CS 701

Charles N. Fischer

Fall 2003

http://www.cs.wisc.edu/~fischer/cs701.html

Class Meets

Tuesdays & Thursdays, 11:00 — 12:15
2321 Engineering Hall

Instructor

Charles N. Fischer
5397 Computer Sciences
Telephone: 262-6635
E-mail: fischer@cs.wisc.edu
Office Hours:
10:30 - Noon, Mondays & Wednesdays, or by appointment

Teaching Assistant

Kent Hunter
1308 Computer Sciences
Telephone: 262-6602
E-mail: khunter@cs.wisc.edu
Office Hours:
10:00 - 11:00
Tuesdays and Thursdays,
or by appointment

Key Dates

- September 23: Project 1 due
- October 21: Project 2 due (tentative)
- October 23: Midterm (tentative)
- November 18: Project 3 due (tentative)
- December 11: Project 4 due
- December ??: Final Exam, date to be determined

Class Text

There is no required text.

Handouts and Web-based reading will be used.

Suggested reference:

Advanced Compiler Design & Implementation,
by Steven S. Muchnick,
published by Morgan Kaufman.

Instructional Computers

Departmental SPARC Processors (nova1-nova60)

You may use your own workstation if it is has a SPARC processor
(test using **`dmesg|grep cpu`**)

Otherwise log onto a SPARC processor to do SPARC-specific assignments

CS701 Projects

1. SPARC Code Optimization
2. Global Register Allocation
   (using Graph Coloring)
3. Global Code Optimizations
4. Individual Research Topics

Academic Misconduct Policy

- You must do your assignments—no copying or sharing of solutions
- You may discuss general concepts and Ideas
- All cases of Misconduct *must* be reported.
- Penalties may be severe.

Reading Assignment

- Get Handout #2 (Chapter 15, Code Optimization) from DoIt.
- Read Chapters 0-6 and Appendices G&H of the SPARC Architecture Manual. Also skim Appendix A.
- Read section 15.2 of Chapter 15.
- Read Assignment #1

Overview of Course Topics

1. Register Allocation

Local Allocation

Avoid unnecessary loads and stores within a *basic block*. Remember and reuse register contents.
Consider effects of *aliasing*.

Global Allocation

Allocate registers within a single subprogram. Choose "most profitable" values. Map several values to the *same* register.

Interprocedural Allocation

Avoid saves and restores across calls. Share globals in registers.

2. Code Scheduling

We can reorder code to reduce latencies and to maximize ILP (*Instruction Level Parallelism*). We must respect *data dependencies* and *control dependencies*.

```
ld   [a],%r1      ld [a],%r1
add %r1,1,%r2     mov 3,%r3
mov 3,%r3         add %r1,1,%r2
```

(before) (after)

3. Automatic Instruction Selection

How do we map an IR (*Intermediate Representation*) into Machine Instructions?

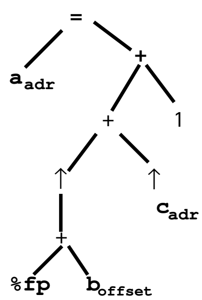
Can we guarantee the *best* instruction sequence?

Idea—Match instruction patterns (represented as trees) against an IR that is a low-level tree. Each match is a generated instruction; the best overall match is the best instruction sequence.

Example:

```
a=b+c+1;
```

In IR tree form:



Generated code:

```
ld   [%fp+b_offset],%r1
ld   [c_adr],%r2
add  %r1,%r2,%r3
add  %r3,1,%r4
st   %r4,[a_adr]
```

Why use four *different* registers?

4. Peephole Optimization

Inspect generated code sequences and replace pairs/triples/tuples with better alternatives.

(before)	(after)
`ld [a],%r1` `mov const,%r2` `add %r1,%r2,%r3`	`ld [a],%r1` `add %r1,const,%r3`
`mov 0,%r1` `OP %r1,%r2,%r3`	`OP %g0,%r2,%r3`

But why not just generate the better code sequence to begin with?

5. Cache Improvements

We want to access data & instructions from the L1 cache whenever possible; misses into the L2 cache (or memory) are *expensive*!

We will layout data and program code with consideration of cache sizes and access properties.

6. Local & Global Optimizations

Identify unneeded or redundant code.

Decide where to place code.

Worry about debugging issues (how reliable are current values and source line numbers after optimization?)

7. Program representations

- Control Flow Graphs
- Program Dependency Graphs
- Static Single Assignment Form (SSA)

Each program variable is assigned to in only *one* place.

After an assignment $x_i = y_j$, the relation $x_i = y_j$ *always* **holds.**

Example:

`if (a)` `    x = 1` `else x = 2;` `print(x)`	`if (a)` `    x1 =1` `else x2 =2;` `x3 = φ(x1,x2)` `print(x3)`

8. Data Flow Analysis

Determine invariant properties of subprograms; analysis can be extended to entire programs.

Model abstract execution.

Prove correctness and efficiency properties of analysis algorithms.

Review of Compiler Optimizations

1. Redundant Expression Elimination (Common Subexpression Elimination)

Use an address or value that has been previously computed. Consider control and data dependencies.

2. Partially Redundant Expression (PRE) Elimination

A variant of Redundant Expression Elimination. If a value or address is redundant along *some* execution paths, add computations to other paths to create a fully redundant expression (which is then removed).

Example:

```
if (i > j)
   a[i] = a[j];
a[i] = a[i] * 2;
```


3. Constant Propagation

If a variable is known to contain a particular constant value at a particular point in the program, replace references to the variable at that point with that constant value.

4. Copy Propagation

After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable (until one or the other of the variables is reassigned).

(This may also “set up” dead code elimination. Why?)

5. Constant Folding

An expression involving constant (literal) values may be evaluated and simplified to a constant result value. Useful when constant propagation is performed.

6. Dead Code Elimination

Expressions or statements whose values or effects are unused may be eliminated.

7. Loop Invariant Code Motion

An expression that is *invariant* in a loop may be moved to the loop's header, evaluated once, and reused within the loop.

Safety and *profitability* issues may be involved.

8. Scalarization (Scalar Replacement)

A field of a structure or an element of an array that is repeatedly read or written may be copied to a local variable, accessed using the local, and later (if necessary) copied back.

This optimization allows the local variable (and in effect the field or array component) to be allocated to a register.

9. Local Register Allocation

Within a *basic block* (a straight line sequence of code) track register contents and reuse variables and constants from registers.

10. Global Register Allocation

Within a subprogram, frequently accessed variables and constants are allocated to registers. Usually there are *many more* register candidates than available registers.

11. Interprocedural Register Allocation

Variables and constants accessed by more than one subprogram are allocated to registers. This can *greatly* reduce call/return overhead.

12. Register Targeting

Compute values directly into the intended target register.

13. Interprocedural Code Motion

Move instructions across subprogram boundaries.

14. Call Inlining

At the site of a call, insert the body of a subprogram, with actual parameters initializing formal parameters.

