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.

Subsequent sections are dedicated to specific optimization techniques. Section 4 discusses common and foundational optimizations, while Section 5 introduces more advanced transformations that require deeper analysis and interplay between passes. Section 6 addresses reoptimization, explaining how earlier optimizations can create opportunities for additional improvements. Finally, the appendices provide full IR and machine code listings, and a concise guide to relevant LLVM tools and flags.

2. The LLVM Optimization Pipeline

LLVM structures its optimizations as a series of passes. Each pass performs a specific transformation or analysis on the LLVM IR. The IR itself is a typed, SSA-based language that serves as the primary intermediate language throughout the LLVM compilation process.

Optimizations can be applied using:

- The `clang` frontend with optimization flags such as `-O1`, `-O2`, or `-O3`.
- The `opt` tool, which applies individual IR-level passes explicitly.

To understand the purpose and effect of these passes, we begin with a simple function written in C that computes the sum of the elements in an array:

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

At first glance, this function appears simple. However, compiling it without optimizations (`-O0`) leads to inefficient LLVM IR. Every variable is placed in memory using `alloca`, and the loop performs multiple loads and stores per iteration.

Unoptimized LLVM IR (excerpt):

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

This version uses memory extensively for variables like `total` and loop index `i`. Such memory operations are expensive compared to using registers.

By enabling even basic optimizations with `-O1` or applying the `mem2reg` pass manually, LLVM can convert stack variables into SSA values. This change alone eliminates many redundant memory accesses and enables further optimizations such as loop-invariant code motion, dead code elimination, and common subexpression elimination.

Optimized LLVM IR (excerpt):

```
%total = phi i32 [0, %entry], [%add, %loop]
...
%add = add nsw i32 %total, %value
```

The optimized version uses SSA form with `phi` nodes to track values across loop iterations. Variables like `total` are kept in virtual registers instead of memory, making the loop body faster and more compact.

These transformations illustrate the power of LLVM's optimization pipeline. Even a small function benefits significantly from systematic application of IR-level passes. Understanding how this pipeline is structured and how passes interact is key to writing efficient compilers and analyzing optimized code.

In the next section, we use this `sum` function as a running example, examining its IR before and after key optimizations, and observing how these affect the generated machine code.

3. Running Example: From C to RISC-V

To better understand the impact of optimizations, we continue with the `sum` function introduced earlier. This simple function provides a concrete basis for comparing unoptimized and optimized code at both the LLVM IR level and the target machine level—in our case, RISC-V.

We begin by compiling the function using Clang with no optimizations:

```
clang -S -emit-llvm -O0 sum.c -o sum.ll
```

The resulting LLVM IR places all local variables on the stack and includes explicit memory operations for every variable access. The loop is implemented with multiple basic blocks and involves repeated loading and storing of values.

Unoptimized LLVM IR Excerpt:

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

Every read or write to `total` or `i` involves accessing memory, which introduces unnecessary overhead. The loop condition and body are separated into distinct blocks, and values must be loaded from memory repeatedly.

We now apply LLVM's `mem2reg` pass, which promotes stack-based variables to register-based SSA variables:

```
opt -mem2reg < sum.ll > sum_opt.ll
```

Optimized LLVM IR Excerpt:

```
%total = phi i32 [0, %entry], [%add, %loop]
...
%add = add nsw i32 %total, %value
```

The `phi` node represents the loop-carried value of `total`. This IR is more compact and avoids unnecessary memory accesses. The variable `%total` is updated each iteration using virtual registers.

Next, we lower the optimized IR to RISC-V assembly using LLVM's static compiler:

```
llc -march=riscv64 sum_opt.ll -o sum.s
```

RISC-V Assembly (Simplified):

```
  li a2, 0              # total = 0
  li a1, 0              # i = 0
loop:
  bge a1, a0, exit      # if i >= n, exit
  slli a3, a1, 2        # offset = i * 4
  add a4, a3, a5        # address = base + offset
  lw a4, 0(a4)          # load A[i]
  add a2, a2, a4        # total += A[i]
  addi a1, a1, 1        # i++
  j loop
exit:
  mv a0, a2             # return total
```

In the optimized version, the loop performs minimal instructions per iteration. The use of RISC-V registers corresponds directly to the SSA values in the IR. The initial version, in contrast, would have involved extra instructions for loading and storing temporary values in memory.

