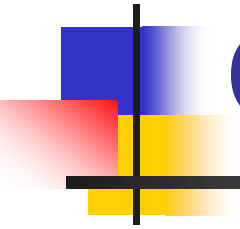
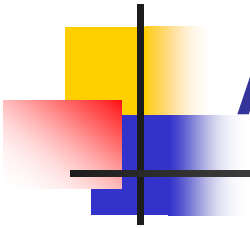


CSE P 501 – Compilers

Analysis & Optimizations & Loops

Hal Perkins

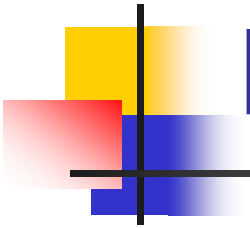
Winter 2008



Agenda

- Dataflow analysis example – live variables
- Loop optimizations
  - Dominators – discovering loops
  - Loop invariant calculations
  - Loop transformations
- A quick look at some memory hierarchy issues

- Largely based on material in Appel ch. 10, 17, 18,21; similar material in other books



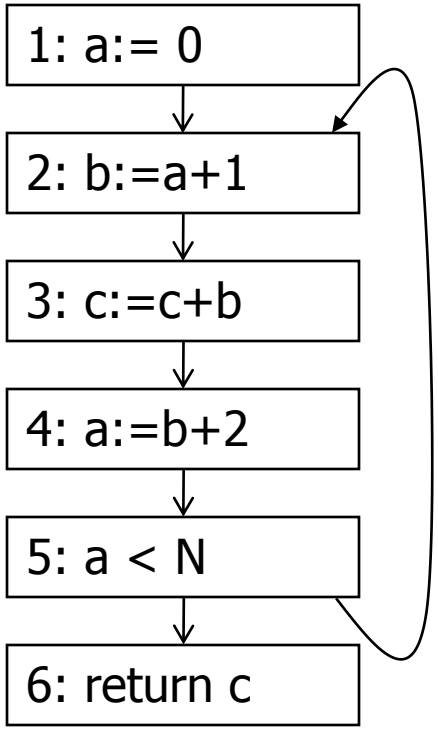
Live Variables

- Recall that two variables can be assigned to the same register if they are not alive at the same time
- ∴ We would like to analyze code to decide when variables are "live"
- Def. A variable is live if it holds a value that may be needed in the future

Example (1 stmt per block)

- Code

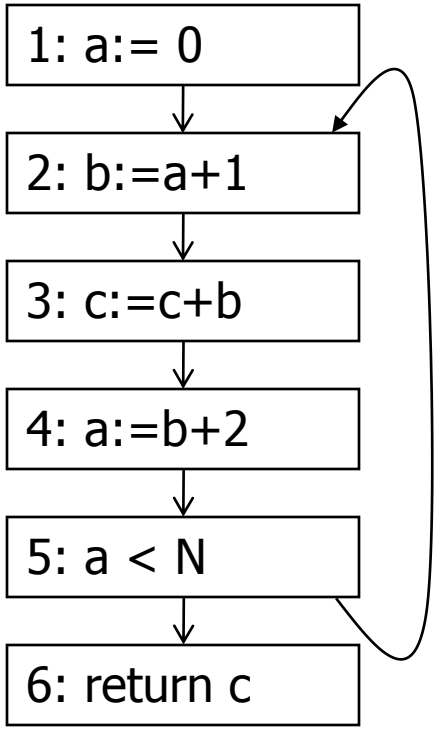
```
    a := 0
L:  b := a+1
    c := c+b
    a := b*2
    if a < N goto L
    return c
```

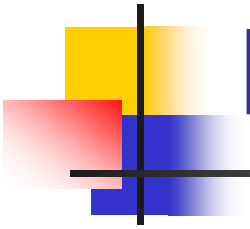


Live Ranges

- b: 2 -> 3, 3 -> 4
- a: 1 -> 2, and
  4 -> 5 -> 2,
  but not 2 -> 3 -> 4
- c: live on entry; live throughout

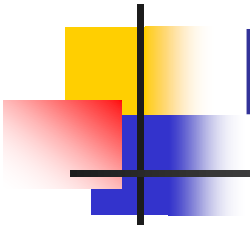
- Liveness analysis flows from the future to the past





