Advanced Compiler Final Report

r12922189 Ren-Hua Yang
Department of Computer Science, National Taiwan University
Email: r12922189@csie.ntu.edu.tw

I. Algorithm Design

Peephole optimization is a compiler technique that examines and simplifies small portions of the intermediate representation (IR) to improve code efficiency. Unlike global optimization techniques, peephole optimization operates locally on small sequences of instructions, typically within a single basic block.

The goals of peephole optimization are:

- To eliminate redundant or trivial computations.
- To simplify operations, such as replacing multiplications by powers of two with bitwise shifts.
- To improve runtime performance and reduce binary size.

For example, a common transformation is replacing ‘add x, 0‘ with ‘x‘, as the addition operation does not change the value of ‘x‘. Similarly, ‘mul x, 1‘ can be simplified to ‘x‘, and ‘xor x, x‘ can be replaced with ‘0‘.

A. Algorithm Description

The proposed algorithm processes the IR of a program and identifies opportunities for peephole optimizations. The steps are as follows:

1) Module and Function Traversal: The algorithm iterates through the program’s module, which contains all the functions. Functions that are only declared (e.g., external library calls) are skipped.

2) Basic Block and Instruction Iteration: For each function, the algorithm processes its basic blocks. A basic block is a linear sequence of instructions with a single entry and exit point. The algorithm examines each instruction sequentially within the block.

3) Pattern Matching: The algorithm identifies patterns that can be simplified. Some common patterns include:

- **Addition with Zero:** Replace ‘add x, 0‘ with ‘x‘.
- **Subtraction by Zero:** Replace ‘sub x, 0‘ with ‘x‘.
- **Multiplication by One or Two:**
  - Replace ‘mul x, 1‘ with ‘x‘.
  - Replace ‘mul x, 2‘ with ‘shl x, 1‘.
- **Division by One:** Replace ‘div x, 1‘ with ‘x‘.
- **Logical Operations:**
  - Replace ‘xor x, x‘ with ‘0‘.
  - Replace ‘and x, 0‘ with ‘0‘.
  - Replace ‘or x, 0‘ with ‘x‘.

4) Replacement and Cleanup: When a pattern is identified, the instruction is replaced with its simplified equivalent. The algorithm updates all references to the original instruction to point to the new value. The obsolete instruction is then removed from the IR to maintain cleanliness.

5) Iterative Application: The algorithm is applied iteratively since one optimization may expose additional opportunities. For example, simplifying a computation might reveal a redundant operation in subsequent instructions.

B. Example

Consider the following LLVM IR before optimization:

```
%1 = add i32 %x, 0
%2 = mul i32 %y, 1
%3 = mul i32 %z, 2
%4 = xor i32 %w, %w
```

After applying peephole optimization:

```
%1add i32 %x, 0 is replaced with %x.
%2 mul i32 y, 1 is replaced with %y.
%3 mul i32 %z, 2 is replaced with shl i32 %z, 1.
%4 xor i32 %w, %w is replaced with 0.
```

The resulting optimized IR becomes:

```
%1 = %x
%2 = %y
%3 = shl i32 %z, 1
%4 = 0
```

This reduces the number of instructions and simplifies computations.

II. Implementation Details

The implementation of the peephole optimization pass leverages the LLVM Pass Manager framework. This section outlines the key steps in implementing the pass, including module traversal, instruction analysis, and transformation.

A. LLVM Pass Manager

The LLVM Pass Manager provides a modular and extensible framework for writing optimization passes. The pass is implemented as a `ModulePass`, which operates on the LLVM IR at the module level. The pass is registered using the `PassPluginLibraryInfo` structure, enabling integration into the LLVM compilation pipeline.

B. Algorithm Workflow

The pass follows a systematic workflow to identify and simplify patterns in the LLVM IR:

1) **Module Traversal:** The pass iterates over each function in the module. Functions without definitions (e.g., external calls) are skipped.
2) **Basic Block and Instruction Iteration:** For each function, the pass iterates through its basic blocks and examines each instruction sequentially.
3) **Pattern Matching:** The pass checks for predefined patterns, such as `add x, 0`, `mul x, 1`, and `xor x, x`. These patterns are identified using LLVM's `BinaryOperator` class.
4) **Replacement and Cleanup:** When a pattern is matched, the pass replaces the instruction with its simplified equivalent. For example, `add x, 0` is replaced with `x`. The obsolete instruction is then removed from the IR using the `eraseFromParent()` method.

