Peephole Optimization

Simon Oehrl and Bao-Loc Nguyen Ngo

Code Optimization

- Improve code quality and efficiency
  - Increase Performance
  - Reduce code size
- Optimization on different abstraction levels
  - High-level programming language
  - Intermediate code
  - Machine code
- Optimization techniques
  - Local optimization
  - Global optimization
  - Peephole optimization
  - ...
Code Optimization

- Improve code quality and efficiency
  - Increase Performance
  - Reduce code size

- Optimization on different abstraction levels
  - High-level programming language
  - Intermediate code
  - Machine code

- Optimization techniques
  - Local optimization
  - Global optimization
  - Peephole optimization
  - ...
Improve code quality and efficiency
  ▶ Increase Performance
  ▶ Reduce code size

Optimization on different abstraction levels
  ▶ High-level programming language
  ▶ Intermediate code
  ▶ Machine code

Optimization techniques
  ▶ Local optimization
  ▶ Global optimization
  ▶ Peephole optimization
  ▶ ...
# Table of Contents

1. **Introduction**
   - Context and Motivation

2. **Peephole Optimization**
   - General
     - Replacement Rules
     - Pattern Matching

3. **Demo**

4. **Replacement Rules Derivation**
   - Finding replacement Rules
   - Automatic rules derivation
   - Validation of Equivalence
   - Cost Comparison

5. **Conclusion**
   - References
   - Peephole Optimization in LLVM
   - Execution Test on other Intermediate Languages
# Peephole Optimization

## Usage Context

- Used within modern compiler (e.g. GCC, LLVM, ACK, ...)
- Typically applied on intermediate or machine code

## Optimization procedure

- Optimize code in a small moving window (peephole)
- Iterative Optimization
Peephole Optimization

; <label>:42
  store double 0.000000e+00, double* %det, align 8
  store i32 0, i32* %j1, align 4
  br label %43

; <label>:43
  %44 = load i32* %j1, align 4
  %45 = load i32* %2, align 4
  %46 = icmp slt i32 %44, %45
  br i1 %46, label %47, label %157

; <label>:47
  %48 = load i32* %2, align 4
  %49 = sub nsw i32 %48, 1
  %50 = sext i32 %49 to i64
  %51 = mul i64 %50, 8
  %52 = call i8* @malloc(i64 %51)
  %53 = bitcast i8* %52 to double**
  store double** %53, double** %m, align 8
  store i32 0, i32* %i, align 4
  br label %54

; <label>:54
  %55 = load i32* %i, align 4
  %56 = load i32* %2, align 4
  %57 = sub nsw i32 %56, 1
  %58 = icmp slt i32 %55, %57
  br i1 %58, label %59, label %73

; <label>:59
  %60 = load i32* %2, align 4
  %61 = sub nsw i32 %60, 1
  %62 = sext i32 %61 to i64
  %63 = mul i64 %62, 8
  %64 = call i8* @malloc(i64 %63)
Peephole Optimization

; <label>:42
  store double 0.000000e+00, double* %det, align 8
  store i32 0, i32* %j1, align 4
  br label %43

; <label>:43
  %44 = load i32* %j1, align 4
  %45 = load i32* %2, align 4
  %46 = icmp slt i32 %44, %45
  br i1 %46, label %47, label %157

; <label>:47
  %48 = load i32* %2, align 4
  %49 = sub nsw i32 %48, 1
  %50 = sext i32 %49 to i64
  %51 = mul i64 %50, 8
  %52 = call i8* @malloc(i64 %51)
  %53 = bitcast i8* %52 to double**
  store double** %53, double*** %m, align 8
  store i32 0, i32* %i, align 4
  br label %54

; <label>:54
  %55 = load i32* %i, align 4
  %56 = load i32* %2, align 4
  %57 = sub nsw i32 %56, 1
  %58 = icmp slt i32 %55, %57
  br i1 %58, label %59, label %73