15. Code Hoisting and Sinking

If the same code sequence appears in two or more alternative execution paths, the code may be *hoisted* to a common ancestor or *sunk* to a common successor. (This reduces code size, but does not reduce instruction count.)

16. Loop Unrolling

Replace a loop body executed N times with an expanded loop body consisting of M copies of the loop body. This expanded loop body is executed N/M times, reducing loop overhead and increasing optimization possibilities within the expanded loop body.

17. Software Pipelining

A value needed in iteration *i* of a loop is computed during iteration *i-1* (or *i-2*, ...). This allows long latency operations (floating point divides and square roots, low hit-ratio loads) to execute in parallel with other operations. Software pipelining is sometimes called *symbolic loop unrolling*.

18. Strength Reduction

Replace an expensive instruction with an equivalent but cheaper alternative. For example a division may be replaced by multiplication of a reciprocal, or a list append may be replaced by cons operations.

19. Data Cache Optimizations

- Locality Optimizations

  Cluster accesses of data values both spacially (within a cache line) and temporally (for repeated use). *Loop interchange* and *loop tiling* improve temporal locality.

- Conflict Optimizations

  Adjust data locations so that data used consecutively and repeatedly don't share the same cache location.

20. Instruction Cache Optimizations

Instructions that are repeatedly re-executed should be accessed from the instruction cache rather than the secondary cache or memory. Loops and "hot" instruction sequences should fit within the cache.

Temporally close instruction sequences should not map to conflicting cache locations.

Reading Assignment

- Read "Modern Microprocessors—A 90 Minute Guide!," by Jason Patterson.

SPARC Overview

- SPARC is an acronym for
  Scalable Processor ARChitecture
- SPARCs are load/store RISC processors

  Load/store means only loads and stores access memory directly.

  RISC (Reduced Instruction Set Computer) means the architecture is simplified with a limited number of instruction formats and addressing modes.

- Instruction format:

  `add %r1,%r2,%r3`

  Registers are prefixed with a %

  Result is stored into last operand.

  `ld [adr],%r1`

  Memory addresses (used only in loads and stores) are enclosed in brackets
- Distinctive features include *Register Windows* and *Delayed Branches*

Register Windows

The SPARC provides 32 general-purpose integer registers, denoted as **`%r0`** through **`%r31`**.

These 32 registers are subdivided into 4 groups:

Globals:	**`%g0`** to **`%g7`**
In registers:	**`%i0`** to **`%i7`**
Locals:	**`%l0`** to **`%l7`**
Out registers:	**`%o0`** to **`%o7`**

There are also 32 floating-point registers, **`%f0`** to **`%f31`**.

A SPARC processor has an implementation-dependent number of *register windows*, each consisting of 16 distinct registers.

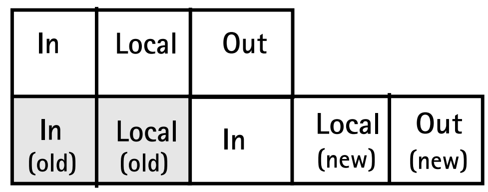
The "in", "local" and "out" registers that are accessed in a procedure depend on the current register window. The "global"

registers are independent of the register windows (as are the floating-point registers).

A register window may be pushed or popped using SPARC **`save`** and **`restore`** instructions.

After a register window push, the "out" registers become "in" registers and a fresh set of "local" and "out" registers is created:

Before **`save`**:



After **`save`**

Why the overlap between "in" and "out" registers? It's a convenient way to pass parameters—the caller puts parameter values in his "out" registers. After a call (and a **`save`**) these values are *automatically* available as "in" registers in the newly created register window.

SPARC procedure calls normally advance the register window. The "in" and "local" registers become hidden, and the "out" registers become the "in" registers of the called procedure, and new "local" and "out" registers become available.

A register window is advanced using the **`save`** instruction, and rolled back using the **`restore`** instruction. These instructions are separate from the procedure **`call`** and **`return`** instructions, and can sometimes be optimized away.

For example, a *leaf procedure*—one that contains no calls—can be compiled without use of **`save`** and **`restore`** if it doesn't need too many registers. The leaf procedure must then make do with the caller's registers, modifying only those the caller treats as volatile.

Register Conventions

Global Registers

`%g0` is unique: It *always* contains 0 and can *never* be changed.

`%g1` to **`%g7`** have global scope (they are unaffected by **`save`** and **`restore`** instructions)

`%g1` to **`%g4`** are volatile across calls; they may be used between calls.

`%g5` to **`%g7`** are reserved for special use by the SPARC ABI (application binary interface)

Local Registers

`%l0` to **`%l7`**

May be freely used; they are unaffected by deeper calls.

In Registers

These are also the caller's out registers; they are unaffected by deeper calls.

`%i0`

Contains incoming parameter 1.

Also used to return function value to caller.

`%i1` to **`%i5`**

Contain incoming parameters 2 to 6 (if needed); freely usable otherwise.

`%i6` (also denoted as **`%fp`**)

Contains frame pointer (stack pointer of caller); it must be preserved.

`%i7`

Contains return address -8 (offset due to delay slot); it must be preserved.

Out Registers

Become the in registers for procedures called from the current procedure.

`%o0`

Contains outgoing parameter 1.

It also contains the value returned by the called procedure.

It is volatile across calls; otherwise it is freely usable.

`%o1` to **`%o5`**

Contain outgoing parameters 2 to 6 as needed.

These are volatile across calls; otherwise they are freely usable.

`%o6` (also denoted as **`%sp`**)

Holds the stack pointer (and becomes frame pointer of called routines)

It is reserved; it must *always* be valid (since TRAPs may modify the stack at any time).

`%o7`

Is volatile across calls.

It is loaded with address of caller on a procedure call.

Special SPARC Instructions

`save %r1,%r2,%r3`

`save %r1,const,%r3`

This instruction pushes a register window *and* does an add instruction (`%r3 = %r1+%r2`). Moreover, the operands (`%r1` and `%r2`) are from the *old* register window, while the result (`%r3`) is in the *new* window.

Why such an odd definition?

It's ideal to allocate a new register window *and* push a new frame.

In particular,

```
save %sp,-frameSize,%sp
```

pushes a new register window. It also adds `-frameSize` (the stack grows downward) to the old stack pointer, initializing the new stack pointer. (The old stack pointer becomes the current frame pointer)

`restore %r1,%r2,%r3`

`restore %r1,const,%r3`

This instruction pops a register window *and* does an add instruction (`%r3 = %r1+%r2`). Moreover, the operands (`%r1` and `%r2`) are from the *current* register window, while the result (`%r3`) is in the *old* window.

Again, why such an odd definition?

It's ideal to release a register window *and* place a result in the return register (`%o0`).

In particular,

```
restore %r1,0,%o0
```

pops a register window. It also moves the contents of `%r1` to `%o0` (in the caller's register window).

