Peephole Optimization
Special-case individual instructions

Use algebraic identities

Knowledge of mema

Can eliminate second instruction without needing any global pairs

See if an obvious replacement is possible: store/load

Examine a few consecutive instructions: 2 to 4

Final pass over generated code:
A.

Worthwhlie recognizing single instructions with a constant operand:

\[ A/1 = A \]
\[ 0 \times A = 0 \]
\[ A \times 1 = A \]
\[ A \times 2 = A + A \]

More delicate with floating-point numbers.

Algebraic Identities
other optimizations. This is an important effect.

Also, seemingly redundant code can be produced by

\[
a = b \ast \text{MAX-TASKS};
\]

\[
\ldots
\]

\[
\text{#define MAX-TASKS 1}
\]

In fact one might write

Why bother to correct such obvious junk code?

Why would anyone write 1 \ast 1?

Is This Ever Helpful?
Replace Multiply by Shift

A := A \times 4;

A := A / 4;

If unsigned, can replace with "shift right".

But signed right arithmetic is a well-known problem.

Lenguage may allow it anyway (traditional C).

But must worry about overflow if language does.

Can be replaced by 2-bit left shift (signed/unsigned).
Addition Chaining for Multiplication

A. Pnueli

Lecture 12: Peephole Optimization

Note similarity with efficient exponentiation method • Faster than one multiply

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

If multiplication is very slow (or on a machine with no multiply instruction like the original SPARC), decomposing a constant operand into a sum of powers of two can be effective:

$$X = X \cdot 128 - X \cdot 4 + 4 + X$$

Honors Compilers, NYU, Fall, 2007
The Right Shift Problem

- Arithmetic Right Shift:
  - Shift right and use sign bit to fill most significant bits

Prior to C99, implementations were allowed to truncate towards towards or away from zero if either operand was negative.

In most languages, \((-5)/2 = -2\)

Which is -3 not -2

\[ \begin{align*}
  &\quad \begin{array}{c}
  \text{SAR} \\
  \text{-5 representable as}
  \end{array} \\
  &\begin{array}{c}
  \begin{array}{c}
  \text{0} \\
  \text{1}
  \end{array} \\
  \end{array}
\end{align*} \]

\[ \begin{align*}
  &\quad \begin{array}{c}
  \text{Shift right and use sign bit to fill most significant bits}
  \end{array} \\
  &\begin{array}{c}
  \begin{array}{c}
  \text{0} \\
  \text{1}
  \end{array} \\
  \end{array}
\end{align*} \]
As a result, lab1 may become dead (unreachable).

\[ \text{JNE lab2} \]

Can be replaced by

\[ \text{JMP lab2} \]

\[ \text{lab1} \]

\[ \text{lab2} \]

\[ \ldots \]

\[ \text{JNE lab1} \]

A jump to an unconditional jump can copy the target address.

Folding Jumps to Jumps

A. Pnueli
As a result label may become dead

RET

Can be replaced by

RET

label RET

...

JMP label

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

Tail Recursion Elimination

A subprogram is tail-recursivie if the last computation is a call to itself.

 Recursive call can be replaced by 

- added label at beginning of body
  
goto start;

  lis = lis.next;

  Recursive call can be replaced by

  end;

  else return last(lis.next);

  if lis.next=null then return lis;

  begin
  function last(lis: list type) return list type is

  a call to itself:

  Tail Recursion Elimination
Advantages of Tail Recursion Elimination

- Specified in Scheme standard.
- In language with no loops, this may be a required constant usage.
- Saves stack space: converts linear stack usage to call with one parameter.
- Saves time: an assignment and jump are faster than a call.
Consider the sequence on the X86:

CALL func
JMP func
RET

Can generate instead:
CALL func
JMP func
RET

Now RET in func returns to original caller, because

single return address on stack.

In Instruction Level Tail Recursion Elimination at the

A. Pnueli

Honors Compilers, NYU, Fall, 2007
Peephole Optimization

Lecture 12: Peephole Optimization in the REALIA

A. Pnueli

Peephole optimization recognizes address recomputation.