This comparison illustrates how LLVM's IR-level optimizations translate into more efficient machine code. The `sum`function, although small, captures a typical pattern—looping over data—and benefits from transformations that generalize to more complex programs. In the following sections, we analyze these transformations in detail.

4. Common Optimization Passes

This section explores foundational optimizations performed in the LLVM middle-end. Each pass targets a specific inefficiency and transforms the IR to produce faster or smaller code. We discuss each pass using a structure: what it does, an example with explanation, its impact, and how it is invoked in LLVM.

Constant Folding

Explanation: Constant folding replaces expressions involving only constants with their computed value at compile time. This saves runtime computation and simplifies the IR.

Example:

```
int x = 3 * 4;
```

Initial LLVM IR:

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

After constant folding:

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

Explanation: LLVM recognizes that `3 * 4` is constant and directly stores the result, removing the multiplication instruction.

Conclusion: This improves performance by eliminating redundant instructions and prepares IR for further simplifications.

LLVM instruction: This optimization is typically part of the `-instcombine` pass and happens automatically at `-O1` or higher.

Dead Code Elimination (DCE)

Explanation: DCE removes instructions whose results are never used. Such code often remains after other transformations or redundant assignments.

Example:

```
int x = 5;
x = 6;
```

LLVM IR before DCE:

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

After DCE:

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

Explanation: The value `5` is never read, so the store is removed.

Conclusion: DCE reduces memory traffic and frees opportunities for register reuse and further simplifications.

LLVM instruction: This pass is invoked using `opt -dce` and is included in standard optimization levels.

Common Subexpression Elimination (CSE)

Explanation: CSE eliminates redundant computations of the same expression by reusing the previously computed value.

Example:

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

LLVM IR before CSE:

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

After CSE:

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

Explanation: The repeated computation of `a + b` is identified and reused.

Conclusion: This reduces ALU workload and opens the door for strength reduction and instruction combining.

LLVM instruction: CSE is integrated into `-gvn` (Global Value Numbering).

Loop-Invariant Code Motion (LICM)

Explanation: LICM moves computations that do not change inside a loop (invariants) to a preheader block, so they are executed only once.

Example:

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

Before LICM, the multiplication happens every iteration. After LICM:

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

Explanation: Since `x * 2` does not depend on `i`, it is safely hoisted outside the loop.

Conclusion: This reduces loop body size, improves CPU pipelining, and increases performance.

LLVM instruction: This pass can be applied with `opt -licm`.

Strength Reduction

Explanation: 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 optimization:

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

After strength reduction:

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

Explanation: Multiplying by 2 becomes a shift-left by 1, which is faster on most hardware.

Conclusion: This saves cycles and improves loop performance, especially in inner loops.

LLVM instruction: This is included in `-instcombine` and is automatically applied in `-O2` and `-O3`.

Register Promotion (mem2reg)

Explanation: Variables stored in memory (via `alloca`) are promoted to SSA registers. This simplifies IR and enables more aggressive optimizations.

Example:

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

After `mem2reg`:

```
%a = 5
```

Explanation: The load and store are replaced by a direct SSA assignment.

Conclusion: This transformation reduces memory traffic and forms the basis for applying further passes like CSE and DCE.

LLVM instruction: Applied using `opt -mem2reg`, it is one of the first passes applied in optimized pipelines.

5. Advanced Optimization Techniques

Beyond the fundamental transformations described earlier, LLVM supports a range of advanced optimizations that further improve performance, particularly in compute-intensive or recursive code. Each technique leverages more complex analyses or architectural knowledge. Below, we explain each one with examples and practical LLVM usage.

Loop Unrolling and Vectorization

Explanation: Loop unrolling duplicates the loop body multiple times to reduce the overhead of branch and condition checks. Vectorization transforms scalar operations into SIMD instructions, processing multiple data elements in parallel.

Example:

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

After unrolling:

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

Explanation: This eliminates loop control logic, reducing branch mispredictions and increasing instruction-level parallelism. If the target architecture supports SIMD, LLVM can generate vector instructions instead of scalar additions.

Conclusion: Unrolling and vectorization significantly boost performance in data-parallel code, especially in scientific or multimedia applications.

LLVM instruction: Apply with `opt -loop-unroll` and `opt -loop-vectorize`; these are also enabled by default at `-O2`or `-O3`.

Tail Call Elimination

Explanation: When a function call is the final action in another function, it can be transformed into a jump, avoiding the overhead of a new stack frame. This is called tail call optimization.