Liveness Analysis

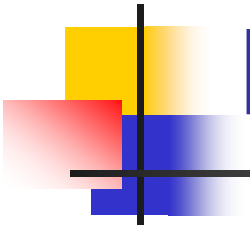
- A variable is *live* on an edge if there is a path from that edge to a use that does not go through any definition
- In a block, a variable is
  - *Live-in* if it is live on any in-edge
  - *Live-out* if it is live on any out-edge



Liveness Analysis Sets

- For each block b
  - use[b] = variable used in b before any def
  - def[b] = variable defined in b & not killed
  - in[b] = variables live on entry to b
  - out[b] = variables live on exit from b

  - Note: slightly different from definitions for same problem in last week's slides



Dataflow equation

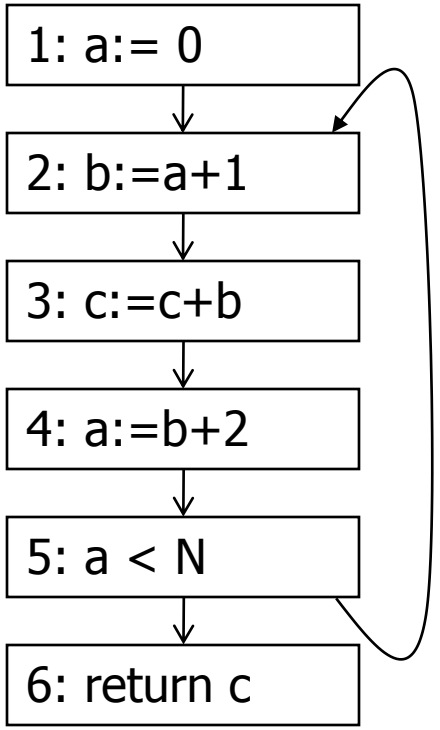
- Given the preceding definitions, we have

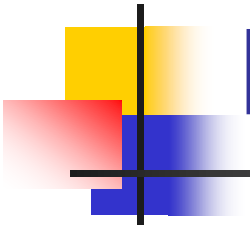
  $in[b] = use[b] \cup (out[b] - def[b])$

  $out[b] = \cup_{s \in succ[b]} in[s]$

- Algorithm
  - Set $in[b] = out[b] = \varnothing$
  - Update in, out until no change
- Evaluation order: back to front is best given information flow

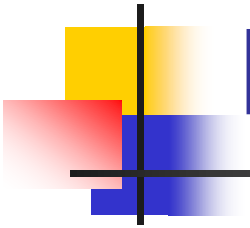
Calculation





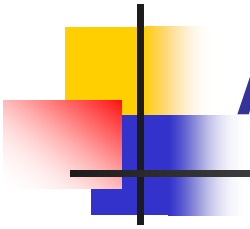
Liveness & Register Allocation

- Liveness information is used to construct an *interference graph*
- If two variables are live at the same point in the program, they *interfere*, and cannot be in the same register
- Graph coloring starts with this information



Liveness in Larger Programs

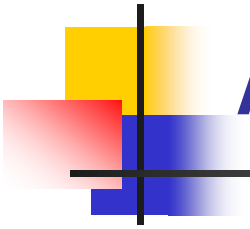
- This example treated individual instructions as nodes
- Only information discovered was where operands were defined and used
- In general:
  - Nodes are basic blocks (faster, smaller problem, less overhead)
  - Most of the time, need to pay attention to semantics of specific operations



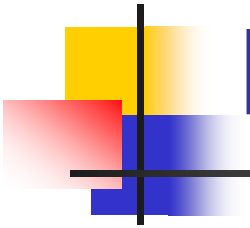
A Few Transformations (1)

- Common subexpression elimination
  - If x op y is calculated at a point where it is available, the recalculation can be dropped
- Constant propagation
  - If t := c and c is a constant, a later use of t can be replaced by c

A Few Transformations (2)

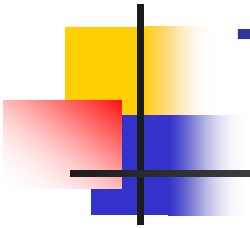


- Copy propagation
  - If t := y and the definition of t is later used where y has not been redefined, can replace t with y
- Dead code elimination
  - If x := exp and x is not live out, then this can be deleted
  - **IF** it doesn't have side effects, and **IF** deleting it doesn't change program semantics