; <label>:59
  %60 = load i32* %2, align 4
  %61 = sub nsw i32 %60, 1
  %62 = sext i32 %61 to i64
  %63 = mul i64 %62, 8
  %64 = call i8* @malloc(i64 %63)
Peephole Optimization

```
; <label>:42
  store double 0.000000e+00, double* %det, align 8
  store i32 0, i32* %j1, align 4
  br label %43

; <label>:43
  %44 = load i32* %j1, align 4
  %45 = load i32* %2, align 4
  %46 = icmp slt i32 %44, %45
  br i1 %46, label %47, label %157

; <label>:47
  %48 = load i32* %2, align 4
  %49 = sub nsw i32 %48, 1
  %50 = sext i32 %49 to i64
  %51 = shl i64 %50, 3
  %52 = call i8* @malloc(i64 %51)
  %53 = bitcast i8* %52 to double**
  store double** %53, double*** %m, align 8
  store i32 0, i32* %i, align 4
  br label %54

; <label>:54
  %55 = load i32* %i, align 4
  %56 = load i32* %2, align 4
  %57 = sub nsw i32 %56, 1
  %58 = icmp slt i32 %55, %57
  br i1 %58, label %59, label %73

; <label>:59
  %60 = load i32* %2, align 4
  %61 = sub nsw i32 %60, 1
  %62 = sext i32 %61 to i64
  %63 = mul i64 %62, 8
  %64 = call i8* @malloc(i64 %63)
```

Peephole Optimization

; <label>:42
  store double 0.000000e+00, double* %det, align 8
  store i32 0, i32* %j1, align 4
  br label %43

; <label>:43
  %44 = load i32* %j1, align 4
  %45 = load i32* %2, align 4
  %46 = icmp sgt i32 %44, %45
  br i1 %46, label %47, label %157

; <label>:47
  %48 = load i32* %2, align 4
  %49 = sub nsw i32 %48, 1
  %50 = sext i32 %49 to i64
  %51 = mul i64 %50, 8
  %52 = call i8* @malloc(i64 %51)
  %53 = bitcast i8* %52 to double**
  store double** %53, double*** %m, align 8
  store i32 0, i32* %i, align 4
  br label %54

; <label>:54
  %55 = load i32* %i, align 4
  %56 = load i32* %2, align 4
  %57 = sub nsw i32 %56, 1
  %58 = icmp sgt i32 %55, %57
  br i1 %58, label %59, label %73

; <label>:59
  %60 = load i32* %2, align 4
  %61 = sub nsw i32 %60, 1
  %62 = sext i32 %61 to i64
  %63 = mul i64 %62, 8
  %64 = call i8* @malloc(i64 %63)

Rule Database

%0 = mul i32 %1, 8
=> %0 = lsh i32 %1, 3
Peephole Optimization

; <label>:42
  store double 0.000000e+00, double* %det, align 8
  store i32 0, i32* %j1, align 4
  br label %43

; <label>:43
  %44 = load i32* %j1, align 4
  %45 = load i32* %2, align 4
  %46 = icmp slt i32 %44, %45
  br i1 %46, label %47, label %157

; <label>:47
  %48 = load i32* %2, align 4
  %49 = sub nsw i32 %48, 1
  %50 = sext i32 %49 to i64
  %51 = mul i64 %50, 8
  %52 = call i8* @malloc(i64 %51)
  %53 = bitcast i8* %52 to double**
  store double** %53, double*** %m, align 8
  store i32 0, i32* %i, align 4
  br label %54

; <label>:54
  %55 = load i32* %i, align 4
  %56 = load i32* %2, align 4
  %57 = sub nsw i32 %56, 1
  %58 = icmp slt i32 %55, %57
  br i1 %58, label %59, label %73

; <label>:59
  %60 = load i32* %2, align 4
  %61 = sub nsw i32 %60, 1
  %62 = sext i32 %61 to i64
  %63 = mul i64 %62, 8
  %64 = call i8* @malloc(i64 %63)

Rule Database

%0 = mul i32 %1, 8
=> %0 = lsh i32 %1, 3

Table of Contents

1 Introduction
   - Context and Motivation

2 Peephole Optimization
   - General
   - Replacement Rules
   - Pattern Matching

3 Demo

4 Replacement Rules Derivation
   - Finding replacement Rules
   - Automatic rules derivation
   - Validation of Equivalence
   - Cost Comparison

5 Conclusion
   - References
   - Peephole Optimization in LLVM
   - Execution Tests on the Intel platform
Replacement Rules

