known bounds.

Example:

```
for (int i = 0; i < 4; i++) {
    A[i] = B[i] + C[i];
}
```

Unrolled:

```
A[0] = B[0] + C[0];
A[1] = B[1] + C[1];
A[2] = B[2] + C[2];
A[3] = B[3] + C[3];
```

Impact:

- Reduces loop control instructions.
- Improves instruction-level parallelism.

VECTORIZATION

- Vectorization transforms scalar operations into vector (SIMD) operations.
- Can dramatically speed up performance on hardware with vector units.

Example:

```
for (int i = 0; i < 4; i++) {
    A[i] = B[i] + C[i];
}
```

After vectorization (conceptually):

```
A[0:3] = B[0:3] + C[0:3];
```

Impact:

- Executes multiple data operations in parallel.
- Highly beneficial for data-parallel tasks.

TAIL CALL ELIMINATION

- Tail call elimination (TCE) is an optimization that replaces a recursive call that appears in the tail position (i.e., the last action in a function) with a direct jump.
- This means that no new stack frame is created, allowing recursion to behave like iteration.

```
int fact(int n, int acc) {
    if (n == 0) return acc;
    return fact(n - 1, n * acc); // tail call
}
```

Why It Matters:

- Recursive functions are elegant but typically use more stack space with each call.
- If not optimized, deep recursive calls can cause stack overflows.
- With TCE, recursive functions can be as efficient as loops, enabling more expressive coding without performance penalties. Examples are factorial, Fibonacci (tail-recursive), tree traversal, etc.

FUNCTION INLINING

- Function inlining replaces a function call with the actual body of the function.
- Helps eliminate call overhead and enables local optimizations.

Example:

```
int square(int x) { return x * x; }
int y = square(5);
```

After inlining:

```
int y = 5 * 5;
```

Impact:

- Reduces function call overhead.
- Enables further simplification and folding.
- Exposes more instructions to subsequent optimizations such as CSE or LICM.

INSTRUCTION COMBINING

- This optimization looks for patterns of instructions that can be simplified or replaced with a more efficient equivalent.
- It is a local simplification process that improves IR clarity and reduces instruction count.

Example:

```
%1 = add i32 %a, 0    ; Adding zero is unnecessary
```

After combining:

```
%1 = %a
```

Benefits:

- Reduces unnecessary computations.
- Simplifies code, helping other passes perform better.

GLOBAL VALUE NUMBERING (GVN)

- GVN is a technique that identifies and eliminates redundant expressions across basic blocks.
- It assigns a unique number to expressions based on their computed value and replaces equivalent ones.

Example:

```
int x = a + b;
...
int y = a + b;
```

Optimized version:

```
int x = a + b;
...
int y = x;
```

Benefits:

- Avoids recomputation of values.
- Reduces register pressure and instruction count.
- Enhances performance, especially in large functions.

GLOBAL VALUE NUMBERING (GVN)

- GVN is a technique that identifies and eliminates redundant expressions across basic blocks.
- It assigns a unique number to expressions based on their computed value and replaces equivalent ones.

Example:

```
int x = a + b;
...
int y = a + b;
```

Optimized version:

```
int x = a + b;
...
int y = x;
```

Benefits:

- Avoids recomputation of values.
- Reduces register pressure and instruction count.
- Enhances performance, especially in large functions.

ADVANCED: CODE HOISTING

Code hoisting is a control-flow optimization that moves expressions out of conditional branches or deeper blocks if they're always executed. This reduces duplication and improves instruction cache efficiency.

Key Points:

- Moves computation to dominator blocks.
- Often used together with loop optimizations.

```
if (x > 0) {
    y = a + b;
    ...
} else {
    y = a + b;
    ...
}
```

```
y = a + b;
if (x > 0) {
    ...
} else {
    ...
}
```

Benefits: Reduces code duplication, simplifies control flow, and exposes more optimization opportunities (like CSE).

ADVANCED: PREDICATE SIMPLIFICATION

Predicate simplification reduces the complexity of conditions in branches or loops when constant conditions or redundant expressions are found.

Key Points:

- Uses constant propagation and dead branch elimination.
- Simplifies control decisions.

Example:

```
if (a > 5 && a > 3)
```

```
if (a > 5)
```

Benefits:

- Reduces number of branches.
- Improves branch prediction and pipeline behavior.