Example:

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

Explanation: This tail-recursive function doesn't need to keep previous frames. LLVM can optimize the call to reuse the current frame.

Conclusion: Tail call elimination improves recursion-heavy code by preventing stack overflow and reducing function call overhead.

LLVM instruction: Enabled automatically when possible; can be confirmed with IR that shows `musttail` or `tail`annotations.

Function Inlining

Explanation: Inlining replaces a function call with the function body. This saves the cost of the call and can expose further optimization opportunities within the caller.

Example:

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

After inlining:

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

Explanation: Inlining removes the function call overhead and enables constant folding in the caller.

Conclusion: Inlining benefits small, frequently called functions and is a prerequisite for several downstream optimizations.

LLVM instruction: Use `opt -inline` or rely on automatic inlining at `-O2`/`-O3` levels.

Instruction Combining (InstCombine)

Explanation: This pass simplifies instruction sequences by applying local rewrite rules. It is a form of peephole optimization over the IR.

Example:

```
%1 = add i32 %a, 0
```

After InstCombine:

```
%1 = %a
```

Explanation: Adding zero is a no-op, so the addition is removed.

Conclusion: Instruction combining simplifies the IR and prepares it for more advanced, global optimizations.

LLVM instruction: Applied using `opt -instcombine`; this pass is heavily used at all optimization levels.

Global Value Numbering (GVN)

Explanation: GVN eliminates redundant expressions by assigning unique numbers to expressions with the same computed value across basic blocks.

Example:

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

LLVM IR before GVN:

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

After GVN:

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

Explanation: GVN detects that `%1` and `%2` compute the same value and reuses the result.

Conclusion: This reduces instruction count and increases opportunities for register reuse and simplification.

LLVM instruction: Apply with `opt -gvn`; this is standard in optimization pipelines.

Control Flow Graph (CFG) Simplification

Explanation: CFG simplification cleans up the control flow by removing unreachable blocks, merging trivial basic blocks, and eliminating redundant branches.

Example:

```
if (1) {
    doSomething();
}
```

Explanation: Since the condition is statically true, the branch can be removed and the block inlined.

Before CFG Simplification: Two separate blocks, conditional jump.

After CFG Simplification: Straight-line code with no condition.

Conclusion: CFG simplification reduces control overhead, shortens paths through the code, and makes later analysis more effective.

LLVM instruction: Use `opt -simplifycfg`; this is often used alongside other cleanup passes.

6. Reoptimization: Enabling Further Improvements

Compiler optimizations often interact in complex ways. Applying a single transformation can expose new opportunities for further improvement that were not initially visible. Reoptimization is the process of reapplying optimization passes—or running new ones—after the IR has been transformed by earlier stages. LLVM's design makes reoptimization a core strategy by structuring its pipelines to allow multiple phases of transformation and analysis.

Explanation: Reoptimization addresses the fact that many optimizations are interdependent. For example, eliminating memory accesses through register promotion (`mem2reg`) may uncover instructions whose results are never used, which can then be eliminated by dead code elimination. Similarly, simplifying arithmetic expressions might expose new opportunities for instruction combining or loop unrolling.

Example 1: Register Promotion and DCE

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

- Initially, all variables are in memory.
- After `mem2reg`, they become SSA variables.
- LLVM then identifies `unused` as dead, and `opt -dce` removes the corresponding code.

Example 2: Simplification and CSE

```
int f(int a, int b) {
    int x = a + b;
    int y = (a + b) * 2;
    return x + y;
}
```

- Instruction combining may fold `(a + b)` into a single value.
- Common subexpression elimination can reuse that result for both `x` and the multiplication.

- GVN may then assign a shared value number to the addition.

Explanation: These sequences demonstrate how one pass enables the applicability of others. If the pipeline ran each pass only once, many of these secondary improvements would be missed.

Conclusion: Reoptimization ensures that transformation phases are not isolated. LLVM's pass pipelines are carefully ordered so that early simplifications lead to subsequent, deeper optimizations. Compiler developers must be aware of this interplay to design effective optimization pipelines.

LLVM instruction: Reoptimization is built into composite optimization levels such as `-O2` and `-O3`, which reapply passes like `-instcombine`, `-dce`, `-simplifycfg`, and `-gvn` in phases. Custom pipelines can be constructed using `opt` with repeated or ordered passes to achieve similar results.