- Replace Sequence of Instruction with faster and/or fewer Instructions
- Many different Replacement Techniques
Null Sequences

- Remove useless instructions

Example

LOV A
LOC 0
ADD
Null Sequences

- Remove useless instructions

Example

LOV A      LOV A
LOC 0 →
ADD
Constant folding

- Replace constant expressions with its result

Example

LOC 8
LOC 2
ADD
Constant folding

- Replace constant expressions with its result

Example

LOC 8  →  LOC 10
LOC 2  ADD
Algebraic Laws

- Use algebraic laws to simplify expressions

Example

LOV A
LOC 7
ADD
LOC 5
SUB

\[(A + 7) - 5\]
### Algebraic Laws

Use algebraic laws to simplify expressions

### Example

<table>
<thead>
<tr>
<th>LOV A</th>
<th>LOV A</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC 7</td>
<td>LOC 7</td>
</tr>
<tr>
<td>ADD</td>
<td>→</td>
</tr>
<tr>
<td>LOC 5</td>
<td>SUB</td>
</tr>
<tr>
<td>SUB</td>
<td>ADD</td>
</tr>
</tbody>
</table>

\[
(A + 7) - 5 = A + (7 - 5)
\]
Common Techniques

Strength reduction

- Replace stronger operations with weaker ones
- Often applicable to multiplication or exponentiation in loops

Example

LOV A
LOC 2
MUL
Strength reduction

- Replace stronger operations with weaker ones
- Often applicable to multiplication or exponentiation in loops

Example

<table>
<thead>
<tr>
<th>LOV A</th>
<th>LOV A</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC 2</td>
<td>DUP 2</td>
</tr>
<tr>
<td>MUL</td>
<td>ADD</td>
</tr>
</tbody>
</table>
Strength reduction

- Replace stronger operations with weaker ones
- Often applicable to multiplication or exponentiation in loops

Example

<table>
<thead>
<tr>
<th>LOV A</th>
<th>→</th>
<th>LOV A</th>
<th>→</th>
<th>LOV A</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC 2</td>
<td></td>
<td>DUP 2</td>
<td></td>
<td>LOC 1</td>
</tr>
<tr>
<td>MUL</td>
<td></td>
<td>ADD</td>
<td></td>
<td>SHL</td>
</tr>
</tbody>
</table>
Special instructions

- Replace Instructions with specialized Instruction
- Usually more relevant on machine code

Example

LOV A
LOC 1
ADD
STV A
Special instructions

- Replace Instructions with specialized Instruction
- Usually more relevant on machine code

Example

| LOV A   | → | INV A   |
| LOC 1   |   | ADD     |
| STV A   |   |         |
# Table of Contents

1. Introduction  
   - Context and Motivation

2. Peephole Optimization  
   - General  
   - Replacement Rules  
   - Pattern Matching

3. Demo

4. Replacement Rules Derivation  
   - Finding replacement Rules  
   - Automatic rules derivation  
   - Validation of Equivalence  
   - Cost Comparison

5. Conclusion  
   - References  
   - Peephole Optimization in LLVM  
   - Execution Tests on the Intel microarchitecture
Many instruction sequences have same semantics
Generalize rules to avoid (semantically) identical rules
Different kinds of matching strategies

Example

<table>
<thead>
<tr>
<th>LOC 7</th>
<th>LOC 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>LOC 5</td>
</tr>
<tr>
<td>LOC 5</td>
<td>SUB</td>
</tr>
<tr>
<td>SUB</td>
<td>ADD</td>
</tr>
</tbody>
</table>
Many instruction sequences have same semantics
Generalize rules to avoid (semantically) identical rules
Different kinds of matching strategies

**Example**

<table>
<thead>
<tr>
<th>LOC 7</th>
<th>LOC 7</th>
<th>LOC 6</th>
<th>LOC 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>LOC 5</td>
<td>ADD</td>
<td>SUB</td>
</tr>
<tr>
<td>LOC 5</td>
<td>SUB</td>
<td>LOC 5</td>
<td>SUB</td>
</tr>
<tr>
<td>SUB</td>
<td>ADD</td>
<td>SUB</td>
<td>ADD</td>
</tr>
</tbody>
</table>
Many instruction sequences have same semantics

