Peephole Optimization in Compiler Design

Meng-Heng Tsai
Department of Computer Science & Information Engineering
Nation Taiwan University
Taipei, Taiwan
b12902022@ntu.edu.tw

***Abstract*—This document is a technical report for my implementation of Peephole Optimization in Compiler Design. The report includes the introduction of Peephole Optimization, my algorithm design, the implementation of Peep Hole Optimization, the result of Peep Hole Optimization, and the conclusion of the report.**

***Index Terms*—peephole optimization, compiler design, optimization**

I. Introduction

Peephole optimization is a local code optimization technique that focuses on improving small sections of code, typically within a short "window" (or peephole) of instructions. The goal is to identify and replace suboptimal instruction sequences with more efficient alternatives. It operates on intermediate code or assembly language (in this report we use llvm) and is a simple yet powerful way to improve performance and reduce code size.

Pattern matching is a core technique used in classic peephole optimization to identify specific sequences of instructions or code patterns that can be simplified or replaced with more efficient equivalents. It operates by scanning the code (usually in an intermediate representation) and applying predefined transformation rules when matching patterns are found.

II. Algorithm Design & Correctness

A. Logic of the Algorithm

I've implemented the classic peephole optimization algorithm. The algorithm works by scanning the code and applying predefined transformation rules when matching patterns are found. (For classic peephole optimization, the transformation rules are defined manually, and hence easy to prove the correctness of the algorithm.)

As we are mainly focusing on implmenting the peephole optimization, I use the **sroa** pass implemented in section VI (as a bonus implementation) before the peephole optimization. The **sroa** pass is a simple pass that splits aggregates (**alloca**, **getelementptr**, **load**, and **store** operations on structs or arrays) into scalar values. This pass is useful because it exposes more opportunities for peephole optimization. Also, without this pass, the peephole optimization may fail to converge as there may be some ill-formed **alloca** instructions that causes indefinite loop or unexpected complexity in the peephole optimization. (It would also fail in the original implementation of the peephole optimization (mainly **instcombine**) in llvm.) The following is the pseudo code of the peephole optimization.

Algorithm 1 Peephole Optimization

```
function BuildOptimizationPatterns( )
    Identify common instruction patterns
    Calculate cost for each pattern
    Generate optimal replacements for patterns
    return pattern dictionary
end function
function OptimizeProgram(program)
    patterns ← BuildOptimizationPatterns()
    for each window in program do
        if window matches a known pattern then
            Replace with more efficient equivalent
        end if
    end for
    Remove unreachable code
    Eliminate redundant instructions
    return optimized program
end function
```

B. Pattern Matching Rules

And here is the pattern matching rules that I've implemented in the peephole optimization:

1) **Multiply by power of 2 → Shift left**

$$x \times 2^n \to x \ll n$$

2) **Division by power of 2 → Shift right**

$$x \div 2^n \to x \gg n$$

3) **Add zero elimination**

$$x + 0 \to x$$

4) **Multiply by zero → zero**

$$x \times 0 \to 0$$

5) **XOR with self → zero**

$$x \oplus x \to 0$$

6) **AND with self → self**

$$x \wedge x \to x$$

7) **OR with self → self**

$$x \vee x \to x$$

8) **NOT NOT → original**

$$\neg(\neg x) \rightarrow x$$

9) **AND with all ones → self**

$$x \wedge \text{ all\_ones } \rightarrow x$$

10) **OR with zero → self**

$$x \vee 0 \rightarrow x$$

11) **Constant Propagation**
If both operands are constants, directly evaluate the operation at compile time.

12) **Subtract zero elimination**

$$x - 0 \rightarrow x$$

13) **Negate zero**

$$-0 \rightarrow 0$$

14) **Multiply by one**

$$x \times 1 \rightarrow x$$

15) **Divide by one**

$$x \div 1 \rightarrow x$$

The correctness of the algorithm can be proved by the correctness of the pattern matching rules. The pattern matching rules are simple and straightforward, and can be easily verified by manual inspection. The algorithm is also deterministic, as it applies the same transformation rules to the same patterns every time.