`call label`

This instruction branches to `label` *and* puts the address of the call into register `%o7` (which will become `%i7` after a `save` is done).

`ret`

This instruction returns from a subprogram by branching to `%i7+8`. Why *8 bytes* after the address of the call? SPARC processors have *delayed branch* instructions, so the instruction immediately after a branch (or a `call`) is executed *before* the branch occurs! Thus two instructions after the call is the normal return point.

`mov const,%r1`

You can load a small constant (13 bits or less) into a register using a `mov`. (`mov` is actually implemented as an `or` of `const` with `%g0`).

But how do you load a 32 bit constant? One instruction (32 bits long) isn't enough. Instead you use:

```
sethi %hi(const),%r1
or %r1,%lo(const),%r1
```

That is, you extract the high order 22 bits of `const` (using `%hi`, an assembler operation). `sethi` fills in these 22 bits into `%r1`, clearing the lowest 10 bits. Then `%lo` extracts the 10 low order bits of const, which are or-ed into `%r1`.

Loading a 64 bit constant (in SPARC V9, which is a 64 bit processor) is far nastier:

```
sethi  %uhi(const),%r_tmp
or     %r_tmp,%ulo(const),%r_tmp
sllx   %r_tmp,32,%r_tmp
sethi  %hi(const),%r
or     %r,%lo(const),%r
or     %r_tmp,%r,%r
```


Delayed Branches

In the SPARC, transfers of control (branches, calls and returns) are *delayed*. This means the instruction *after* the branch (or call or return) is executed *before* the transfer of control.

For example, in SPARC code you often see

```
ret
restore
```

The register window restore occurs first, then a return to the caller occurs.

Another example is

```
call subr
mov 3,%o0
```

The load of **subr**'s parameter is placed after the call to **subr**. But the **mov** is done before **subr** is actually called.

Why are Delayed Branches Part of the SPARC Architecture?

Because of pipelining, several instructions are partially completed before a branch instruction can take effect. Rather than lose the computations already done, one (or more!) partially completed instructions can be allowed to complete before a branch takes effect.

How does a Compiler Exploit Delayed Branches?

A peephole optimizer or code scheduler looks for an instruction logically before the branch that can be placed in the branch's *delay slot*. The instruction should not affect a conditional branch's branch decision.

(before)	(after)
`mov  3,%o0` `call subr` `nop`	`call subr` `mov  3,%o0`

Another possibility is to “hoist” the target instruction of a branch into the branch’s delay slot.

```
    call  subr        call  subr+4
    nop               mov  100,%l1
    ...               ...
 subr:             subr:
    mov  100,%l1      mov  100,%l1
```

(before) (after)

Hoisting branch targets doesn’t work for conditional branches—we don’t want to move an instruction that is executed *sometimes* (when the branch is taken) to a position where it is *always* executed (the delay slot).

Annulled Branches

An *annulled branch* (denoted by a “**,a**” suffix) executes the instruction in the delay slot *if* the branch is taken, but *ignores* the instruction in the delay slot if the branch isn't taken.

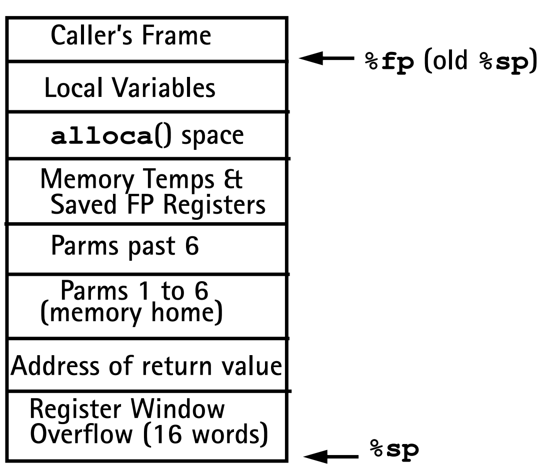
With an annulled branch, a target of a conditional branch can be hoisted into the branch’s delay slot.

```
    bz   else         bz,a  else+4
    nop               mov  100,%l1
  ! then code       ! then code
    ...               ...
 else:             else:
    mov  100,%l1      mov  100,%l1
```

(before) (after)

SPARC Frame Layout (on Run-Time Stack)

The Stack Grows Downward



Minimum frame Size (in gcc) is 112 bytes! (16+1+6+4 words, double aligned)

Examples of SPARC Code

```
int incr(int i){
   return i+1; }
```

Unoptimized:

```
incr:
     save    %sp, -112, %sp
     st      %i0, [%fp+68]
     ld      [%fp+68], %o1
     add     %o1, 1, %o0
     mov     %o0, %i0
     b       .LL2
          nop
.LL2:
     ret
     restore
```

```
int   main(){
   int a;
   return incr(a);}
```

Unoptimized:

```
main:
   save    %sp, -120, %sp
   ld      [%fp-20], %o0
   call    incr, 0
   nop
   mov     %o0, %i0
   b       .LL3
   nop
.LL3:
   ret
   restore
```



```
int incr(int i){
   return i+1; }
```

Optimized:

```
incr:
    retl
    add   %o0, 1, %o0
```

```
int   main(){
   int a;
   return incr(a);}
```

Optimized:

```
main:
   save    %sp, -112, %sp
! Where is a ????
   call    incr, 0
   nop
   ret
   restore %g0, %o0, %o0
```


With More Extensive Optimization (including inlining) we get:

```
incr:
   retl
   add     %o0, 1, %o0

main:
   retl
   add     %o0, 1, %o0
```


Reading Assignment

S. Kurlander, T. Proebsting and C. Fischer, “Efficient Instruction Scheduling for Delayed-Load Architectures,” *ACM Transactions on Programming Languages and Systems*, 1995. (Linked from class Web page)

“On the Fly” Local Register Allocation

Allocate registers as needed during code generation.

Partition registers into 3 classes.

- Allocatable

  Explicitly allocated and freed; used to hold a variable, literal or temporary.

  On SPARC: Local registers & unused In registers.

- Reserved

  Reserved for specific purposes by OS or software conventions.

  On SPARC: **`%fp`**, **`%sp`**, return address register, argument registers, return value register.

- Work

  Volatile—used in short code sequences that need to use a register.

  On SPARC: **`%g1`** to **`%g4`**, unused out registers.

Register Targeting

Allow “end user” of a value to state a register preference in AST or IR.

or

Use Peephole Optimization to eliminate unnecessary register moves.

or

Use *preferencing* in a graph coloring register allocator.

Register Tracking

Improve upon standard getReg/ freeReg allocator by *tracking* (remembering) register contents.

Remember the value(s) currently held within a register; store information in a *Register Association List.*

Mark each value as *Saved* (in memory) or *Unsaved* (in memory).

Each value in a register has a *Cost*. This is the cost (in instructions) to restore the value to a register.