Generalize rules to avoid (semantically) identical rules

Different kinds of matching strategies

Example

<table>
<thead>
<tr>
<th>LOC 7</th>
<th>LOC 7</th>
<th>LOC 6</th>
<th>LOC 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>LOC 5</td>
<td>ADD</td>
<td>LOC 5</td>
</tr>
<tr>
<td>LOC 5</td>
<td>SUB</td>
<td>LOC 5</td>
<td>SUB</td>
</tr>
<tr>
<td>SUB</td>
<td>ADD</td>
<td>SUB</td>
<td>ADD</td>
</tr>
</tbody>
</table>

...
Many instruction sequences have same semantics
Generalize rules to avoid (semantically) identical rules
Different kinds of matching strategies

Example

<table>
<thead>
<tr>
<th>LOC %0</th>
<th>LOC %0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>LOC %1</td>
</tr>
<tr>
<td>LOC %1</td>
<td>SUB</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
</tr>
</tbody>
</table>
Demo
Table of Contents

1 Introduction
   Context and Motivation

2 Peephole Optimization
   General
   Replacement Rules
   Pattern Matching

3 Demo

4 Replacement Rules Derivation
   Finding replacement Rules
   Automatic rules derivation
   Validation of Equivalence
   Cost Comparison

5 Conclusion
   References
   Peephole Optimization in LLVM
   Execution Testing and future directions
How to find replacement rules?

Manual rule creation

- Replacement rules written by hand
- Using common techniques

Automatic rules derivation

- Deriving rules using superoptimization technique
Table of Contents

1 Introduction
   Context and Motivation

2 Peephole Optimization
   General
   Replacement Rules
   Pattern Matching

3 Demo

4 Replacement Rules Derivation
   Finding replacement Rules
   **Automatic rules derivation**
   Validation of Equivalence
   Cost Comparison

5 Conclusion
   References
   Peephole Optimization in LLVM
   Execution Tests on the datasets listed...
Superoptimization

- Given a set of instruction sequences to optimize
- Calculate every possible instruction sequence out of a set of instructions
- Compare outcome of calculated sequences to original sequences
- Check if new found equivalent sequence has lower cost
Superoptimization

- Given a set of instruction sequences to optimize
- Calculate every possible instruction sequence out of a set of instructions
  - Compare outcome of calculated sequences to original sequences
  - Check if new found equivalent sequence has lower cost
Superoptimization

- Given a set of instruction sequences to optimize
- Calculate every possible instruction sequence out of a set of instructions
- Compare outcome of calculated sequences to original sequences
- Check if new found equivalent sequence has lower cost
Superoptimization

- Given a set of instruction sequences to optimize
- Calculate every possible instruction sequence out of a set of instructions
- Compare outcome of calculated sequences to original sequences
- Check if new found equivalent sequence has lower cost
Table of Contents

1 Introduction
   Context and Motivation

2 Peephole Optimization
   General
   Replacement Rules
   Pattern Matching

3 Demo

4 Replacement Rules Derivation
   Finding replacement Rules
   Automatic rules derivation
   Validation of Equivalence
   Cost Comparison

5 Conclusion
   References
   Peephole Optimization in LLVM
   Execution Tests on the intermediate
Equivalence Test

**Execution Test: Process**
- Run both instruction sequences on machine
- Use multiple vectors as input
- Compare outcome
- Discard sequences with not matching results

**Execution Test: Property**
- Stochastical test
- Accuracy depends on test effort
## Execution Test

### Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Original</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOV X</td>
<td>LOV X</td>
<td></td>
</tr>
<tr>
<td>LOV Y</td>
<td>LOV Y</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td>SUB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input Vector (X,Y)</th>
<th>Stack</th>
<th>Original</th>
<th>Stack</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>[0]</td>
<td>[0]</td>
<td>[0]</td>
<td>✓</td>
</tr>
<tr>
<td>(1,1)</td>
<td>[2]</td>
<td></td>
<td>[0]</td>
<td>✗</td>
</tr>
</tbody>
</table>
**Execution Test**

**Example**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Original</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOV X</td>
<td>LOV X</td>
<td></td>
</tr>
<tr>
<td>LOC 2</td>
<td>SHR</td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>SHL</td>
<td></td>
</tr>
<tr>
<td>LOC 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Input Vector** | **Stack**