ADVANCED: BRANCH ELIMINATION AND FOLDING

When the outcome of a conditional branch is statically known, the branch can be eliminated or simplified.

Key Points:

- Applied after constant propagation or condition simplification.
- Also folds identical branches together.

Example:

```
if (true) { x = 5; } else { x = 6; }  ➡  x = 5;
```

Impact:

- Smaller IR.
- Enables further CFG simplification.

ADVANCED: SPECULATIVE EXECUTION

Speculative execution hoists computations out of conditional blocks even when not guaranteed to be needed, assuming they're safe and cheap.

Key Points:

- Trades extra computation for simpler control flow.
- Risk: wasted work if branch isn't taken.

Example:

Move an expression out of an if block assuming no side effects.

Benefits:

- Better instruction-level parallelism.
- Improves CPU pipeline usage.

ADVANCED: LOOP FUSION

Loop fusion merges adjacent loops that iterate over the same range, reducing loop overhead and improving cache usage.

Example:

```
for (int i = 0; i < n; i++) A[i] = B[i] + 1;
for (int i = 0; i < n; i++) C[i] = A[i] * 2;
```

Becomes:

```
for (int i = 0; i < n; i++) {
   A[i] = B[i] + 1;
   C[i] = A[i] * 2;
}
```

Benefits:

- Better data locality.
- Reduces control overhead and increases parallelism.

ADVANCED: LOOP FISSION (DISTRIBUTION)

Opposite of loop fusion: splits a loop into multiple loops when beneficial (e.g., due to memory access patterns or parallelization needs).

Example:

```
for (int i = 0; i < n; i++) {
   A[i] = B[i] + 1;
   C[i] = D[i] * 2;
}
```

Becomes:

```
for (int i = 0; i < n; i++) A[i] = B[i] + 1;
for (int i = 0; i < n; i++) C[i] = A[i] * 2;
```

Why Split?

- Avoids cache conflicts.
- Improves parallelization opportunities.

CORE OPTIMIZATIONS

Core techniques include:

- Constant Folding: Compute constants at compile time.
- Dead Code Elimination: Remove unused instructions.
- Loop-Invariant Code Motion: Move stable values outside loops.
- Common Subexpression Elimination: Eliminate redundancy.
- Strength Reduction: Replace expensive operations with cheaper ones.
- Register Promotion: Convert memory to SSA.

These form the backbone of any optimized pipeline.

ADVANCED OPTIMIZATIONS

Beyond the core, LLVM provides powerful transformations that enable deeper improvements.

Key advanced techniques:

- Loop Unrolling: Reduce control overhead and increase instruction-level parallelism.
- Vectorization: Exploit SIMD hardware for data-parallel tasks.
- Function Inlining: Remove call overhead and enable further optimization.
- Tail Call Elimination: Turn recursion into iteration.
- Global Value Numbering: Track and reuse equivalent expressions.
- CFG Simplification: Flatten and streamline control flow.

These optimizations are often layered after core clean-up passes.

SUMMARY OF OPTIMIZATION TYPES (1)

Here's a summary of the optimizations we've covered, organized by the phase of the compiler pipeline in which they typically occur.

COMPILER STAGE	OPTIMIZATION TYPE	DESCRIPTION
Frontend	Constant Folding	Evaluate constant expressions at compile time
Frontend	Function Inlining	Replace function calls with the function body
Middle-end	Dead Code Elimination (DCE)	Remove code that does not affect program behavior
Middle-end	Common Subexpression Elimination	Reuse identical expressions to avoid recomputation
Middle-end	Loop-Invariant Code Motion (LICM)	Move loop-independent code outside the loop
Middle-end	Strength Reduction	Replace expensive ops with cheaper equivalents

SUMMARY OF OPTIMIZATION TYPES (2)

Here's a summary of the optimizations we've covered, organized by the phase of the compiler pipeline in which they typically occur.

COMPILER STAGE	OPTIMIZATION TYPE	DESCRIPTION
Middle-end	Register Promotion (mem2reg)	Replace memory variables with SSA registers
Middle-end	Instruction Combining	Simplify instruction patterns
Middle-end	Global Value Numbering (GVN)	Identify equivalent values across blocks
Backend	Tail Call Elimination	Reuse stack frames for tail-recursive calls
Backend	Loop Unrolling	Duplicate loop body to reduce control overhead
Backend	Vectorization	Convert scalar ops to SIMD
Backend	CFG Simplification	Remove unreachable or redundant branches

WHAT IS REOPTIMIZATION?

