Register allocation
More on spilling

Intro to optimization

Register allocation review:
Simplify-select algorithm
   Remove node with degree < k, push on stack
   Recurse on reduced (simplified) graph
   Select color for top of stack after recursion returns

If no coloring for a node
   Spill node to memory
   May need to spill more than one
If node is spilled to memory:
   Insert store instruction to store value in memory
   Insert load instruction to retrieve value at every use point

Assume t100 to be spilled:
   mov %i0,t100
   ...
   add t100,t101,t101

Rewrite instructions as:
   store %i0,x(fp)
   ...
   load x(fp),txxx
   add txxx,t101,t101

NB: still want to assign txxx to register

Must re-run register allocation on new code, but...

Lifetime of txxx is very short
   Will interfere with fewer other registers
   Graph coloring more likely

Alternate (similar) approach: split a lifetime
   Create two (or more) lifetimes
   Add a move instruction at joins
1: mov 1, a
2: add a, a, c
3: loop:
4: and a, c, b
5: xor c, b, d
6: sub b, d, e
7: or d, e, f
8: add e, a, g
9: xor f, g, c
10: add g, c, a
11: and a, c, b
12: cmp g, b
13: bne loop
14: nop
Split graph may not be colorable
  May need to spill
  Split and spill interact well

Split intelligently…
  Create 3 lifetimes
  2 with usage
  Middle with no usage
  Middle ideal for spill

Want to spill temps with limited usage
  Minimal cost for spill
Want to spill long lived temps
  Further simplifies graph
Define cost function to order temps
  More usage adds to cost
  Divide cost by lifetime
Spill least costly temp with degree > k

Usage not simple count
Not every instruction executed once
  Within loops count more
  Within if counts less
Consider CFG
Usage in loop
  Count as more (often 10)
  Multiply by depth of loops
  Conditional usage counts as half
Spill has inherent cost
  Memory access slower than reg access
  Makes easy cost function
Split has only potential cost
  added move instruction(s)
  Move is cheaper
  May be coalesced
Need different heuristic

Never split inside loop body
  Multiplies cost of added move
  Biggest win in splitting long lifetimes
Find longest stretch without usage
  Split before, after stretch

```
loop1:
mov 1,i       
add 1,i,i     +
cmp 1,i,10     +
ble loop1     *
mov 8,j       *
add j,3,k     *
cmp k,1       *
bne exit      *
```

```
loop2:
sub 1,i,i     +
cmp 1,0       +
bgt loop2     *
```

exit:

life time of i
has long stretch
of instructions
not using i

[End of document]
split example

```assembly
loop1:
    mov 1, i1
    add i1, i1
    cmp i1, 10
    ble loop1
    mov 8, j
    add j, 3, k
    cmp k, i1
    bne exit

loop2:
    sub i1, i1
    cmp i1, 0
    bgt loop2

exit:
```

split example

```assembly
loop1:
    mov 1, i1
    add i1, i1
    cmp i1, 10
    ble loop1
    mov 8, j
    add j, 3, k
    cmp k, i1
    bne exit

loop2:
    sub i1, i1
    cmp i1, 0
    bgt loop2

exit:
```

split example

```assembly
loop1:
    mov 1, i1
    add i1, i1
    cmp i1, 10
    ble loop1
    mov 8, j
    add j, 3, k
    cmp k, i1
    bne exit

loop2:
    sub i1, i1
    cmp i1, 0
    bgt loop2

exit:
```
Introduction to optimization

Optimization: change the generated code to
reduce the cpu time required to execute the code,
reduce the memory used in executing the code
or both

Optimization cannot change any behavior
directly visible to the program and
defined by the language semantics

If code is correct
(uses only defined constructs)
optimization will generate same result
after optimization

What constructs are undefined?
order of evaluation in many languages
\(a[++] = b[++]\)
is \(a[0] == b[1]\) or \(a[1] == b[0]\)?
usage of uninitialized variables
int i;
a[i] == ?
Within these constraints the compiler is free to do anything it wants.

Three basic approaches to making code faster:
1. Do less work
   Don’t do things that don’t change the answer
2. Do work less often
   Don’t redo work that was already done
3. Do easier work
   Use a cheaper way to do work

1. Do less work.

Instructions that don’t have an effect on the result can be removed
   Coalesced moves are an easy example

Dead variable removal is more important
   If every def of an instruction is unused, the instruction has no effect
   Variable assigned to is dead
   Instruction can be removed
   Removing instruction may kill earlier variables

Dead variable removal
   Simplest data-flow based optimization
   Also based on liveness graph

   Variable assignment is dead if value of assignment never used
Recall meaning of data flow sets
  def : location is modified by instr
  out : value in location is exported (used at some later point)

For any assembler instruction
  If (singleton) def set not empty and
  intersection of def and out is empty
  Instruction is dead assignment
  Can be deleted if no side-effect

1: \texttt{a = b * 2} \hspace{1em} \texttt{def: (a), out: [a,b]}
2: \texttt{c = a + b} \hspace{1em} \texttt{def: (c), out: [b]}
3: \texttt{d = b / 2} \hspace{1em} \texttt{def: (d), out:[d]}
4: \texttt{\%o0 = d} \hspace{1em} \texttt{def : \%o0, out: \%o0}

Assume no more usage of a,b,c or d
Instruction 2
  Defines value for c, exports only b
  Intersection is empty
  c is dead variable at assignment
Delete instruction 2

Deleting instruction 2
  No longer any usage of a
  out for 1 is now just \{ b \}
  Intersection with def is empty
Delete instruction 1 also
2. Do work less often

Some work needs to be done
Just not many times
Best example: loop invariant instructions

Instr within a loop is executed many times
(in general)

if it computes same result every iteration
Why execute every time?
Compiler can be move it outside loop
Executed only once then

Another example
common subexpression elimination

consider code

\[ a = i \cdot j + 3; \]
\[ b = i \cdot j - 4; \]

Why compute \(i \cdot j\) twice?

