Lecture 24: Peephole optimization, JIT compilation, course wrap-up

David Hovemeyer

December 5, 2022

601.428/628 Compilers and Interpreters

Today

- Peephole optimization
- JIT compilation
- Course wrap-up

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)
// Operands:
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 D and E are dead
  "DE"
),
```

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

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:
    - After LVN+reg alloc, 0.46 s
    - With peephole optimization, 0.24s
- It feels like cheating!

JIT Compilation

JIT compilation

Bytecode interpreters can achieve reasonably good performance, they are still significantly slower than native machine code.

Modern JIT-based language runtimes (e.g., JVM with Hotspot) can be competitive with optimizing ahead-of-time compilers

- Sometimes they are superior!

The essential challenge of JIT compilation

An ahead-of-time compiler can liberally perform optimizations that are expensive in terms of runtime and memory use

A JIT compiler is more constrained, because its running time and memory use compete with the running program

Techniques used by JIT-enabled language runtimes:

- Use a variety of execution strategies
- Do JIT compilation selectively
- Do optimizations selectively

The fundamental principle of performance

The efficiency of code matters in proportion to how frequently it is executed

So, effort should be focused on improving the performance of the (typically small) fraction of code in the program that actually has an impact on performance

Execution strategies, profiling

A language runtime can use a variety of strategies to execute program code

Strategy	Advantage	Disadvantage
Interpretation	Easy, quick to start	Code execution is slow
Simple JIT compilation	Somewhat faster code execution	JIT compilation takes time, missed optimization opportunities
Optimized JIT compilation	Fastest code execution	JIT compilation could be expensive (time and memory)

Idea:

- Always start with interpretation
- Continually profile the running program to determine where time is spent
- As performance-critical procedures are found, use more expensive JIT compilation

Profiling

Language runtime must have a way to record where time is being spent

- Overhead of instrumentation for profiling must be kept low!
- Counting procedure calls is fairly cheap
- Counting basic block executions: much more expensive
  - Could do this infrequently

Simple JIT compilation

A “first stage” JIT can use simple and fast optimization and code generation techniques

- Might not produce the best code possible, but can do significantly better than interpretation

Again, see Gosling, “Java Intermediate Bytecodes”

Optimized JIT compilation

Once the language runtime has identified performance critical code (e.g., a core loop computation), it can apply more sophisticated optimization techniques

Compiler IR and optimization passes must be designed to be runtime and memory efficient

30+ years of research on this (hard to summarize)

Lots of difficult issues: multiple program threads, interaction with garbage collector, on-stack replacement, etc.

Course wrap-up

Where we are

- We've implemented
  - A realistic AST-based interpreter
  - A realistic ahead-of-time optimizing compiler
- Perhaps not state of the art, but good enough to be useful

Where to go next? (interpreter)

- Bytecode interpreters
  - Higher performance than AST-based interpreters
  - Not too difficult to implement
- Garbage collection
- JIT compilation/managed runtime

Where to go next? (compiler)

- We ended with compilation techniques that produce reasonably good code
  - Within a factor of 2 of `gcc -O2`?
- Next frontier: *global techniques*
  - SSA form: suitable for global analyses and transformations because each the representation is explicit about how values are propagated across basic-block boundaries
  - Global register allocation (e.g., by graph coloring)
  - Loop-invariant code motion (remove computations out of loops if the computed value is guaranteed to be the same on every iteration)