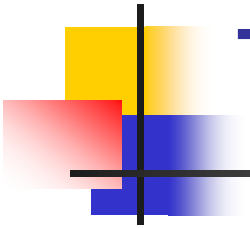
Faster Dataflow Analysis

- Besides working with basic blocks, there are some other ways to speed things up
  - Pick the right order to process blocks
    - Depends on whether information flows forwards or backwards
  - Can calculate def-use chains to link uses to define points (but SSA is the more general way to do this)
  - Use worklist algorithms to recalculate only things that have changed instead of redoing complete analysis



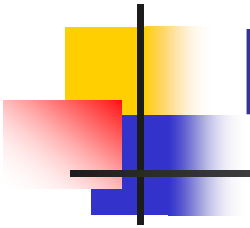
Tranformations & Analysis

- Transformations and optimizations change the information in the analysis
- Examples
  - If a:=b op c is removed as dead code, a previous assignment b:=… (or c:=…) might become dead
  - Eliminating one common subexpression might expose another



Transformation & Analysis

- Brute force strategy
  - Perform global analysis
  - Do as many dataflow-based optimizations as possible
  - Repeat until no more changes found
- But this can require lots of recalculation if transformations cascade badly



Example

- Suppose we discover that this statement is useless

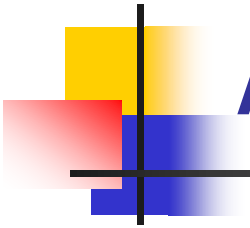
  x := a0 + a1 + a2 + … + an

- What happens if we eliminate it?

What we discovered

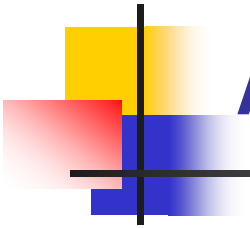
- What we really discovered is that
  x := tnm1 + an
  is dead
- When this is eliminated the next pass discovers that
  tnm1 := … is dead
- etc…

```
t1 := a0 + a1
t2 := t1 + a2
t3 := t2 + a3
…
tnm1 :=
        tnm2 + anm1
x := tnm1 + an
```



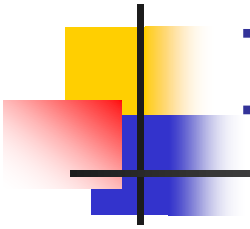
Analysis Strategies (1)

- Cutoff: quit after some small (fixed) number of rounds (e.g., 3)
  - Assumes additional rounds unlikely to do much good
- Do cascading analysis that remains valid after transformations
  - Value numbering is an example of this



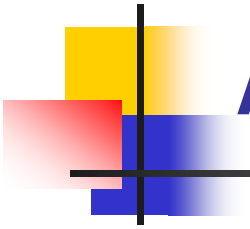
Analysis Strategies (2)

- Incremental dataflow analysis
  - When optimizer changes something, tweak the analysis information to reflect the new situation



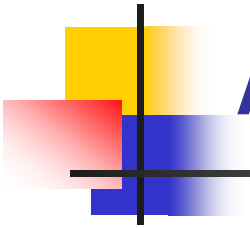
Incremental Liveness Analysis

- Example: suppose we delete a:= b op c
  - a is no longer defined here; if it was live-out, it is now live-in at this point
  - b and c are no longer used here; if they are not live-out at this point, they are no longer live-in



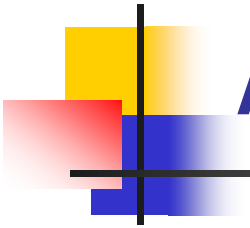
Alias Analysis

- So far we have (mostly) assumed simple variables – i.e., compiler temporaries or local scalar variables
  - Names are unambiguous
  - Behavior of external variables (i.e, memory locations) not analyzed
    - i.e., we store or load from memory without considering how locations may interact



Aliases

- A variable or memory location may have multiple names or *aliases*
  - Call-by-reference parameters
  - Variables whose address is taken (&x)
  - Expressions that dereference pointers (p.x, *p)
  - Expressions involving subscripts (a[i])
  - Variables in nested scopes



Aliases vs Optimizations

- Example:

  p.x := 5; q.x := 7; a := p.x;

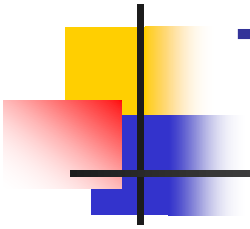
  - Does reaching definition analysis show that the definition of p.x reaches a?
  - (Or: do p and q refer to the same variable?)
  - (Or: *can* p and q refer to the same thing?)