And here's an simple example in **./tests/easy.c**:

```
int x = 1;
int y = 2;
int z = x + y;
printf("Sum of x and y is: %d\n", z);
int d = 234214213;
int e = 2134123;
```

Fig. 1. **easy.c** before peephole optimization (headers and main function are omitted)

And the result after the peephole optimization is:

```
printf("Sum of x and y is: %d\n", 3);
```

Fig. 2. **easy.c** after peephole optimization (headers and main function are omitted)

After the peephole optimization, the code is simplified by constant propagations and redundant code elimination, so the code size is reduced. Note that the whole process of the peephole optimization should be done within LLVM IR. However, the example is simplified to show the peephole optimization in C code.

III. Implementation Details

A. LLVM Pass Implementation Methodology

The PeepHole optimization pass is implemented as an LLVM pass that performs peephole optimizations on LLVM IR(Intermediate Representation) .The pass identifies and replaces specific patterns in the IR with more efficient equivalents .The pass is structured as a class **PeepHolePass** that inherits from **PassInfoMixin<PeepHolePass>** .The main functionality is encapsulated in the **run** method, which iterates over the instructions in a function and applies the defined optimization patterns.

B. Key Data Structures and Algorithms Used

- **Pattern Structure** :
  - The **Pattern** structure is used to define individual optimization patterns.Each pattern consists of :
    - A **matcher** function that identifies if an instruction matches the pattern .
    - A **replacement** function that generates the optimized instruction .
    - A **costDelta** indicating the cost difference between the original and optimized instructions.
- **Patterns Vector** :
  - A vector of **Pattern** structures is used to store all the optimization patterns .This vector is initialized in the **initializePatterns** method.
- **Dead Code Elimination(DCE)** :
  - The **performDCE** method is used to remove dead code after applying the optimizations.It identifies instructions that have no uses, are not terminators, and do not have side effects, and then removes them from the function.

C. Integration with the LLVM Optimization Pipeline

The pass is integrated into the LLVM optimization pipeline using the **PassPluginLibraryInfo** structure .The **llvmGetPassPluginInfo** function registers the pass with the LLVM pass manager. The pass can be invoked using the name "peephole" in the LLVM pass pipeline.

D. Handling of Edge Cases and Special Conditions

I've implemented a class **TransformationVerifier** to verify the correctness of the transformation rules. The class will ensure the transformation preserve the semantics of the code. The class will check the correctness of the transformation by using **ScalorEvolution** analysis in LLVM.

To be specific, The **TransformationVerifier** class ensures that instruction transformations preserve the program's correctness. It performs various checks to validate key properties of the original and transformed instructions, including type compatibility, control flow preservation, memory access patterns, data dependencies, arithmetic equivalence, and exception behavior. The class leverages LLVM's analysis infrastructure to verify the correctness of the transformation

rules, I implemented the following checks as methods of the **TransformationVerifier** class:

1) Type Compatibility: Verifies that the original and transformed values have matching types.
2) Control Flow Preservation: Ensures terminator status and side-effect behaviors remain consistent.
3) Memory Access Patterns: Uses MemorySSA to confirm alignment in memory read/write operations and checks the preservation of memory dependencies.
4) Data Dependencies: Tracks read-after-write (RAW) and write-after-write (WAW) dependencies to avoid introducing circular dependencies or violating dependency constraints.
5) Arithmetic Equivalence: Employs **ScalarEvolution** to validate mathematical equivalence for arithmetic instructions.
6) Exception Behavior: Confirms consistent exception-throwing properties.

The verify method integrates these checks by leveraging analyses such as MemorySSAAnalysis and ScalarEvolutionAnalysis from the FunctionAnalysisManager. This modular approach ensures semantic equivalence between the original and transformed instructions while maintaining program correctness.