The cost of allocating a register is the sum of the costs of the values it holds.

$$\text{Cost(register)} = \sum_{\text{values} \in \text{register}} \text{cost(values)}$$

When we allocate a register, we will choose the *cheapest* one.

If 2 registers have the same cost, we choose that register whose values have the *most distant* next use.

(Why most distant?)

Costs for the SPARC

0	Dead Value
1	Saved Local Variable
1	Small Literal Value (13 bits)
2	Saved Global Variable
2	Large Literal Value (32 bits)
2	Unsaved Local Variable
4	Unsaved Global Variable

Register Tracking Allocator

```
reg getReg() {
  if ( ∃ r ∈ regSet and cost(r) == 0)
      choose(r)
  else {
      c = 1;
      while(true) {
        if ( ∃ r ∈ regSet and cost(r) == c){
            choose r with cost(r) == c and
                  most distant next use of
                  associated values;
            break;
         }
         c++;
      }
      Save contents of r as necessary;
  }
  return r;
}
```


- Once a value becomes dead, it may be purged from the register association list without any saves.
- Values no longer used, but unsaved, can be purged (and saved) at *zero* cost.
- Assignments of a register to a simple variable may be *delayed*—just add the variable to the Register's Association List entry as unsaved.

  The assignment may be done later or made *unnecessary* (by a later assignment to the variable)
- At the end of a basic block all unsaved values are stored into memory.

Example

```
int a,b,c,d; // Globals
a = 5;
b = a + d;
c = b - 7;
b = 10;
```

Naive Code

```
mov   5,%l0
st    %l0,[a]
ld    [a],%l0
ld    [d],%l1
add   %l0,%l1,%l0
st    %l0,[b]
ld    [b],%l0
sub   %l0,7,%l0
st    %l0,[c]
mov   10,%l0
st    %l0,[b]
```

18 instructions are needed (memory references take 2 instructions)

With Register Tracking

Instruction Generated	%l0	%l1
`mov 5,%l0`	5(S)	
`! Defer assignment to a`	5(S), a(U)	
`ld [d], %l1`	5(S), a(U)	d(S)
`!d unused after next inst`		
`add %l0,%l1,%l1`	5(S), a(U)	b(U)
`!b is dead after next inst`		
`sub %l1,7,%l1`	5(S), a(U)	c(U)
`! %l1 has lower cost`		
`st %l1, [c]`	5(S), a(U)	
`mov 10, %l1`	5(S), a(U)	b(U), 10(S)
`! save unsaved values`		
`st %l0, [a]`		b(U), 10(S)
`st %l1,[b]`		

12 instructions (rather than 18)

Pointers, Arrays and Reference Parameters

When an array, reference parameter or pointed-to variable is read, all unsaved register values that might be aliased must be *stored*.

When an array, reference parameter or pointed-to variable is written, all unsaved register values that might be aliased must be *stored*, then *cleared* from the register association list.

Thus if **`a[3]`** is in a register and **`a[i]`** is assigned to, **`a[3]`** must be stored (if unsaved) and removed from the association list.

Optimal Expression Tree Translation—Sethi-Ullman Algorithm

Reference: R. Sethi & J. D. Ullman, "The generation of optimal code for arithmetic expressions," Journal of the ACM, 1970.

Goal: Translate an expression tree using the *fewest* possible registers.

Approach: Mark each tree node, N, with an *Estimate* of the minimum number of registers needed to translate the tree rooted by N.

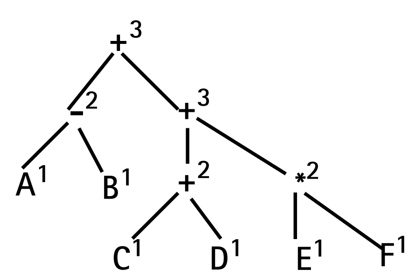
Let RN(N) denote the Register Needs of node N.

In a Load/Store architecture (ignoring immediate operands):

RN(leaf) = 1

RN(Op) =
    If RN(Left) = RN(Right)
        Then RN(Left) + 1
        Else Max(RN(Left), RN(Right))

Example:



Key Insight of SU Algorithm

Translate subtree that needs more registers *first*.

Why?

After translating one subtree, we'll need a register to hold its value.

If we translate the more complex subtree first, we'll still have enough registers to translate the less complex expression (without *spilling* register values into memory).

Specification of SU Algorithm

TreeCG(tree *T, regList RL);

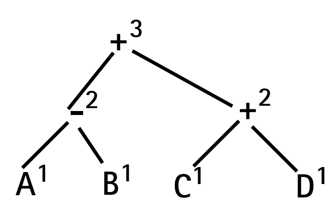
Operation:

- Translate expression tree T using only registers in RL.
- RL must contain at least 2 registers.
- Result of T will be computed into head(RL).

Summary of SU Algorithm

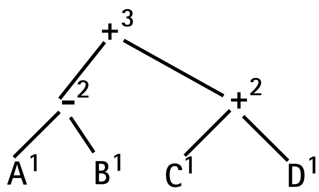
```
if T is a node (variable or literal)
     load T into R1 = head(RL)
else  (T is a binary operator)
   Let R1 = head(RL)
   Let R2 = second(RL)
   if  RN(T.left) >= Size(RL) and
       RN(T.right) >= Size(RL)
       (A spill is unavoidable)
        TreeCG(T.left, RL)
        Store R1 into a memory temp
        TreeCG(T.right, RL)
        Load memory temp into R2
        Generate (OP R2,R1,R1)
  elsif RN(T.left) >= RN(T.right)
        TreeCG(T.left, RL)
        TreeCG(T.right, tail(RL))
        Generate (OP R1,R2,R1)
  else
        TreeCG(T.right, RL)
        TreeCG(T.left, tail(RL))
        Generate (OP R2,R1,R1)
```


Example (with Spilling)



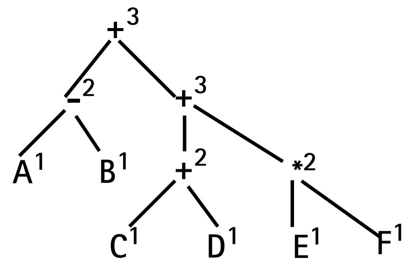
Assume only 2 Registers;
RL = [`%l0`,`%l1`]

We Translate the left subtree first (using 2 registers), store its result into memory, translate the right subtree, reload the left subtree's value, then do the final operation.



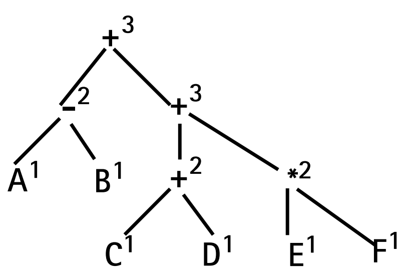
```
ld  [A], %l0
ld  [B], %l1
sub %l0,%l1,%l0
st  %l0, [temp]
ld  [C], %l0
ld  [D], %l1
add %l0,%l1,%l0
ld  [temp], %l1
add %l1,%l0,%l0
```


Larger Example



Assume 3 Registers;
RL = [%l0,%l1,%l2]

Since right subtree is more complex, it is translated first.