- Re-optimization is the process of reapplying optimization passes after initial transformations have been made.
- It allows the compiler to take advantage of new optimization opportunities that emerge after earlier changes.

Why It Matters:

- Many optimizations enable further optimizations.
- For example, after `mem2reg`, many loads/stores become removable by DCE.

Key Idea:

- Optimizations should not be applied in isolation. Instead, passes are run in phases to maximize their cumulative effect.

RE-OPTIMIZATION IN PRACTICE

- Re-optimization is not a single technique but a design philosophy in compiler pipelines.

Typical pipeline design:

- **Early passes**: Simplify and clean IR (e.g., constant folding, mem2reg).
- **Middle passes**: Apply major transformations (e.g., GVN, LICM).
- **Late passes**: Polish and refine (e.g., instcombine, DCE, CFG simplification).

Benefits of re-optimization:

- Higher-quality final code.
- Greater coverage of optimization opportunities.
- Maintains performance across a wide variety of input programs.

EXAMPLE OF REOPTIMIZATION

Let's look at a case where one optimization reveals new opportunities for another.

Code:

```
int x = a + b;
...
int y = a + b;
```

- Initially, both additions are treated independently.
- After common subexpression elimination (CSE), y is replaced with x.
- Then, dead code elimination might remove one of the now-unused operations.

Takeaway:

- Optimizations build on each other.
- Re-optimization ensures no opportunity is missed due to pass ordering.

MISSED OPPORTUNITIES WITHOUT RE-OPTIMIZATION

- Without re-optimization, the compiler might stop after applying just one round of transformations.

Consequences:

- Redundant computations that could have been eliminated persist.
- Dead code introduced by transformations remains.
- The final machine code is less efficient.

Takeaway: Re-optimization is essential for maximizing the effectiveness of a compiler's optimization strategy.

OPTIMIZATION GOALS AND TRADE-OFFS (1)

Compiler optimization isn't just about "making things faster." Optimizations must balance multiple, sometimes conflicting goals like performance, size, power, and portability. Understanding these trade-offs is key for designing effective optimization strategies.

Common Optimization Goals:

- **Execution Speed**: Reduce instruction count, improve pipeline usage.
- **Binary Size**: Shrink executable for constrained devices (e.g., -Os, -Oz).
- **Power Efficiency**: Avoid energy-expensive instructions (important in mobile/embedded).
- **Compilation Time**: Keep build time reasonable in large codebases.
- **Maintainability & Debuggability**: Avoid aggressive transformations in debug builds.

OPTIMIZATION GOALS AND TRADE-OFFS (2)

Trade-Off Examples:

Inlining for speed vs code size:

```
int square(int x) { return x * x; }
int y = square(5);  // May inline or not based on goals
```

Loop unrolling:

+ Improves ILP and branch prediction.
- Increases code size—bad for instruction cache.

Function inlining and tail-call elimination:

+ Improves performance.
- Makes stack traces harder to understand during debugging.

Takeaway: Optimization is goal-driven. A change that helps one metric may hurt another—compiler designers must **prioritize**!

COST MODELS AND HEURISTICS (1)

Compilers use cost models and heuristics to decide **when** and **how** to apply optimizations. These models estimate the performance impact of transformations based on static analysis or runtime profiling.

Static Cost Models:

- Estimate based on instruction counts, loop nesting, register pressure.
- Use architecture-specific weights for operations (e.g., mul vs add).
- Trade-off complexity vs speed: Cost model itself must be efficient.

Example Heuristic:

- Inline if:
- Function is small (≤ 10 instructions)
- Called frequently
- Not recursive

COST MODELS AND HEURISTICS (2)

LLVM Example: Inlining Cost Heuristic

```
; Only inlined if expected to save more than N cycles
define i32 @foo() #0
attributes #0 = { inlinehint }
```

Heuristic-based passes:

- **Loop unrolling:** Decide unroll factor based on estimated benefit.
- **GVN/CSE:** Skip if recomputation is cheaper than storing result.