Aliases vs Optimizations

- Example

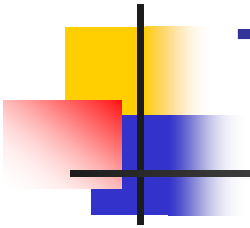
```
void f(int *p, int *q) {
  *p = 1; *q = 2;
  return *p;
}
```

  - How do we account for the possibility that p and q might refer to the same thing?



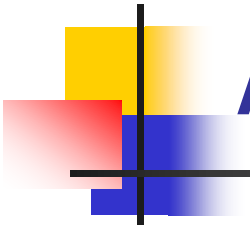
Types and Aliases (1)

- In Java, ML, MiniJava, and others, if two variables have incompatible types they cannot be names for the same location
  - Also helps that programmer cannot create arbitrary pointers to storage in these languages



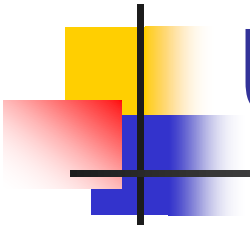
Types and Aliases (2)

- Strategy: Divide memory locations into *alias classes* based on type information (every type, array, record field is a class)
- Implication: need to propagate type information from the semantics pass to optimizer
  - Not normally true of a minimally typed IR
- Items in different alias classes cannot refer to each other



Aliases and Flow Analysis

- Idea: Base alias classes on points where a value is created
  - Every new/malloc and each local or global variable whose address is taken is an alias class
  - Pointers can refer to values in multiple alias classes (so each memory reference is to a set of alias classes)
  - Use to calculate "may alias" information (e.g., p "may alias" q at program point s)



Using "may-alias" information

- Treat each alias class as a "variable" in dataflow analysis problems
- Example: framework for available expressions
  - Given statement s: M[a]:=b,

    gen[s] = { }

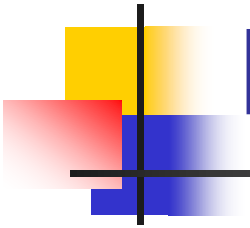
    kill[s] = { M[x] | a may alias x at s }

May-Alias Analysis

- Without alias analysis, #2 kills M[t] since x and t might be related
- If analysis determines that "x may-alias t" is false, M[t] is still available at #3; can eliminate the common subexpression and use copy propagation

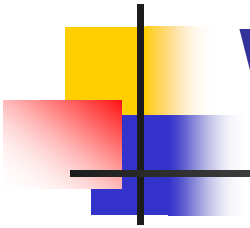
- Code

```
1:  u := M[t]
2:  M[x] := r
3:  w := M[t]
4:  b := u+w
```



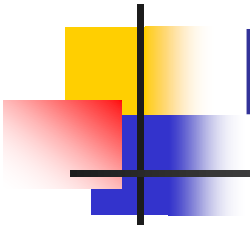
Loops

- Much of he execution time of programs is spent here
- ∴ worth considerable effort to make loops go faster
- ∴ want to figure out how to recognize loops and figure out how to "improve" them



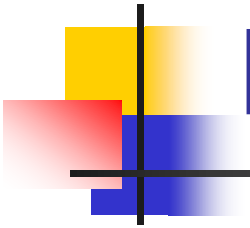
What's a Loop?

- In a control flow graph, a loop is a set of nodes S such that:
  - S includes a *header node* h
  - From any node in S there is a path of directed edges leading to h
  - There is a path from h to any node in S
  - There is no edge from any node outside S to any node in S other than h



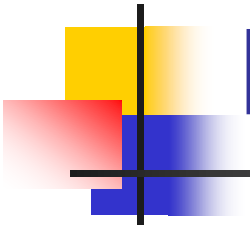
Entries and Exits

- In a loop
  - An *entry node* is one with some predecessor outside the loop
  - An *exit node* is one that has a successor outside the loop
- Corollary of preceding definitions: A loop may have multiple exit nodes, but only one entry node

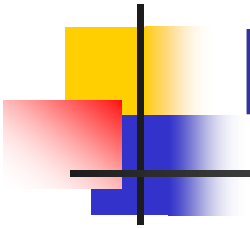


Reducible Flow Graphs

- In a reducible flow graph, any two loops are either nested or disjoint
- Roughly, to discover if a flow graph is reducible, repeatedly delete edges and collapse together pairs of nodes (x,y) where x is the only predecessor of y
- If the graph can be reduced to a single node it is reducible
  - Caution: this is the "powerpoint" version of the definition – see a good compiler book for the careful details

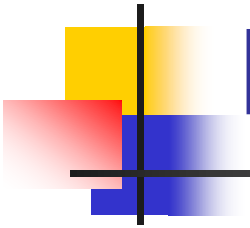


Example: Is this Reducible?