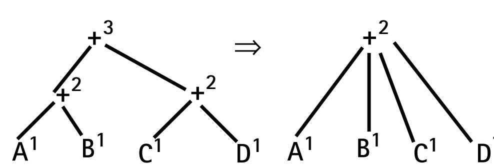
```
ld  [C], %l0
ld  [D], %l1
add %l0,%l1,%l0
ld  [E], %l1
ld  [F], %l2
mul %l1,%l2,%l1
add %l0,%l1,%l0
ld  [A], %l1
ld  [B], %l2
sub %l1,%l2,%l1
add %l1,%l0,%l0
```


Refinements & Improvements

- Register needs rules can be modified to model various architectural features.

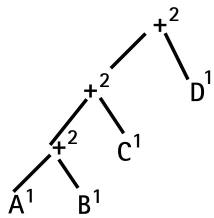
  For example, Immediate operands, that need not be loaded into registers, can be modeled by the following rule:

  RN(literal) = 0 if literal may be used as an immediate operand
- Commutativity & Associativity of operands may be exploited:



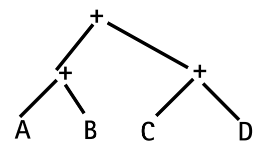
Is Minimizing Register Use Always Wise?

SU minimizes the *number* of registers used but at the *cost* of reduced ILP.



Since only 2 registers are used, there is little possibility of parallel evaluation.

When more registers are used, there is often more potential for parallel evaluation:



Here as many as *four* registers may be used to increase parallelism.

Reading Assignment

- Read Section 15.3 (Register Allocation and Temporary Management) from Chapter 15
- Get Class Handout 3 from DOIT.
- Read Chaitin's paper, "Register Allocation via Coloring."

Optimal Translation for DAGs is Much Harder

If variables or expression values may be *shared* and *reused*, optimal code generation becomes NP-Complete.

Example: `a+b*(c+d)+a*(c+d)`

We must decide how long to hold each value in a register. Best orderings may "skip" between subexpressions

Reference: R. Sethi, "Complete Register Allocation Problems," *SIAM Journal of Computing*, 1975.

Scheduling Expression Trees

Reference: S. Kurlander, T. Proebsting and C. Fischer, "Efficient Instruction Scheduling for Delayed-Load Architectures," *ACM Transactions on Programming Languages and Systems*, 1995. (Linked from class Web page)

The Sethi-Ullman Algorithm minimizes register usage, without regard to code scheduling.

On machines with *Delayed Loads*, we also want to avoid stalls.

What is a Delayed Load?

Most pipelined processors require a delay of one or more instructions between a load of register R and the first use of R.

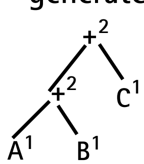
If a register is used "too soon," the processor may stall execution until the register value becomes available.

```
ld   [a],%r1
                 ← Stall!
add  %r1,1,%r1
```

We try to place an instruction that doesn't use register R immediately after a load of R.

This allows useful work instead of a wasteful stall.

The Sethi-Ullman Algorithm generates code that will stall:



```
ld  [A], %l0
ld  [B], %l1   <---- Stall!
add %l0,%l1,%l0
ld  [C], %l1   <---- Stall!
add %l0,%l1,%l0
```

In fact, if we use the fewest possible registers, stalls are *Unavoidable*!

Why?

Loads increase the number of registers in use.

Binary operations decrease the number of registers in use (2 Operands, 1 Result).

The load that brings the number of registers in use up to the minimum number needed *must* be followed by an operator that uses the just-loaded value. This implies a stall.

We'll need to allocate an *extra register* to allow an independent instruction to fill each delay slot of a load.

Extended Register Needs

Abbreviated as *ERN*

ERN(Identifier) = 2

ERN(Literal) = 1

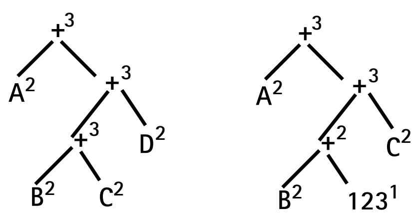
ERN(Op) =

If ERN(Left) = ERN(Right)

Then ERN(Left) + 1

Else Max(ERN(Left), ERN(Right))

Example



Idea of the Algorithm

1. Generate instructions in the same order as Sethi-Ullman, but use Pseudo-Registers instead of actual machine registers.
2. Put generated instructions into a "Canonical Order" (as defined below).
3. Map pseudo-registers to actual machine registers.

What are Pseudo-Registers?

They are unique temporary locations, unlimited in number and generated as needed, that are used to model registers prior to register allocation.

Canonical Form for Expression Code

(Assume R registers will be used)

Desired instruction ordering:

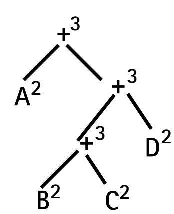
1. R load instructions
2. Pairs of Operator/Load instructions
3. Remaining operators

This canonical form is obtained by "sliding" load instructions upward (earlier) in the original code ordering.

Note that:

- Moving loads upward is *always* safe, since each pseudo-register is assigned to only once.
- No more than R registers are ever live.

Example




```
ld   [B], PR1
ld   [C], PR2
add PR1,PR2,PR3
ld   [D], PR4
add PR3,PR4,PR5
ld   [A], PR6
add PR6,PR5,PR7
```

Let R = 3, the minimum needed for a delay-free schedule.

Put into Canonical Form:

```
ld   [B], PR1
ld   [C], PR2
ld   [D], PR4
add PR1,PR2,PR3
ld   [A], PR6
add PR3,PR4,PR5
add PR6,PR5,PR7
```

(Before Register Assignment)

```
ld   [B], %l0
ld   [C], %l1
ld   [D], %l2
add %l0,%l1,%l0
ld   [A], %l1
add %l0,%l2,%l0
add %l1,%l0,%l0
```

(After Register Assignment)

No Stalls!

Does This Algorithm Always Produce a Stall-Free, Minimum Register Schedule?

Yes—if one exists!

For very simple expressions (one or two operands) no stall-free schedule exists.

For example: `a=b;`

```
ld  [b], %l0
st  %l0, [a]
```


Why Does the Algorithm Avoid Stalls?

Previously, certain "critical" loads had to appear just before an operation that used their value.

Now, we have an "extra" register. This allows critical loads to move up one or more places, avoiding any stalls.

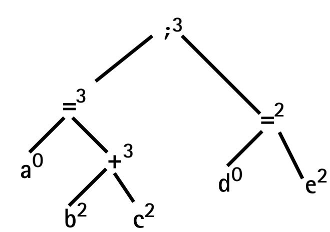
How Do We Schedule Small Expressions?