Conclusion

Compiler optimization is a foundational aspect of modern software development, bridging the gap between human-readable programs and efficient machine-executable code. In this report, we explored the LLVM compiler infrastructure, emphasizing how its modular and extensible design enables a wide range of optimizations.

We began by motivating the need for optimization and explaining the structure of LLVM's pipeline. Through a concrete running example, we demonstrated how even simple C code can benefit from several layers of transformation. Sections 4 and 5 provided a deep dive into both common and advanced optimization techniques, illustrating how they operate, their effects on performance, and how LLVM applies them using specific passes.

Importantly, we highlighted the concept of reoptimization, showing how applying one transformation can reveal opportunities for others. LLVM's pass pipeline is designed to exploit these opportunities through careful pass scheduling and multiple optimization phases.

Understanding these optimizations empowers developers and compiler engineers alike. Developers gain insight into how their code might be transformed and optimized, which can inform better coding practices. Compiler engineers can appreciate the design trade-offs in building effective pass sequences and targeting specific hardware architectures like RISC-V.

Ultimately, the techniques discussed here form the basis for advanced topics such as profile-guided optimization, link-time optimization, and JIT compilation—all of which build upon the principles established in the LLVM backend.

References

- Appel, A. W., & Palsberg, J. (2002). *Modern Compiler Implementation in C*. Cambridge University Press.
- Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
- Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.
- LLVM Project. (n.d.). https://llvm.org

Appendices

Appendix A: Full IR Before and After Optimization

This appendix contains the full LLVM IR of the `sum` function before and after applying optimizations. The IR was generated using Clang with the following command:

```
clang -S -emit-llvm -O0 sum.c -o sum.ll
```

This unoptimized version includes extensive use of `alloca`, `store`, and `load`, representing variables in memory:

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

After applying `mem2reg` and enabling SSA construction with:

```
opt -mem2reg < sum.ll > sum_opt.ll
```

the IR is transformed into a more compact and register-efficient form:

```
%total = phi i32 [0, %entry], [%add, %loop]
...
%add = add nsw i32 %total, %value
```

This allows LLVM to eliminate many memory accesses and paves the way for loop optimizations and instruction simplification.

Appendix B: RISC-V Code for Unoptimized and Optimized Versions

Using the LLVM static compiler `llc`, we convert the IR to RISC-V assembly:

```
llc -march=riscv64 sum.ll -o sum_unopt.s
llc -march=riscv64 sum_opt.ll -o sum_opt.s
```

The unoptimized RISC-V code includes excessive memory instructions and indirect address calculations:

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

In contrast, the optimized version avoids unnecessary loads and stores:

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

This illustrates the dramatic effect LLVM optimizations have when targeting a specific architecture like RISC-V.

Appendix C: Summary of LLVM Tools and Optimization Flags

This appendix provides an overview of LLVM tools and command-line options relevant for exploring and applying optimizations.

`clang`

LLVM's C/C++ frontend, used to generate IR or compile directly to machine code.

- `-O0`, `-O1`, `-O2`, `-O3`: Specify the level of optimization.
- `-S`: Output assembly.
- `-emit-llvm`: Emit LLVM IR in human-readable (`.ll`) or bitcode (`.bc`) format.
- Example:

```
clang -S -emit-llvm -O2 sum.c -o sum.ll
```

`opt`

Applies specific LLVM passes to IR.

- Common flags:
    - `-mem2reg`: Promote stack allocations to registers.
    - `-instcombine`: Simplify instruction sequences.
    - `-dce`: Dead code elimination.
    - `-gvn`: Global value numbering.
    - `-licm`: Loop-invariant code motion.
    - `-loop-unroll`: Loop unrolling.
    - `-loop-vectorize`: Vectorization.
    - `-simplifycfg`: Control flow graph simplification.
- Example:

```
opt -mem2reg -instcombine -dce -S input.ll -o output.ll
```

`llc`

Translates LLVM IR to target-specific assembly.

- `-march=riscv64`: Target RISC-V 64-bit architecture.
- Example:

```
llc -march=riscv64 input.ll -o output.s
```

`llvm-dis` / `llvm-as`

- `llvm-dis`: Convert LLVM bitcode (`.bc`) to text IR (`.ll`).
- `llvm-as`: Convert text IR (`.ll`) to LLVM bitcode (`.bc`).

These tools are invaluable for inspecting, modifying, and testing optimization behavior in a controlled manner.