Why Cost Models Are Hard? Actual runtime behavior depends on:

- CPU architecture
- Input data
- Cache and branch predictor state

Takeaway: Heuristics aren't perfect but necessary—they help keep compile times fast while still yielding solid results.

OPTIMIZATION CORRECTNESS AND SEMANTICS PRESERVATION (1)

Optimizations must **preserve the semantics** of the original program. Even transformations that look "obviously better" can be incorrect if they assume too much.

Challenges in Maintaining Correctness:

- Undefined behavior: Can expose latent bugs.
- Side effects: Must avoid reordering operations with visible impact.
- Floating point math: Naive transformations may violate precision or ordering.

Example of Risk: `x / 2 + y / 2  !=  (x + y) / 2`

This may not hold for signed integers due to rounding behavior.

OPTIMIZATION CORRECTNESS AND SEMANTICS PRESERVATION (2)

LLVM's Strategies for Ensuring Correctness:

- **Pass validation:** IR is checked after each pass.
- **Verifier tool:** opt -verify ensures well-formed IR.
- **Control flags:** e.g., -fno-strict-aliasing, -ffast-math control optimization assumptions.

Formal Verification (in research):

- CompCert: formally verified C compiler.
- Translation validation: Compare input/output IR for equivalence.

Takeaway: A broken optimization is worse than none. Compiler writers must carefully ensure that transformations respect the language semantics and ABI contracts.

EMERGING TOPICS — AI-ASSISTED OPTIMIZATION (1)

- Compiler optimizations are traditionally based on static heuristics and cost models.
- Recently, AI and machine learning are being explored to assist or guide optimizations.

Examples of AI applications:

- Predicting which optimization pass sequences will be most effective.
- Learning heuristics for inlining or loop unrolling.
- Automatically tuning parameters for specific architectures.

EMERGING TOPICS — AI-ASSISTED OPTIMIZATION (2)

Challenges:

- Large search spaces, slow compilation feedback.
- Ensuring correctness and reproducibility.

Potential impact:

- More adaptive, architecture-aware, and performant compilation strategies.

DOES THE ORDER OF OPTIMIZATION PASSES AFFECT THE RESULT? (1)

- Yes—the order of applying optimization passes can significantly affect the final generated code. This is known as the **phase ordering problem**.
- Some optimizations enable others, while others may block or undo potential improvements.
- Compiler designers must carefully schedule passes for maximal benefit.

Key Points:

- **Passes have dependencies**: e.g., mem2reg unlocks DCE.
- **Reordering can lead to missed opportunities**.
- Some transformations are only profitable if applied after others.

DOES THE ORDER OF OPTIMIZATION PASSES AFFECT THE RESULT? (1)

Example in LLVM:

```
; Original
%a = add i32 %x, 0
%y = mul i32 %a, 1
```

If **instruction combining** is done *before* DCE:

- `%a` becomes `%x`
- `%y` becomes `%x`

Then **DCE** removes redundant `%a` and `%y`.

If **DCE** is done first: `%a` and `%y` are still used → nothing eliminated

Takeaway: The **ordering of passes** is crucial. That's why compilers like LLVM group them into well-engineered **pipelines**, often using **re-optimization** to iterate through passes for better results.

PIPELINE BEHAVIOR IN LLVM

- LLVM structures its optimizations as phases where multiple passes are applied and often repeated.
- This design enables **continuous refinement** of the IR until no more substantial improvements can be made.

What Happens:

- **Early passes** simplify expressions and eliminate dead code.
- **Mid-phase passes** like Global Value Numbering (GVN) or Loop-Invariant Code Motion (LICM) act on a cleaner IR.
- **Final simplification passes** polish the IR before code generation.

Benefit:

- Maximizes the impact of optimizations.
- Keeps the compiler efficient and adaptable to different kinds of input programs.

FUTURE DIRECTIONS AND OPEN PROBLEMS

- As hardware becomes more complex, compilers must adapt to produce efficient code.

Open research areas:

- Combining traditional compiler theory with AI-driven decision making.
- Improving compile-time performance while applying deep optimization sequences.
- Better support for heterogeneous architectures (e.g., CPU + GPU).

Final thought:

- Understanding traditional optimizations lays the foundation for exploring and contributing to the next generation of compiler technology.

See you at the next Lecture!

Don’t forget to consult the Q&A blog on Brightspace!