<table>
<thead>
<tr>
<th>(X)</th>
<th>Original</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>[0]</td>
<td>[0] ✓</td>
</tr>
<tr>
<td>(1)</td>
<td>[0]</td>
<td>[0] ✓</td>
</tr>
<tr>
<td>(65535)</td>
<td>[65534]</td>
<td>[65534] ✓</td>
</tr>
</tbody>
</table>
# Table of Contents

1. **Introduction**  
   - Context and Motivation

2. **Peephole Optimization**  
   - General  
   - Replacement Rules  
   - Pattern Matching

3. **Demo**

4. **Replacement Rules Derivation**  
   - Finding replacement Rules  
   - Automatic rules derivation  
   - Validation of Equivalence  
   - **Cost Comparison**

5. **Conclusion**  
   - References  
   - Peephole Optimization in LLVM  
   - Execution Test results and analysis
Cost Comparison

Cost Functions

- Runtime
- Code size
- Memory usage
Conclusion

- Optimization technique used in modern compilers
- Improve code quality and efficiency
- Using replacement rules for optimization
- Generalize with patterns
- Automatic generation of rules possible
  - Generated optimizer speedup of 1.7 to factor of 10 over conventional optimizer
References

- Elif Aktolga and Supervisor Dr. Des Watson. Pattern Matching Strategies for Peephole Optimisation
- Henry Massalin. Department of Computer Science Columbia University New York, NY 10027. Superoptimizer - A Look at the Smallest Program
Example

```c
int vadd(int a)
{
    return a*2
}
```

**llvm IR code**

```llvm
%a.addr = alloca i32, align 4
store i32 %a, i32* %a.addr, align 4
%0 = load i32* %a.addr, align 4
%mul = mul nsw i32 %0, 2
ret i32 %mul

→ %mul = shl i32 %0, 1
```
### Example

<table>
<thead>
<tr>
<th>Original Sequence</th>
<th>Replacement Candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ld r0, x} )</td>
<td>( \text{ld r0, x} )</td>
</tr>
<tr>
<td>( \text{ld r1, y} )</td>
<td>( \text{ld r1, y} )</td>
</tr>
<tr>
<td>( \text{add r0, r1} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vector ((x,y))</th>
<th>Original Sequence</th>
<th>Replacement Candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0,0))</td>
<td>((0,0))</td>
<td>((0,0)) ✔</td>
</tr>
<tr>
<td>((1,1))</td>
<td>((2,1))</td>
<td>((1,1)) ✗</td>
</tr>
</tbody>
</table>
Generate boolean formula for machine states
  ▶ Register
  ▶ Memory
  ▶ Stack

Express each instruction as formula that changes machine state

Express equivalence relation of final machine states as constraint

Use SAT solver to check if constraint holds
Remove multiplication from a loop

Occurs often during array access

Example

```c
for (I = 0, A = 0; I < 10; ++I) {
    A = I * 4;
    ...
}
```
▪ Remove multiplication from a loop
▪ Occurs often during array access

Example

```c
for (I = 0, A = 0; I < 10; ++I) {
    ...
    A += 4;
}
```
## Strength Reduction in Loops

### Example

<table>
<thead>
<tr>
<th>ZRV I</th>
<th>ZRV I</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZRV A</td>
<td>ZRV A</td>
</tr>
<tr>
<td>LOV I</td>
<td>LOV I</td>
</tr>
<tr>
<td>LOC 10</td>
<td>LOC 10</td>
</tr>
<tr>
<td>BGE 7</td>
<td>BGE 7</td>
</tr>
<tr>
<td>LOC 4 →</td>
<td>LOC 4</td>
</tr>
<tr>
<td>LOV I</td>
<td>LOC 4</td>
</tr>
<tr>
<td>MUL</td>
<td>LOV A</td>
</tr>
<tr>
<td>STV A</td>
<td>ADD</td>
</tr>
<tr>
<td>...</td>
<td>STV A</td>
</tr>
<tr>
<td>INV I</td>
<td>INV I</td>
</tr>
<tr>
<td>BRA -8</td>
<td>BRA -8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>