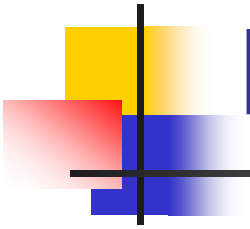


Example: Is this Reducible?

Reducible Flow Graphs in Practice

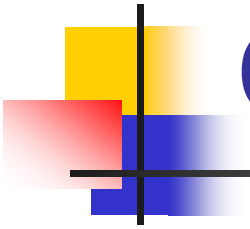


- Common control-flow constructs yield reducible flow graphs
  - if-then[-else], while, do, for, break(!)
- A C function without goto will always be reducible
- Many dataflow analysis algorithms are very efficient on reducible graphs, but
- We don't need to assume reducible control-flow graphs to handle loops



Finding Loops in Flow Graphs

- We use *dominators* for this
- Recall
  - Every control flow graph has a unique start node s0
  - Node x dominates node y if every path from s0 to y must go through x
  - A node x dominates itself

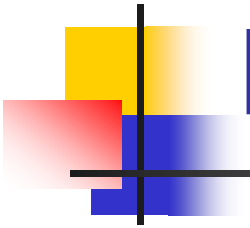


Calculating Dominator Sets

- D[n] is the set of nodes that dominate n
  - D[s0] = { s0 }
  - D[n] = { n } $\cup$ ($\cap_{p \in pred[n]}$ D[p])
- Set up an iterative analysis as usual to solve this
  - Except initially each D[n] must be all nodes in the graph – updates make these sets smaller if changed

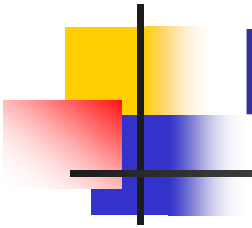
Immediate Dominators

- Every node n has a single *immediate dominator* idom(n)
  - idom(n) differs from n
  - idom(n) dominates n
  - idom(n) does not dominate any other dominator of n
- Fact (er, theorem): If a dominates n and b dominates n, then either a dominates b or b dominates a
  - ∴ idom(n) is unique

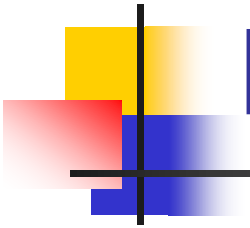


Dominator Tree

- A *dominator tree* is constructed from a flowgraph by drawing an edge form every node in n to idom(n)
  - This will be a tree. Why?

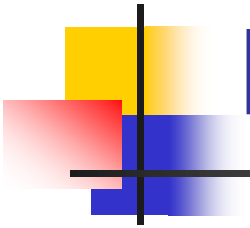


Example



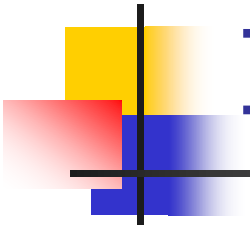
Back Edges & Loops

- A flow graph edge from a node n to a node h that dominates n is a *back edge*
- For every back edge there is a corresponding subgraph of the flow graph that is a loop



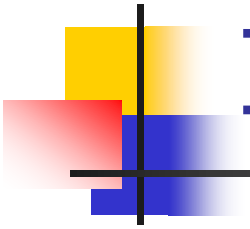
Natural Loops

- If h dominates n and n->h is a back edge, then the *natural loop* of that back edge is the set of nodes x such that
  - h dominates x
  - There is a path from x to n not containing h
- h is the *header* of this loop
- Standard loop optimizations can cope with loops whether they are natural or not



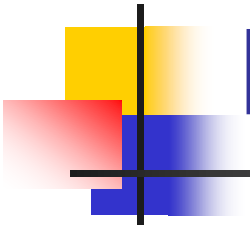
Inner Loops

- Inner loops are more important for optimization because most execution time is expected to be spent there
- If two loops share a header, it is hard to tell which one is "inner"
  - Common way to handle this is to merge natural loops with the same header



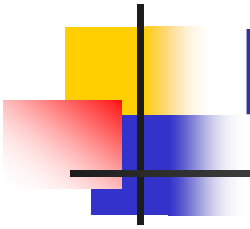
Inner (nested) loops

- Suppose
  - A and B are loops with headers a and b
  - a ≠ b
  - b is in A
- Then
  - The nodes of B are a proper subset of A
  - B is nested in A, or B is the *inner loop*



