Lecture 21: Peephole optimization

David Hovemeyer

November 18, 2024

601.428/628 Compilers and Interpreters

Today

- Peephole optimization

Peephole optimization

Code generation

- The main responsibility of a code generator is to generate *working* code
- It's ok to generate *inefficient* code, especially if it will be easy to remove the inefficiencies later
- But...how easy will it be to remove the inefficiences?

Peephole optimization

- A basic code generator will generate *working* code using specific *idioms*
- If these idioms are easy to recognize, we can replace them with better (more efficient) idioms!
  - Will this work?
  - Under what circumstances does replacing a code sequence preserve correctness?

Before

```
movq    %rdx, %r10
imulq   $8, %r10
movq    %r10, %rsi
movq    %r9, %r10
addq    %rsi, %r10
movq    %r10, %r8
movq    (%r8), %rcx
```

After

```
movq      (%r9,%rdx,8), %rcx
```

Pattern/transformation

```
// Simplify 64 bit ALU operations
pm(
  // match instructions
  // Operands:
  //   A = first (left) source operand
  //   B = temporary code register (probably %r10)
  //   C = second (right) source operand
  //   D = destination operand (probably an allocated temporary)
  {
    matcher( m_opcode(MINS_MOVQ), { m_mreg(A), m_mreg(B) } ),
    matcher( m_opcode_alu_q(A),   { m_any(C), m_mreg(B) } ),
    matcher( m_opcode(MINS_MOVQ), { m_mreg(B), m_mreg(D) } ),
  },

  // rewrite
  {
    gen( g_opcode(MINS_MOVQ), { g_prev(A), g_prev(D) } ),
    gen( g_opcode(A),         { g_prev(C), g_prev(D) } ),
  },

  "B", // B must be dead
  "CD" // C and D must be different locations
),
```

Effect

```
movq      %rdx, %r10
imulq     $8, %r10
movq      %r10, %rsi
```

is transformed into

```
movq      %rdx, %rsi
imulq     $8, %rsi
```

Effect

```
movq      %r9, %r10
addq      %rsi, %r10
movq      %r10, %r8
```

is transformed into

```
movq      %r9, %r8
addq      %rsi, %r8
```

After transformations

```
movq    %rdx, %rsi
imulq   $8, %rsi
movq    %r9, %r8
addq    %rsi, %r8
movq    (%r8), %rcx
```

Pattern/transformation

```
// Simplify 64 bit array loads with computed element address
pm(
  // match instructions
  // Operands:
  {
    matcher( m_opcode(MINS_MOVQ),  { m_mreg(A),     m_mreg(C) } ),
    matcher( m_opcode(MINS_IMULQ), { m_imm(8),      m_mreg(C) } ),
    matcher( m_opcode(MINS_MOVQ),  { m_mreg(D),     m_mreg(E) } ),
    matcher( m_opcode(MINS_ADDQ),  { m_mreg(C),     m_mreg(E) } ),
    matcher( m_opcode(MINS_MOVQ),  { m_mreg_mem(E), m_mreg(F) } ),
  },

  // rewrite
  {
    gen( g_opcode(MINS_MOVQ), { g_mreg_mem_idx(D, A, 8), g_prev(F) } ),
  },

  // make sure C and E are dead
  "CE"
),
```

Effect

```
movq      %rdx, %rsi
imulq     $8, %rsi
movq      %r9, %r8
addq      %rsi, %r8
movq      (%r8), %rcx
```

is transformed into

```
movq      (%r9,%rdx,8), %rcx
```

Before

```
movl    %r14d, %r10d
cmpl    $250000, %r10d
setl    %r10b
movzbl  %r10b, %r11d
movl    %r11d, %r9d
cmpl    $0, %r9d
jne     .L10
```

After

```
cmpl    $250000, %r14d
jl      .L10
```

Pattern/transformation

```
// simplify comparisons
pm(
  // match instructions
  {
    matcher( m_opcode(MINS_MOVL), { m_mreg(A), m_mreg(B) } ),
    matcher( m_opcode(MINS_CMPL), { m_any(C), m_mreg(B) } ),
  },

  // rewrite
  {
    gen( g_opcode(MINS_CMPL), { g_prev(C), g_prev(A) } ),
  },

  // make sure that B is dead
  "B"
),
```

Effect

```
movl    %r14d, %r10d
cmpl    $250000, %r10d
```

is transformed into

```
cmpl    $250000, %r14d
```

After transformation

```
cmpl    $250000, %r14d
setl    %r10b
movzbl  %r10b, %r11d
movl    %r11d, %r9d
cmpl    $0, %r9d
jne     .L10
```

Pattern/transformation

```
// Simplify control flow (jump if true)
pm(
  // match instructions
  {
    matcher( m_opcode(MINS_CMPL),       { m_any(A), m_any(B) } ),
    matcher( m_opcode(MINS_SETL, 6, A), { m_mreg(C) } ),
    matcher( m_opcode(MINS_MOVZBL),     { m_mreg(C), m_mreg(D) } ),
    matcher( m_opcode(MINS_MOVL),       { m_mreg(D), m_mreg(E) } ),
    matcher( m_opcode(MINS_CMPL),       { m_imm(0), m_mreg(E) } ),
    matcher( m_opcode(MINS_JNE) ,       { m_label(F) } ),
  },

  // rewrite
  {
    gen( g_opcode(MINS_CMPL),    { g_prev(A), g_prev(B) } ),
    gen( g_opcode_j_from_set(A), { g_prev(F) } ),
  },

  // make sure that C, D, and E are dead
  "CDE"
),
```

Effect

```
cmpl    $250000, %r14d
setl    %r10b
movzbl  %r10b, %r11d
movl    %r11d, %r9d
cmpl    $0, %r9d
jne     .L10
```

is transformed into

```
cmpl    $250000, %r14d
jl      .L10
```

Preserving correctness

- When an idiom is simplified, some instructions assigning to register might be eliminated
- So, the transformation is only correct if those registers are not alive at the end of the idiom
- Solution: liveness dataflow analysis for machine registers
  - Don't apply transformation if any eliminated values will be needed elsewhere in the code
- In some cases it may be necessary to guarantee that matched operands are not the same
  - E.g., because the transformed code updates a register in a different way than the original code, so if that same register is used as a source operand, its value would be different than expected

Implementing peephole transformations

- These are local transformations (within basic block)
  - Build control-flow graph, transform each basic block
- Multiple rounds can be necessary
  - One transformation can enable another

Implementing peephole transformations (continued)

- Primary challenges:
  - Matching instruction sequences
  - Replacing matched sequence with replacement (substituting matched opcodes/operands as appropriate)
- Peephole optimization can be *very* effective at improving code quality
  - E.g., example 29 (Linux, Ryzen 5 5600X):
    - After LVN+reg alloc, 0.28 s
    - With peephole optimization, 0.19 s
- It feels like cheating!

Interaction with register allocation

- As implemented in the reference solution, the low-level peephole optimizer runs *after* register allocation
- However, it can eliminate the use of some machine registers!
  - So, a machine register might be allocated but then not used
- Could register allocation be deferred until after low-level code generation?
  - Is a bit of a chicken-and-egg problem, since whether an operand is a machine register or memory location affects which instruction(s) can be emitted

Possible approach

Initial register allocation (assigning machine registers to virtual registers) assumes an unlimited number of "fake" machine registers.

- These are placeholders for real machine registers
- No spills or restores are emitted

Peephole optimizer will eliminate some references to (fake) machine registers.

The "real" local register allocator runs on low-level code after peephole optimization:

- Each "fake" machine register is replaced by a real one
- Spills/restores might be necesssary