C. Code Example: Addition Simplification

The following code snippet demonstrates how the pass simplifies the `add x, 0` pattern:

```
if (Opcode == Instruction::Add) {
  if (*CInt = dyn_cast(Op1)) {
      if (CInt->isZero()) {
          BinOp->replaceAllUsesWith(Op0);
          BinOp->eraseFromParent();
      }
  }
}
```

In this snippet, the instruction is checked to see if it is an addition operation where the second operand is zero. If matched, the redundant addition is replaced by the first operand, and the original instruction is deleted.

D. Iterative Application

The pass operates iteratively to ensure that all possible optimizations are applied. For instance, simplifying one instruction might expose further opportunities for optimization in subsequent passes.

Algorithm 1 Example Test Cases for Peephole Optimization

Input: Integer variable x
Output: Optimized result for specific operations
function TEST_ADD0(x)
    return $x + 0$
end function
function TEST_MUL1(x)
    return $x \times 1$
end function
function TEST_SUB0(x)
    return $x - 0$
end function
function TEST_MUL2(x)
    return $x \times 2$
end function
function TEST_DIV1(x)
    if $x \neq 0$ **then** ▷ Handle division by zero edge case
        return $x \div 1$
    else
        return 0
    end if
end function
function TEST_XOR0(x)
    return $x \oplus 0$
end function
function TEST_XORX(x)
    return $x \oplus x$
end function
function TEST_AND0(x)
    return $x \wedge 0$
end function
function TEST_ANDX(x)
    return $x \wedge x$
end function
function TEST_OR0(x)
    return $x \vee 0$
end function
function TEST_ORX(x)
    return $x \vee x$
end function
function TEST_SUBX(x)
    return $x - x$
end function
function MAIN
    $a \leftarrow 42$
    Print(TEST_ADD0(a))
    Print(TEST_MUL1(a))
    Print(TEST_SUB0(a))
    Print(TEST_MUL2(a))
    Print(TEST_DIV1(a))
    Print(TEST_XOR0(a))
    Print(TEST_XORX(a))
    Print(TEST_AND0(a))
    Print(TEST_ANDX(a))
    Print(TEST_OR0(a))
    Print(TEST_ORX(a))
    Print(TEST_SUBX(a))
end function

III. Experimental Evaluation & Correctness Proof

The correctness of the proposed peephole optimization pass was verified using both experimental results and informal reasoning. Specifically, the provided `test.c` file contains a comprehensive set of test cases that verify the pass's ability to correctly detect and optimize the patterns described earlier. Each test case corresponds to a specific pattern and evaluates whether the transformation results in the expected optimized output.

A. Verification Through Test Cases

By executing the `test.c` file, we confirmed that the pass accurately detects and handles all specified patterns. Each pattern was tested individually, and the resulting output matched the expected behavior:

- Redundant instructions, such as `add x, 0`, were correctly simplified to `x`.
- Logical identities, such as `xor x, x`, were replaced with constant values (e.g., `0`).
- Multiplications involving constants, such as `mul x, 2`, were replaced with equivalent but more efficient bitwise shifts (e.g., `shl x, 1`).

This experimental validation ensures that the pass performs as expected for all tested patterns, and no errors were observed during execution.

B. Informal Proof of Correctness

The correctness of the peephole optimization pass can also be reasoned informally. Each pattern transformation involves either the elimination of redundant operations or a direct substitution of equivalent instructions. These transformations are simple and straightforward, preserving the semantics of the original program:

- **Elimination:** Instructions such as `add x, 0` and mul x, 1 are redundant and can be safely removed without altering the program's output.
- Substitution: Transformations like `mul x, 2` to `shl x, 1` are semantically equivalent and introduce no behavioral changes.

Since the optimization pass iteratively examines each instruction in isolation, the correctness of each transformation is maintained throughout the process. No global invariants are violated, and data dependencies remain intact.

C. Conclusion

Given the simplicity of the transformations and the iterative inspection of each instruction, the correctness of the proposed peephole optimization pass can be assured. While the reasoning presented here constitutes an informal proof, the combination of theoretical soundness and experimental validation provides strong evidence of the pass's reliability.