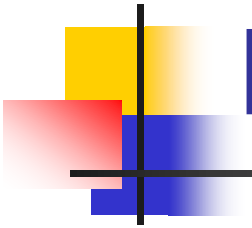
Loop-Nest Tree

- Given a flow graph G
  1. Compute the dominators of G
  2. Construct the dominator tree
  3. Find the natural loops (thus all loop-header nodes)
  4. For each loop header h, merge all natural loops of h into a single loop: loop[h]
  5. Construct a tree of loop headers s.t. h1 is above h2 if h2 is in loop[h1]

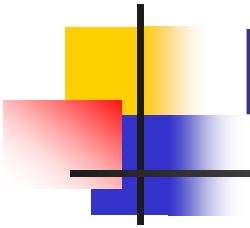


Loop-Nest Tree details

- Leaves of this tree are the innermost loops
- Need to put all non-loop nodes somewhere
  - Convention: lump these into the root of the loop-nest tree

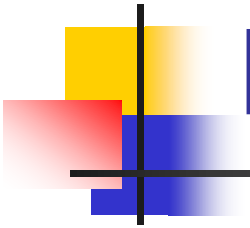


Example



Loop Preheader

- Often we need a place to park code right before the beginning of a loop
- Easy if there is a single node preceding the loop header h
  - But this isn't the case in general
- So insert a *preheader* node p
  - Include an edge p->h
  - Change all edges x->h to be x->p

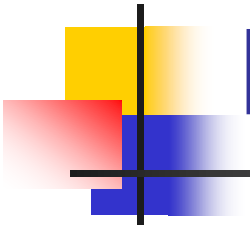


Loop-Invariant Computations

- Idea: If x := a1 op a2 always does the same thing each time around the loop, we'd like to *hoist* it and do it once outside the loop
- But can't always tell if a1 and a2 will have the same value
  - Need a conservative (safe) approximation

Loop-Invariant Computations

- d: x := a1 op a2 is *loop-invariant* if for each ai
  - ai is a constant, or
  - All the definitions of ai that reach d are outside the loop, or
  - Only one definition of ai reaches d, and that definition is loop invariant
- Use this to build an iterative algorithm
  - Base cases: constants and operands defined outside the loop
  - Then: repeatedly find definitions with loop-invariant operands



Hoisting

- Assume that d: x := a1 op a2 is loop invariant. We can hoist it to the loop preheader if
  - d dominates all loop exits where x is live-out, and
  - There is only one definition of x in the loop, and
  - x is not live-out of the loop preheader
- Need to modify this if a1 op a2 could have side effects or raise an exception

Hoisting: Possible?

- Example 1

```
L0: t := 0
L1: i := i + 1
    t := a op b
    M[i] := t
    if i < n goto L1
L2: x := t
```

- Example 2

```
L0: t := 0
L1: if i ≥ n goto L2
    i := i + 1
    t := a op b
    M[i] := t
    goto L1
L2: x := t
```

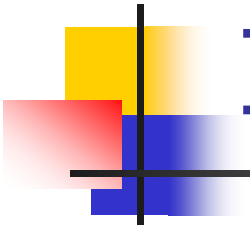
Hoisting: Possible?

- Example 3

```
L0: t := 0
L1: i := i + 1
    t := a op b
    M[i] := t
    t := 0
    M[j] := t
    if i < n goto L1
L2: x := t
```

- Example 4

```
L0: t := 0
L1: M[j] := t
    i := i + 1
    t := a op b
    M[i] := t
    if i < n goto L1
L2: x := t
```



Induction Variables

- Suppose inside a loop
  - Variable i is incremented or decremented
  - Variable j is set ot i*c+d where c and d are loop-invariant
- Then we can calculate j's value without using i
  - Whenever i is incremented by a, increment j by c*a

Example

- Original

```
      s := 0
      i := 0
L1:   if i ≥ n goto L2
      j := i*4
      k := j+a
      x := M[k]
      s := s+x
      i := i+1
      goto L1
L2:
```

- Do
  - Induction-variable analysis to discover i and j are related induction variables
  - Strength reduction to replace *4 with an addition
  - Induction-variable elimination to replace i ≥ n
  - Assorted copy propagation