Compute \(i \cdot j\) once and use the answer twice

3. Do cheaper work

Look for bargains

Simplest way is to use cheaper ops
\[ i = j \cdot 2 \text{ is the same as } i = j + j \]
\[ i = j \cdot 4 \text{ is the same as } i = j < < 2 \]
Multiply is much more expensive
than add or shift
May be cheaper to recompute than reload from memory
Is this legal?
   What if another process/thread wrote to memory
In C and C++,
   Compiler can assume no other writers, unless variable declared “volatile”
In Java
   Compiler must assume other writers

Optimizations scope
   range of instructions considered
Peephole optimizers
   Look at one or a few instructions
      e.g. reduce operator strength, remove jumps to jumps
Basic block optimization
   Can do some interesting optimizations
      Much easier when focusing on isolated blocks
Loop optimization
   Biggest bang for the buck
Function-wide optimization
Global optimization

When does optimization occur?
Three choices, depending on optimization:

1. Before code generation (on IR)
   Machine independent optimizations
2. Before register allocation (on assem)
   Many peephole
3. After register allocation
   Remaining peephole
Cover seven peephole optimizations:

- Redundant moves
- Constant folding
- Constant conditional branches
- Jump to jump
- Jump over jump
- Reduction in strength
- Special instructions

Common generate code that looks like

- move loc2 loc1
- move loc1 loc2
- Second instruction unnecessary, just need
  - move loc2 loc1
- More common (and more costly)
  - where loc1 or loc2 is memory
- This optimization can be performed at any stage

If one or both operands are constants, can do computation at compile time
For example:
- add const1 const2 loc2
  - Can be replaced with
    - move const1+const2,loc2
    - Especially important when combined with constant propagation
  - Can be done on IR
  - Can use tile patterns in code selection
If comparison for branch is constant
Can be turned into an unconditional branch or removed altogether

For example:

```assembly
cmp 0,1
beq L1
nop
L2:
```
can be replaced with
```
L2:
```

Alternatively, the code
```
cmp 0,1
bne L1
nop
L2:
```
Can be replaced with
```
br L1
nop
L2:
```
Can be performed at any stage

Many jumps to jumps generated
for example
```
br L1
...
L1: br L2
```
Can be replaced with
```
br L2
...
L1: br L2
```
Conditional branches
second opportunity for removing extra branches

For example:
```
beq L1
nop
br L2
nop
L1:  
```
can be replaced with
```
bnz L2
nop
L1:  
```

Not all operations are created equal
Want to use cheapest instruction available
Many are generic to all machines

<table>
<thead>
<tr>
<th>operation</th>
<th>replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>i^2</td>
<td>i+i</td>
</tr>
<tr>
<td>i^2n</td>
<td>i &lt;&lt; n</td>
</tr>
<tr>
<td>x^2</td>
<td>x*x</td>
</tr>
<tr>
<td>x/n</td>
<td>x*(1/n)</td>
</tr>
</tbody>
</table>

Machine independent, generally done on IR

Some are specific to a particular machine:

on x86

<table>
<thead>
<tr>
<th>operation</th>
<th>replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov reg,0</td>
<td>xor reg,reg</td>
</tr>
<tr>
<td>mov i,n</td>
<td>lea i,n(j)</td>
</tr>
<tr>
<td>add i,j</td>
<td></td>
</tr>
</tbody>
</table>

Best done after register allocation
Many machines offer special faster instructions
May be easier to recognize in peephole
than during code generation
Other peephole optimists may enable instruction
Always machine specific
Generally done after register allocation

consider x86 code
mov word[0(eax)],word[0(ebx)]
mov word[2(eax)],word[2(ebx)]
Can be replaced with
mov dword[0(eax)],dword[0(ebx)]
Combining four memory accesses into two

x86 (or mips) code
beq L1
mov r1,r2
L1:
can be replaced by cmov instruction
cmov eq,r1,r2
L1:
Loses branch
Very big win for pipeline
Peephole optimizer given 4 args:
  Code fragment
  Target basic block
  Next basic block
  Branch target block (if any)

Uses several functions:
  Remove basic block (on fragment)
  Remove instruction (on block)
  Replace instruction (on block)
  Insert instruction before i (on block)

Code loops over list of instructions
  Could be IR or assem

Extract
  Current instr and next 1-4 instrs
  First real instr of next block
  First real instr of branch target blk

Look for patterns over these instructions
  Much like tile matching in maximal munch

Edit instructions as needed
  Restart loop after change