Small expressions (one or two operands) are common. We'd like to avoid stalls when scheduling them.

Idea—Blend small expressions together into larger expression trees, using "," and ";" like binary operators.

Example

`a=b+c; d=e;`




```
ld   [b], PR1
ld   [c], PR2
add  PR1,PR2,PR3
st   PR3, [a]
ld   [e], PR4
st   PR4, [d]
```

Orginal Code

```
ld   [b], PR1
ld   [c], PR2
ld   [e], PR4
add  PR1,PR2,PR3
st   PR3, [a]
st   PR4, [d]
```

In Canonical Form

```
ld   [b], %l0
ld   [c], %l1
ld   [e], %l2
add  %l0,%l1,%l0
st   %l0, [a]
st   %l2, [d]
```

After Register Assignment

Global Register Allocation

Allocate registers across an entire subprogram.

A Global Register Allocator must decide:

- What values are to be placed in registers?
- Which registers are to be used?
- For how long is each *Register Candidate* held in a register?

Live Ranges

Rather than simply allocate a value to a fixed register throughout an entire subprogram, we prefer to *split* variables into *Live Ranges*.

What is a Live Range?

It is the span of instructions (or basic blocks) from a definition of a variable to all its uses.

Different assignments to the same variable may reach distinct & disjoint instructions or basic blocks.

If so, the live ranges are *Independent*, and may be assigned *Different* registers.

Example

```
a = init();
for (int i = a+1; i < 1000; i++){
   b[i] = 0; }
a = f(i);
print(a);
```

The two uses of variable `a` comprise *Independent* live ranges.

Each can be allocated separately.

If we insisted on allocating variable `a` to a fixed register for the whole subprogram, it would *conflict* with the loop body, greatly reducing its chances of successful allocation.

Granularity of Live Ranges

Live ranges can be measured in terms of individual instructions or basic blocks.

Individual instructions are more precise but basic blocks are less numerous (reducing the size of sets that need to be computed).

We'll use basic blocks to keep examples concise.

You can define basic blocks that hold only one instruction, so computation in terms of basic blocks is still fully general.

Computation of Live Ranges

First construct the Control Flow Graph (CFG) of the subprogram.

For a Basic Block b:

Let Preds(b) = the set of basic blocks that are Immediate Predecessors of b in the CFG.

Let Succ(b) = the set of basic blocks that are Immediate Successors to b in the CFG.

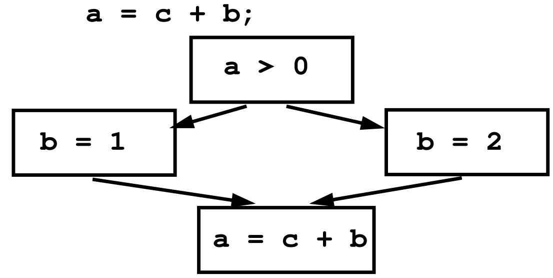
Control Flow Graphs

A Control Flow Graph (CFG) models possible execution paths through a program.

Nodes are basic blocks and arcs are potential transfers of control.

For example,

```
if (a > 0)
      b = 1;
else  b = 2;
a = c + b;
```



For a Basic Block b and Variable V:

Let DefsIn(b) = the set of basic blocks that contain definitions of V that reach (may be used in) the beginning of Basic Block b.

Let DefsOut(b) = the set of basic blocks that contain definitions of V that reach (may be used in) the end of Basic Block b.

If a definition of V reaches b, then the register that holds the value of that definition must be allocated to V in block b.

Otherwise, the register that holds the value of that definition may be used for other purposes in b.

The sets Preds and Succ are derived from the structure of the CFG.

They are given as part of the definition of the CFG.

DefsIn and DefsOut must be computed, using the following rules:

1. If Basic Block b contains a definition of V then
   DefsOut(b) = $\{b\}$
2. If there is no definition to V in b then
   DefsOut(b) = DefsIn(b)
3. For the First Basic Block, b_0:
   DefsIn(b_0) = ϕ
4. For all Other Basic Blocks
   $$\text{DefsIn}(b) = \bigcup_{p \in \text{Preds}(b)} \text{DefsOut}(p)$$

Reading Assignment

- Read George and Appel's paper, "Iterated Register Coalescing." (Linked from Class Web page)
- Read Larus and Hilfinger's paper, "Register Allocation in the SPUR Lisp Compiler."

Liveness Analysis

Just because a definition reaches a Basic Block, b, *does not* mean it must be allocated to a register at b.

We also require that the definition be *Live* at b. If the definition is dead, then it will no longer be used, and register allocation is unnecessary.

For a Basic Block b and Variable V:

LiveIn(b) = true if V is Live (will be used before it is redefined) at the beginning of b.

LiveOut(b) = true if V is Live (will be used before it is redefined) at the end of b.

LiveIn and LiveOut are computed, using the following rules:

1. If Basic Block b has no successors then
   LiveOut(b) = false
2. For all Other Basic Blocks

$$\text{LiveOut}(b) = \bigvee_{s \in \text{Succ}(b)} \text{LiveIn}(s)$$

3. LiveIn(b) =
   If V is used before it is defined in
   Basic Block b
   Then true
   Elsif V is defined before it is
   used in Basic Block b
   Then false
   Else LiveOut(b)

Merging Live Ranges

It is possible that each Basic Block that contains a definition of v creates a *distinct* Live Range of V.

$\forall$ Basic Blocks, b, that contain a definition of V:

$$\text{Range}(b) = \{b\} \cup \{k \mid b \in \text{DefsIn}(k) \;\&\; \text{LiveIn}(k)\}$$

This rule states that the Live Range of a definition to V in Basic Block b is b plus all other Basic Blocks that the definition of V reaches and in which V is live.

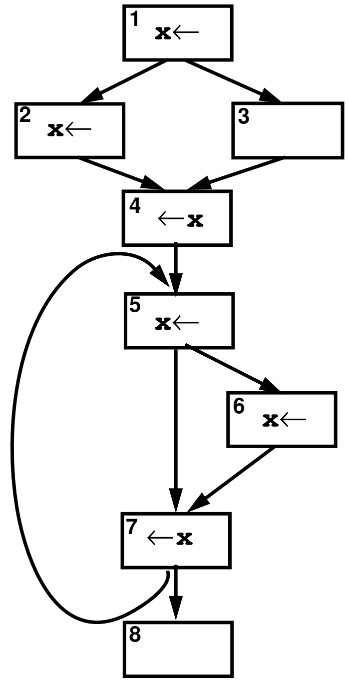
If two Live Ranges overlap (have one of more Basic Blocks in common), they *must* share the same register too. (Why?)