IV. Experimental Evaluation

A. Test Cases and Benchmarks

I've tested the peephole optimization on a few test cases to evaluate its effectiveness. The test cases include simple arithmetic operations, for loops, and extreme case that match all the patterns in my optimization pass. The goal is to demonstrate the reduction in code size and improvement in performance achieved by the peephole optimization, as well as to verify the correctness of the transformation rules. The test cases are written in C and compiled to LLVM IR using clang with -O0 optimization level.

1) **hello.c**: A simple test case with only a printf statement. (The Peephole optimization should not change the code.)
2) **easy.c**: Test cases with really basic arithmetic operations and a printf statement for illustration.
3) **dead_code.c**: Test cases with dead code that should be elminated by the peephole optimization.
4) **arithmetic.c**: Test cases with more complex arithmetic operations to test the correctness of the peephole optimization.
5) **extreme.c**: A test case that includes all the pattern matching rules to evaluate the effectiveness of the peephole optimization.
6) **random.c**: A test case with 1000 random arithmetic operations (generated by **random_gen.py**) to evaluate the reduction in code size and improvement in performance.
7) **array.c**: A test case doing fibonacci sequence calculation with array. (The Peephole optimization shouldn't change the code a lot.)

The test cases are designed to cover a range of scenarios and patterns that can be optimized by the peephole optimization pass. The benchmarks are run on the CSIE workstation ws1. I run the test cases with and without the peephole optimization pass enabled, and another with O1 flag to compare the code size and performance improvements. As the execution time and memory usage may vary on different machines, the results are mainly used for relative comparison.

To measure the performance, I write a simple shell script to compile the target file with clang and emit the LLVM IR, then use **opt** command to run the pass. Finally, I compile the optimized LLVM file back to the executable file and run it to measure the performance. The performance is measured by the time taken to execute the program, the memory usage, and also the IR code size (**.text** part evaluated by **llvm-size** command). Given the nature of the execution time and memory usage, the results may be noisy, so I run each test case for 20 times and take the average.

B. Performance Measurements and Analysis

1) Result Matrix: Please refer to Table I for the performance metrics for each test case.

2) Analysis & Comparison: The performance metrics for each test case, as shown in Table I, provide a comprehensive comparison between the unoptimized code and the code optimized with peephole techniques and uses the default O1 as reference. The key metrics analyzed include IR code size, execution time, and memory usage.

3) Improvement Significance: I use t-test to test the significance of the improvement in the performance metrics. For the case in **tests/extreme.c**, the statistics are shown in Table II

And I get the p-value that much smaller than 0.0001, which indicates that the improvement is significant. With 95% confidence, the execution time difference will be in the range of 2.6739 to 2.8301. But not all the test cases have significant improvement, as the peephole optimization is not always effective in all cases.

- **IR Code Size:** The IR code size is significantly reduced in the optimized code across all test cases. For example, in the 'tests/arithmetic.c' test case, the IR code size decreased from 2716 bytes to 2538 bytes with the pass and further to 2430 bytes with the -O1 optimization. This reduction is consistent across other test cases, indicating that the peephole optimization effectively eliminates redundant instructions and optimizes instruction sequences.
- **Execution Time:** The execution time shows a notable improvement in the optimized code. For instance, the 'tests/extreme.c' test case's average runtime decreased from 4.984 seconds to 2.232 seconds with the pass and further to 0.464 seconds with the -O1 optimization. Similar improvements are observed in other test cases, demonstrating that the optimized code executes more efficiently, reducing the overall runtime.
- **Memory Usage:** The memory usage remains relatively unchanged across most test cases, with minor variations. For example, in the 'tests/array.c' test case, the average