Result

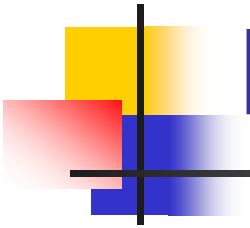
- Original

```
     s := 0
     i := 0
L1:  if i ≥ n goto L2
     j := i*4
     k := j+a
     x := M[k]
     s := s+x
     i := i+1
     goto L1
L2:
```

- Transformed

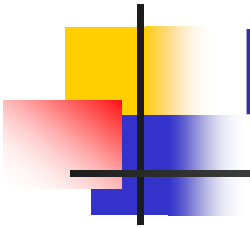
```
     s := 0
     k' = a
     b = n*4
     c = a+b
L1:  if k' ≥ c goto L2
     x := M[k']
     s := s+x
     k' := k'+4
     goto L1
L2:
```

Details are somewhat messy – see your favorite compiler book



Loop Unrolling

- If the body of a loop is small, most of the time is spent in the "increment and test" code
- Idea: reduce overhead by *unrolling* – put two or more copies of the loop body inside the loop



Loop Unrolling

- Basic idea: Given loop L with header node h and back edges si->h
  1. Copy the nodes to make loop L' with header h' and back edges si'->h'
  2. Change all backedges in L from si->h to si->h'
  3. Change all back edges in L' from si'->h' to si'->h

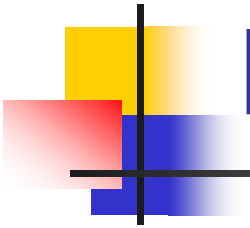
Unrolling Algorithm Results

- Before

```
L1: x := M[i]
    s := s + x
    i := i + 4
   if i<n goto L1 else L2
L2:
```

- After

```
L1: x := M[i]
    s := s + x
    i := i + 4
    if i<n goto L1' else L2
L1':x := M[i]
    s := s + x
    i := i + 4
   if i<n goto L1 else L2
L2:
```



Hmmmm....

- Not so great – just code bloat
- But: use induction variables and various loop transformations to clean up

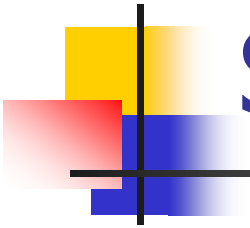
After Some Optimizations

- Before

```
L1: x := M[i]
    s := s + x
    i := i + 4
    if i<n goto L1' else L2
L1':x := M[i]
    s := s + x
    i := i + 4
   if i<n goto L1 else L2
L2:
```

- After

```
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
    if i<n goto L1 else L2
L2:
```



Still Broken

- But in a different, better(?) way
- Good code, but only correct if original number of loop iterations was even
- Fix: add an epilogue to handle the "odd" leftover iteration

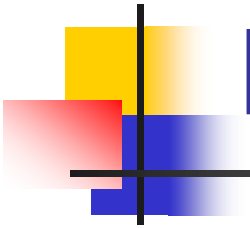
Fixed

- Before

```
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
  if i<n goto L1 else L2
L2:
```

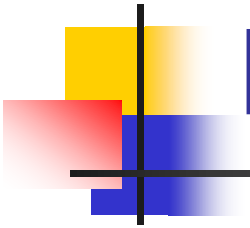
- After

```
    if i<n-8 goto L1 else L2
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
    if i<n-8 goto L1 else L2
L2: x := M[i]
    s := s+x
    i := i+4
    if i < n goto L2 else L3
L3:
```



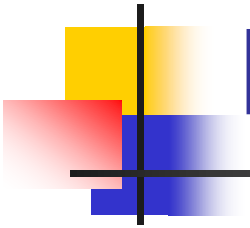
Postscript

- This example only unrolls the loop by a factor of 2
- More typically, unroll by a factor of K
  - Then need an epilogue that is a loop like the original that iterates up to K-1 times



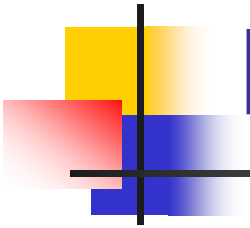
Memory Heirarchies

- One of the great triumphs of computer design
- Effect is a large, fast memory
- Reality is a series of progressively larger, slower, cheaper stores, with frequently accessed data automatically staged to faster storage (cache, main storage, disk)
- Programmer/compiler typically treats it as one large store. Bug or feature?



