Peephole Optimization

Lecture 12: Peephole Optimization
A. Pnueli

Honors Compilers, NYU, Spring, 2007

Final pass over generated code:

examine a few consecutive instructions:

2 to 4
See if an obvious replacement is possible: Store/Load

Final pass over generated code:

x = y
z = w

Examine a few consecutive instructions:
2 to 4

Use algebraic identities

Worth recognizing single instructions with a constant operand:

A

A

A

A / 1 = A

Moredelicate with floating-point numbers.

/ 1 = A

A * 0 = 0

A * 1 = A

A * 2 = A + A

Is this ever helpful?

Why bother to correct such obvious junk code?

Why would anyone write X = 1;

Special-case individual instructions

Use algebraic identities

knowledge of mema

Can eliminate second instruction without needing any global

pairs

MOV mema = %eax

MOV %eax = mema

More delicate with floating-point numbers.

As other optimizations.

Also, seemingly redundant code can be produced by

a = b * MAX-TASKS;

...#define MAX-TASKS 1

In fact one might write

Why bother to correct such obvious junk code?

Why would anyone write X = 1?

Algebraic identities

Worth remembering single instructions with a constant operand:
Replace Multiply by Shift

A := A / 4;

Can be replaced by 2-bit left shift (signed/unsigned)

But must worry about overflow if language does not allow it; anyway (traditional C)

Language may allow it but shift right arithmetic is a well-known problem

If unsigned, can replace with „shift right“:

A := A / 4;

But shift right arithmetic may replace with „shift right“:

A := A / 4;

Can be replaced by 2-bit left shift (signed/unsigned)

A := A / 4;

Folding Jumps to Jumps

A Jump to an unconditional jump can copy the target address

A result label may become dead (unreachable)

JNE label

Can be replaced by

JMP label

... 

JNE label

Note similarity with efficient exponentiation method

Twee shifts, one subtract and one add, which may be effective:

const X := 125 = X * 128 - X * 4 + X

If multiplier is very slow (or on a machine with no multiplier instruction)

was negative

truncates towards zero if either operand

Prior to C99, implementations were allowed to

in most languages (−5)/2 = −2

Which is −3 not −2

5 representable as

Shiftright and use sign bit to shift most significant bits

Arithmetic Right Shift:

SAR

The Right Shift Problem

Addition Chains for Multiplication

Note similarity with efficient exponentiation method

Twee shifts, one subtract and one add, which may be effective:

const operand into a sum of powers of two can be

intrinsic (like the original SPARC), decomposing a

If multiplier is very slow (or on a machine with no multiplier instruction)
Lecture 12: Peephole Optimization

A. Pnueli

Jump to Return

A jump to a "return" can be replaced by a "return".

Advantages of Tail Recursion Elimination

Saves time: an assignment and jump are faster than a call with one parameter.

Saves stack space: converts linear stack usage to constant usage.

In languages with no loops, this may be a required optimization: specified in Scheme standard.

Tail Recursion Elimination at the Instruction Level

Consider the sequence on the X86:

CALL func

Can generate instead:

JMP func

Now RET in func returns to original caller, because

CALL func pushes return address on stack, RET in body

JMP func

Can generate instead:

RET

CALL func

RET

JMP func

As a result, label may become dead

Jump to Return

A. Pnueli

Honors Compilers, NYU, Spring 2007

297
Lecture 12: Peephole Optimization

A. Pnueli

Peephole Optimization in the REALIA COBOL Compiler

Full compiler for standard COBOL targeted to the IBM PC. Now distributed by Computer Associates. Run in 150K bytes, but must be able to handle very large programs that run on mainframes.

No global optimization possible: multiple linear passes over code, no global data, no flow graph.

Multiple peephole optimizations, compiler iterates until code is stable. Each pass scans code backwards to minimize address recomputations.

Typical COBOL Control Structures

Process-Balance:

if Balance is negative then

else

Send-Bill

end-if.

Record-Credit

Send-Bill

Pb: cmp balance, 0

jnl L1

Call Sb

jmp L2

L1: Call Rc

L2: ret

Sb:

:::

ret

Rc:

:::

ret

Fold Jump to Return Statement

Pb: cmp balance, 0

jnl L1

Call Sb

jmp L2

L1: Call Rc

L2: ret

Sb:

:::

ret

Rc:

:::

ret

•

Perform Assembly: Equivalent to

Simple Assembly:

Perform Equivalent to Call Pb: cmp balance, 0

jnl L1

Call Sb

jmp L2

L1: Call Rc

L2: ret

Sb:

:::

ret

Rc:

:::

ret

•

Jump to “Return”
Corresponding Assembly

- JumptoFollowingInstructionisaNoop

\[ Pb: cmp balance, 0 \]
\[ jnl Rc \]
\[ jmp Sb \]
\[ ret \]

Final Code

- Final code as efficient as inlining.

\[ Pb: cmp balance, 0 \]
\[ jnl Rc \]
\[ jmp Sb \]
\[ ret \]

JumptoFollowingInstructionisaNoop

- Willbecomeuseless

\[ L1: jmp Rc \]
\[ ret \]
\[ Sp \]
\[ jmp Sb \]
\[ ret \]

CodeFollowingaJumpisUnreachable

- Tolered

\[ L1: jmp Rc \]
\[ ret \]
\[ Sp \]
\[ jmp Sb \]
\[ ret \]

Iterate till no further change

- Yield further optimization opportunities

All transformations are local.

- Each optimization may yield further optimization opportunities.

- Each optimization may yield further optimization opportunities.

- Final code as efficient as inlining.

- All optimizations are local. Each optimization may yield further optimization opportunities.
Arcane Tricks

Eliminating Max Jumps on X86

Consider typical maximum computation:

1. \[ A > B \] then \( C := A \);
2. \[ A \leq B \] then \( C := B \);

For simplicity assume all unsigned and all in registers.

Super-compiler:

1. Use simple-minded ASM code.
2. More instructions but no jumps.

3. Eliminating max jumps on X86


Consider typical maximum computation.

- One jump in either case:
  
  - L1: MOV B, C
  - JMP L2
  - MOV A, C
  - JNE L1
  - CMP A, B

- Simple minded ASM code.

More instructions but NO JUMPS.

- OR: B = A, C = \( \overline{A} \)
- AND: C = 0, A = \( \overline{A} \)
- CMP A, B

- CMP A, B

- MOV %eax, C
- SBB %eax, %eax
- AND %eax, %eax
- NOT %eax
- CMP %eax, 0
- SBB %eax, %eax
- AND %eax, %eax

More instructions but NO JUMPS.

- OR: B = A, C = \( \overline{A} \)
- AND: C = 0, A = \( \overline{A} \)
- CMP A, B

More instructions but NO JUMPS.

- OR: B = A, C = \( \overline{A} \)
- AND: C = 0, A = \( \overline{A} \)
- CMP A, B

End if,

if \( C = B \) then

else

if \( A \leq B \) then

else

if \( A > B \) then

else

if \( B > A \) then

else

if \( A > B \) then

else

if \( A \leq B \) then

else

if \( B > A \) then

else

if \( A > B \) then

else

if \( A \leq B \) then

else

if \( B > A \) then

else

if \( A > B \) then

else

if \( A \leq B \) then

else

if \( B > A \) then

else

if \( A > B \) then