TABLE I
PERFORMANCE METRICS FOR TEST CASES

Test File	Metric	Without Pass	With Pass	With -O1	Pass Time
tests/arithmetic.c	Average Runtime (s)	11.091	9.749	6.767	42 ms
	Average Memory Usage (KB)	1152.000	1152.000	1152.000	—
	IR Code Size (bytes)	2716	2538	2430	—
tests/array.c	Average Runtime (s)	0.277	0.391	0.267	31 ms
	Average Memory Usage (KB)	195924.400	196081.200	195993.200	—
	IR Code Size (bytes)	1553	1535	1524	—
tests/dead_code.c	Average Runtime (s)	0	0	0	33 ms
	Average Memory Usage (KB)	1152.000	1152.000	1152.000	—
	IR Code Size (bytes)	1320	1274	1258	—
tests/easy.c	Average Runtime (s)	0	0	0	33 ms
	Average Memory Usage (KB)	1152.000	1152.000	1152.000	—
	IR Code Size (bytes)	1330	1290	1276	—
tests/extreme.c	Average Runtime (s)	4.984	2.232	0.464	42 ms
	Average Memory Usage (KB)	1152.000	1152.000	1152.000	—
	IR Code Size (bytes)	1596	1528	1481	—
tests/hello.c	Average Runtime (s)	0	0	0	33 ms
	Average Memory Usage (KB)	1152.000	1152.000	1152.000	—
	IR Code Size (bytes)	1289	1274	1258	—
tests/random.c	Average Runtime (s)	0	0	0	80 ms
	Average Memory Usage (KB)	1152.000	1152.000	1152.000	—
	IR Code Size (bytes)	23721	2195	2188	—

TABLE II
SUMMARY STATISTICS FOR WITHOUT PASS GROUP AND WITH PASS GROUP IN **TESTS/EXTREME.C**.

Group	Mean	SD	SEM	N
Without Pass	4.984	0.1325	0.0296	20
With Pass	2.232	0.1105	0.0247	20

memory usage slightly increased from 195924.400 KB to 196081.200 KB with the pass but remained stable with the -O1 optimization. This indicates that while the peephole optimization primarily focuses on reducing code size and execution time, it does not significantly impact memory usage.

- **Pass Time:** The pass time is the time taken to run the peephole optimization pass on the test case. The pass time is relatively low, indicating that the optimization process is efficient and does not introduce significant overhead.

Overall, the analysis of the performance metrics shows that the peephole optimization techniques applied result in substantial improvements in IR code size and execution time, with minimal impact on memory usage. These findings validate the effectiveness of the optimization strategies employed.

V. FUTURE WORK

I found that the peephole optimization is quite effective in reducing the code size and improving the performance of the code. However, the peephole optimization is not always effective in all cases. When I was finding the pattern matching rules, I found that there are many cases that are not covered by my implementation. And this should be done by a automatic way to find the pattern matching rules without manual effort (As the manual effort is quite time-consuming and error-prone).

According to the paper [1] and [2], the modern automatic peephole optimization can be done by iterative way, which tries to find the best optimization by computing the cost of the code and the optimization, and check the correctness using a SMT solver. For example, the peephole optimization can be done by using the Z3 SMT solver or Alive2 to check the correctness of the optimization.

VI. BONUS IMPLEMENTATION: SROA PASS

As mentioned in section II, I implemented a simple pass that splits aggregates (**alloca**, **getelementptr**, **load**, and **store** operations on structs or arrays) into scalar values, removing the need of stack memory allocations. This pass is useful because it exposes more opportunities for peephole optimization and avoids undefined behavior as peephole might not be able to address with complex structures.

A. Algorithm Design & Correctness

The SROA Pass identifies memory allocations that can be simplified, promotes them to scalar variables in registers, and repeats this process until no further optimizations are possible. This transformation improves program performance by reducing memory access overhead and leveraging faster CPU registers. Below is the pseudo code of the SROA pass (high level overview):