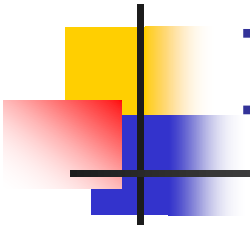
Memory Issues

- Byte load/store is often slower than whole (physical) word load/store
  - Unaligned access is often extremely slow
- Temporal locality: accesses to recently accessed data will usually find it in the (fast) cache
- Spatial locality: accesses to data near recently used data will usually be fast
  - "near" = in the same cache block
- But – accesses to blocks that map to the same cache block will cause thrashing



Data Alignment

- Data objects (structs) often are similar in size to a cache block (≈ 8 words)
  - ∴ Better if objects don't span blocks
- Some strategies
  - Allocate objects sequentially; bump to next block boundary if useful
  - Allocate objects of same common size in separate pools (all size-2, size-4, etc.)
- Tradeoff: speed for some wasted space



Instruction Alignment

- Align frequently executed basic blocks on cache boundaries (or avoid spanning cache blocks)
- Branch targets (particularly loops) may be faster if they start on a cache line boundary
- Try to move infrequent code (startup, exceptions) away from hot code
- Optimizing compiler should have a basic-block ordering phase (& maybe even loader)

Loop Interchange

- Watch for bad cache patterns in inner loops; rearrange if possible
- Example

```
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < p; k++)
      a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
```

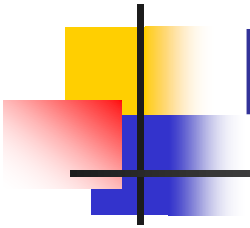
  - b[i,j+1,k] is reused in the next two iterations, but will have been flushed from the cache by the k loop

Loop Interchange

- Solution for this example: interchange j and k loops

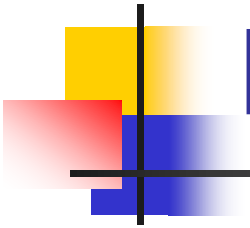
```
for (i = 0; i < m; i++)
  for (k = 0; k < p; k++)
    for (j = 0; j < n; j++)
      a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
```

  - Now b[i,j+1,k] will be used three times on each cache load
  - Safe because loop iterations are independent



Loop Interchange

- Need to construct a data-dependency graph showing information flow between loop iterations
- For example, iteration (j,k) depends on iteration (j’,k’) if (j’,k’) computes values used in (j,k) or stores values overwritten by (j,k)
  - If there is a dependency and loops are interchanged, we could get different results – so can’t do it

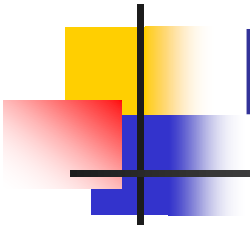


Blocking

- Consider matrix multiply

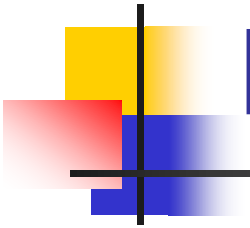
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    c[i,j] = 0.0;
    for (k = 0; k < n; k++)
      c[i,j] = c[i,j] + a[i,k]*b[k,j]
  }
```

- If A, B fit in the cache together, great!
- If they don't, then every b[k,j] reference will be a cache miss
- Loop interchange (i<->j) won't help; then every a[i,k] reference would be a miss



Blocking

- Solution: reuse rows of A and columns of B while they are still in the cache
- Assume the cache can hold 2*c*n matrix elements (1 < c < n)
- Calculate c $\times$ c blocks of C using c rows of A and c columns of B



Blocking

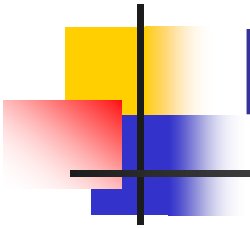
- Calculating c × c blocks of C

```
for (i = i0; i < i0+c; i++)
  for (j = j0; j < j0+c; j++) {
    c[i,j] = 0.0;
    for (k = 0; k < n; k++)
      c[i,j] = c[i,j] + a[i,k]*b[k,j]
  }
```

Blocking

- Then nest this inside loops that calculate successive c × c blocks

```
for (i0 = 0; i0 < n; i0+=c)
  for (j0 = 0; j0 < n; j0+=c)
    for (i = i0; i < i0+c; i++)
      for (j = j0; j < j0+c; j++) {
        c[i,j] = 0.0;
        for (k = 0; k < n; k++)
          c[i,j] = c[i,j] + a[i,k]*b[k,j]
      }
```



Parallelizing Code

- This is only a taste of how we can rearrange loops. There is a whole class of transformations that attempt to discover loops that can be vectorized or done in parallel, which we won't get to. Look for more in the Dragon Book or other sources.