Multiple peephole optimizations, compiler iterates until code is stable. Each pass scan code backwards to

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

Large programs that run on mainframes

- Runs in 150k bytes, but must be able to handle very

- Full compiler for standard COBOL targeted to the IBM

Full compiler distributed by Computer Associates

COBOL Compiler

Honors Compilers, NYU, Fall 2007

Runsin150kbytes, butmustbeabletohandleverylargeprograms thatrunonmainframes

- Runs in 150K bytes, but must be able to handle very large programs that run on mainframes

- Full compiler for standard COBOL targeted to the IBM

- Full compiler distributed by Computer Associates

- COBOL Compiler

- Honors Compilers, NYU, Fall 2007
Typical COBOL Code: Control Structures and Perform Blocks

- Process-Balance.
  - if Balance is negative then
  - perform Send-Bill
  - perform Record-Credit
  - end-if.
  - Send-Bill.
  - Record-Credit.

Lecture 12: Peephole Optimization

A. Pnueli

Honors Compilers, NYU, Fall, 2007
Simple Assembly: Perform Equivalent to

Call

Pb: cmp balance, 0
    jnl L1
    Call Sb
    jmp L2
    Call Rc

L1: Call
L2: ret
Sb: ret
Rc: ret
Fold Jump to Return Statement
Corresponding Assembly

- Jump to unconditional jump

- folded

- will become useless

\[ \text{balance,0} \]

\[ \text{cmp} \]

\[ \text{jnl} \]

\[ \text{Call} \]

\[ \text{ret} \]

\[ \text{cmpbalance,0} \]

\[ \text{jnlL1} \]

- folded

\[ \text{will become useless} \]

\[ \text{Sb} : \]

\[ \text{Rc} : \]

\[ \text{L1: jmp} \]

\[ \text{L2: ret} \]

\[ \text{Sb: ret} \]

\[ \text{Rc: ret} \]

\[ \text{L1: jmp} \]

\[ \text{L2: ret} \]

\[ \text{Sb: ret} \]

\[ \text{Rc: ret} \]
Peephole Optimization

Code Following a Jump is Unreachable

- folded
- unreachable
- unreachable

Pb: cmp balance, 0
   jnl
   jmp
   ret

Rc:
   jmp
   ret
   ret

Sb:
   jmp
   ret
   ret

Rc:
   jmp
   ret
   ret
Jump to next instruction

Jump to Following Instruction is a No-op

Pb: cmp balance, 0

JumptoFollowingInstructionisaNoop
Iterate till no further change

Yield further optimization opportunities.

All transformations are local. Each optimization may

Final code as efficient as inlining.

```
ret
  ...
RC:
ret
  ...
Sp:
  \inl\ RC
Pb: cmp balance,0
```

Final Code
Lecture 12: Peephole Optimization

A. Pnueli

Arcane Tricks

Consider typical maximum computation

\[
\text{if } A > B \text{ then } C := A; \\
\text{else } C := B; \\
\text{endif;}
\]

For simplicity assume all unsigned and all in registers

End If;

\[
\text{end If;}
\]

\[
C := B; \\
\text{else} \\
C := A; \\
\text{if } A > B \text{ then}
\]

Honors Compilers, NYU, Fall 2007
Eliminating Max Jumps on X86

Simple minded ASM code

```
CMP A, B
JNAE L1
MOV A, C
JMP L2
L1: MOV B, C
L2: One jump in either case
```
Super-compiler: exhaustive search of instruction patterns to uncover similar tricks.

More instructions but no jumps:

- CMPA, B; CF=1 if B > A, CF=0 if A > B
- SBB%eax, %eax; all 1's if B > A, all 0's if A > B
- MOV%eax, C; all 0's if B > A, all 1's if A > B
- AND%eax, C; A if B > A, 0 if A > B
- AND B, %eax; all 0's if B > A, all 1's if A > B
- AND A, C; 0 if B > A, A if A > B
- OR%eax, C; 0 if B > A, A if A > B

Architectures-specific trick: use subtract with borrow instruction and carry flag.

Computing Max Without Jumps on X86