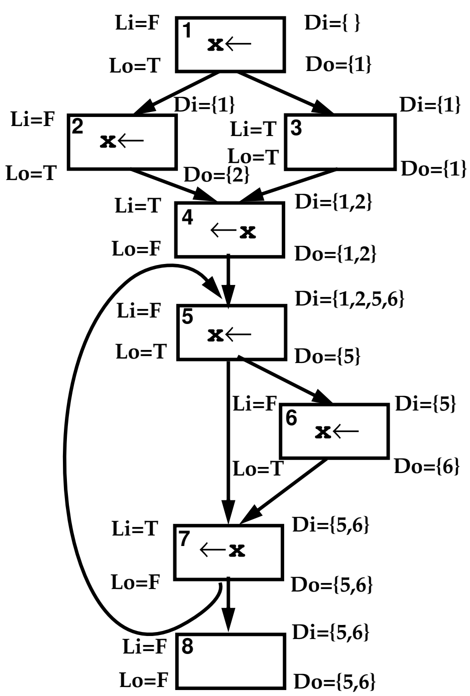
Therefore,

If $Range(b_1) \cap Range(b_2) \neq \phi$

Then replace
$Range(b_1)$ and $Range(b_2)$
with $Range(b_1) \cup Range(b_2)$

Example





The Live Ranges we Compute are

Range(1) = {1} U {3,4} = {1,3,4}

Range(2) = {2} U {4} = {2,4}

Range(5) = {5} U {7} = {5,7}

Range(6) = {6} U {7} = {6,7}

Ranges 1 and 2 overlap, so

Range(1) = Range(2) = {1,2,3,4}

Ranges 5 and 6 overlap, so

Range(5) = Range(6) = {5,6,7}

Interference Graph

An *Interference Graph* represents interferences between Live Ranges.

Two Live Ranges *interfere* if they share one or more Basic Blocks in common.

Live Ranges that interfere *must* be allocated different registers.

In an Interference Graph:

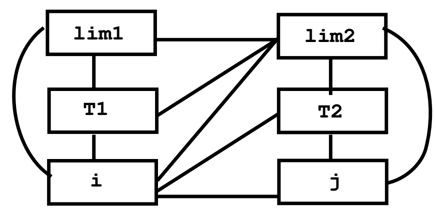
- Nodes are Live Ranges
- An undirected arc connects two Live Ranges if and only if they interfere

Example

```
int p(int lim1, int lim2) {
  for (i=0; i<lim1 && A[i]>0;i++){}
  for (j=0; j<lim2 && B[j]>0;j++){}
  return i+j;
}
```

We optimize array accesses by placing `&A[0]` and `&B[0]` in temporaries:

```
int p(int lim1, int lim2) {
  int *T1 = &A[0];
  for (i=0; i<lim1 && *(T1+i)>0;i++){}
  int *T2 = &B[0];
  for (j=0; j<lim2 && *(T2+j)>0;j++){}
  return i+j;
}
```



Register Allocation via Graph Coloring

We model global register allocation as a Coloring Problem on the Interference Graph

We wish to use the fewest possible colors (registers) subject to the rule that two connected nodes can't share the same color.

Optimal Graph Coloring is NP-Complete

Reference:

"Computers and Intractability," M. Garey and D. Johnson, W.H. Freeman, 1979.

We'll use a Heuristic Algorithm originally suggested by Chaitin et. al. and improved by Briggs et. al.

References:

"Register Allocation Via Coloring," G. Chaitin et. al., Computer Languages, 1981.

"Improvement to Graph Coloring Register Allocation," P. Briggs et. al., PLDI, 1989.

Coloring Heuristic

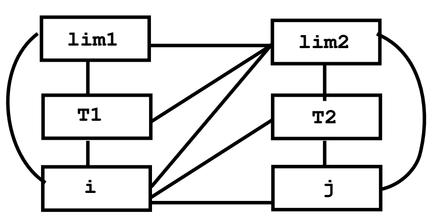
To R-Color a Graph (where R is the number of registers available)

1. While any node, n, has < R neighbors:
   Remove n from the Graph.
   Push n onto a Stack.
2. If the remaining Graph is non-empty:
   Compute the Cost of each node.
   The Cost of a Node (a Live Range) is the number of extra instructions needed if the Node isn't assigned a register, scaled by 10^{loop_depth}.
   Let NB(n) =
   Number of Neighbors of n.
   Remove that node n that has the smallest Cost(n)/NB(n) value.
   Push n onto a Stack.
   Return to Step 1.

3. While Stack is non-empty:
   Pop n from the Stack.
   If n's neighbors are assigned fewer than R colors
   Then assign n any unassigned color
   Else leave n uncolored.

Example

```
    int p(int lim1, int lim2) {
    int *T1 = &A[0];
    for (i=0; i<lim1 && *(T1+i)>0;i++){}
    int *T2 = &B[0];
    for (j=0; j<lim2 && *(T2+j)>0;j++){}
    return i+j;
}
```



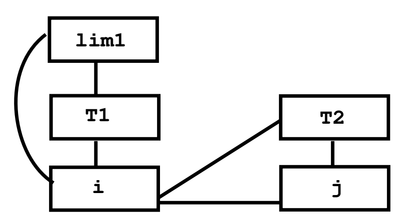
	lim1	lim2	T1	T2	i	j
Cost	11	11	11	11	42	42
Cost/ Neighbors	11/3	11/5	11/3	11/3	42/5	42/3

Do a 3 coloring

Since no node has fewer than 3 neighbors, we remove a node based on the minimum Cost/Neighbors value.

`lim2` is chosen.

We now have:



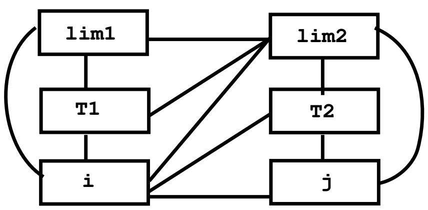
Remove (say) `lim1`, then `T1`, `T2`, `j` and `i` (order is arbitrary).

The Stack is:

i
j
T2
T1
lim1
lim2

Assuming the colors we have are R1, R2 and R3, the register assignment we choose is

`i`:R1, `j`:R2, `T2`:R3, `T1`:R2, `lim1`:R3, `lim2`:spill



Color Preferences

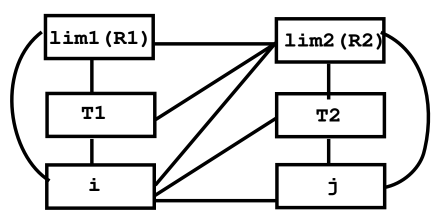
Sometimes we wish to assign a particular register (color) to a selected Live Range (e.g., a parameter or return value) *if possible*.

We can mark a node in the Interference Graph with a *Color Preference*.

When we unstack nodes and assign colors, we will avoid choosing color c if an uncolored neighbor has indicted a preference for it. If only color c is left, we take it (and ignore the preference).