Algorithm 2 SROA Pass

function RUN(Function F, AnalysisManager AM)
    Retrieve DominatorTree (DT) if needed
    if not TRANSFORM(F, DT) **then**
        return Preserve All Analyses
    end if
    return Preserve CFG Analyses
end function
function TRANSFORM(Function F, DominatorTree DT)
    $Changed \leftarrow$ PROMOTEALLOCATIONS(F, DT)
    return $Changed$
end function
function PROMOTEALLOCATIONS(Function F, DominatorTree DT)
    repeat
        $Allocas \leftarrow$ Find promotable memory allocations in F
        if $Allocas$ is empty **then**
            break
        end if
        Promote $Allocas$ to registers
    until $Allocas$ is empty
    return true if changes occurred
end function

The SROA pass is designed to optimize memory allocations by promoting them to registers. For example, consider the following C code snippet:

```
struct Point {
    int x, y;
} p;
p.x = 42;
p.y = 13;

int sum = p.x + p.y;
```

It can be optimized to the following equivalent code:

```
int x = 42;
int y = 13;

int sum = x + y;
```

This approach reduces memory access overhead and improves performance by promoting memory allocations to registers. To prove that SROA pass does not change semantics, the key observation is taht SROA pass will preserve the memory locations and the values stored in them.

After SROA, the individual components of the aggregate are stored in separate scalar variables (or registers). These variables are still manipulated in the same way as the original struct fields, and no new memory locations are introduced that would alter the program's state. In fact, the memory locations (i.e., stack or heap addresses) of the individual fields of the struct are replaced with scalar variables. Since the SROA pass only replaces memory access to the struct fields with register or direct variable usage, there is no change to the program's overall memory semantics.

B. Implementation Details

1) Methodology: The SROA(Scalar Replacement of Aggregates) pass is designed to promote memory allocations to registers, thereby optimizing the code. The algorithm is implemented in the **SROA** class, which is a subclass of **PassInfoMixin<SROA>**. Below is a detailed explanation of the algorithm:

2) Key Data Structures and Algorithms Used:

- **Class Definition:**
  - The **SROA** class has a boolean member **RequiresDomTree** to indicate if a Dominator Tree is needed.
- **Constructor:**
  - Initializes **RequiresDomTree** with a default value of **true**.
- **Run Method:**
  - Entry point of the pass.
  - Retrieves the Dominator Tree if required.
  - Calls **runOnFunction** for the transformation.
- **runOnFunction Method:**
  - Performs the main transformation logic.
  - Calls **promoteAllocas** to promote eligible allocas to registers.
- **promoteAllocas Method:**
  - Identifies and promotes allocas that are safe to promote.
  - Iterates over instructions in the entry block of the function.
  - Collects promotable allocas into a vector.
  - Promotes these allocas using **PromoteMemToReg**.
- **isAllocaPromotable Method:**
  - Checks if an alloca is safe to promote.
  - Ensures the allocated type is sized and the alloca is static.
  - Checks for instructions that make promotion unsafe, such as volatile loads/stores and complex GEPs.

3) Integration with the LLVM Optimization Pipeline: The SROA pass integrates with the LLVM optimization pipeline during the optimization phase, working with other passes to promote memory allocations to registers and improve performance. This pass is applied before the peephole optimization as mentioned in II to ensure that memory allocations are efficiently promoted to registers, setting the stage for further optimizations.

4) Handling of Edge Cases and Special Conditions: The SROA pass handles edge cases and special conditions by:

- Ensuring only static and sized allocas are considered for promotion.
- Checking for instructions that could make promotion unsafe, such as volatile loads/stores and complex GEPs.
- Preserving CFG analyses to maintain control flow graph integrity after transformations.

REFERENCES

[1] Davidson, Jack & Fraser, Christopher. (1984). Automatic generation of peephole optimizations. Sigplan Notices - SIGPLAN. 39. 111-116. 10.1145/989393.989407.

[2] Bhatt, Chirag & Bhadka, Harshad. (2013). Peephole Optimization Technique for analysis and review of Compiler Design and Construction. IOSR Journal of Computer Engineering (IOSR-JCE) 2278-0661 (impact Factor - 1.69). 9. 80-86. 10.9790/0661-0948086.