Example

Assume in our previous example that lim1 has requested register R1 and lim2 has requested register R2 (because these are the registers the parameters are passed in).



i
j
T2
T1
lim1
lim2

Now when `i`, `j` and `T1` are unstacked, they respect `lim1`'s and `lim2`'s preferences:

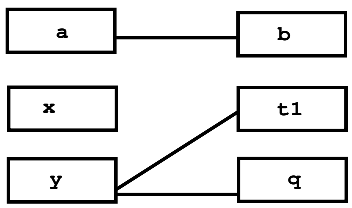
`i`:R3, `j`:R1, `T2`:R2, `T1`:R2, `lim1`:R1, `lim2`:spill

Using Coloring to Optimize Register Moves

A nice “fringe benefit” of allocating registers via coloring is that we can often *optimize away* register to register moves by giving the source and target the *same color*.

Consider

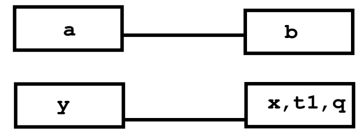
```
Live in: a,b
t1 = a + b
x = t1
y = x + 1
q = t1
Live out: y,q
```



We’d like `x`, `t1` and `q` to get the same color. How do we “force” this?

We can “merge” `x`, `t1` and `q` together:

```
Live in: a,b
t1 = a + b
x = t1
y = x + 1
q = t1
Live out: y,q
```



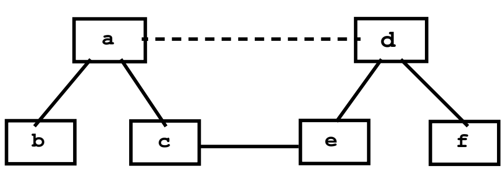
Now a 2-coloring that optimizes away both register to register moves is trivial.

Reckless Coalescing

Originally, Chaitin suggested merging *all* move-related nodes that don’t interfere.

This is *reckless*—the merged node may not be colorable!

(Is it worth a spill to save a move??)



This Graph is 2-colorable before the reckless merge, but *not* after.

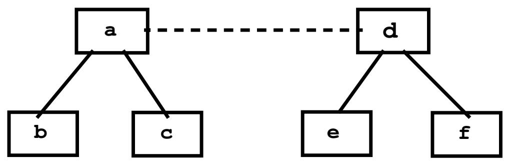
Conservative Coalescing

In response to Chaitin’s reckless coalescing approach, Briggs suggested a *more conservative* approach.

See “Improvement to Graph Coloring Register Allocation,” P. Briggs et. al., ACM Toplas, May 1994.

Briggs suggested that two move-related nodes should be merged *only if* the combined source and target node has fewer than R neighbors.

This *guarantees* that the combined node will be colorable, but may miss some optimization opportunities.



After a merge of nodes `a` and `d`, there will be four neighbors, but a 2-coloring is still possible.

Iterated Coalescing

This is an intermediate approach, that seeks to be safer than reckless coalescing and more effective than conservative coalescing. It was proposed by George and Appel.

1. Build:

   Create an Interference Graph, as usual. Mark source-target pairs with a special move-related arc (denoted as a dashed line).

2. Simplify:

   Remove and stack non-move-related nodes with < R neighbors.

3. Coalesce:

   Combine move-related pairs that will have < R neighbors after coalescing.

Repeat steps 2 and 3 until only nodes with R or more neighbors or move-related nodes remain or the graph is empty.

4. Freeze:

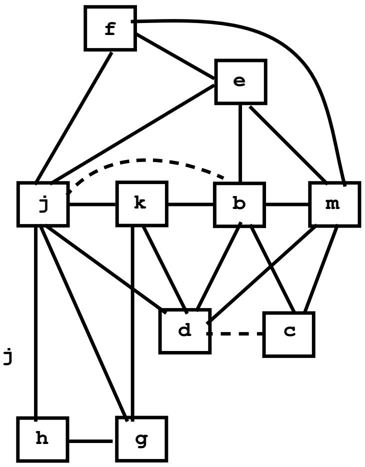
   If the Interference Graph is non-empty:
   Then If there exists a move-related node with < R neighbors
   Then: "Freeze in" the move and make the node non-move-related.
   Return to Steps 2 and 3.
   Else: Use Chaitin's Cost/Neighbors criterion to remove and stack a node.
   Return to Steps 2 and 3.

5. Unstack:

   Color nodes as they are unstacked as per Chaitin and Briggs.

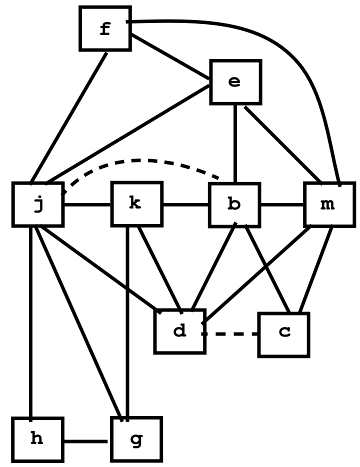
Example

```
Live in: k,j
g = mem[j+12]
h = k-1
f = g*h
e = mem[j+8]
m = mem[j+16]
b = mem[f]
c = e+8
d = c
k = m+4
j = b
goto d
Live out: d,k,j
```



Assume we want a 4-coloring.

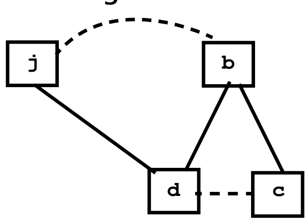
Note that neither **j&b** nor **d&c** can be conservatively colored.



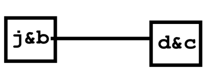
We simplify by removing nodes with fewer than 4 neighbors.

We remove and stack: **g**, **h**, **k**, **f**, **e**, **m**

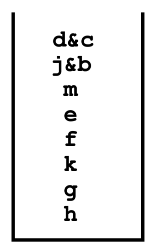
The remaining Interference Graph is



We can now conservatively coalesce the move-related pairs to obtain



These remaining nodes can now be removed and stacked.



We can now unstack and color:

d&c:R1, **j&b**:R2, **m**:R3, **e**:R4, **f**:R1, **k**:R3, **h**:R1, **g**:R4

No spills were required and both moves were optimized away.

Reading Assignment

- Read David Wall's paper, "Global Register Allocation at Link Time."

Priority-Based Register Allocation

Alternatives to Chaitin-style register allocation are presented in:

- Hennessy and Chow, "The priority-based coloring approach to register allocation," ACM TOPLAS, October 1990.
- Larus and Hilfinger, "Register allocation in the SPUR Lisp compiler," SIGPLAN symposium on Compiler Construction, 1986.

These papers suggest two innovations:

1. Use of a *Priority Value* to choose nodes to color in an Interference Graph.

   A Priority measures
   (Spill cost)/(Size of Live Range)

   The idea is that small live ranges with a high spill cost are ideal candidates for register allocation.
   As the size of a live range grows, it becomes less attractive for register allocation (since it "ties up" a register for a larger portion of a program).

2. Live Range Splitting

   Rather than spill an entire live range that can't be colored, the live range is split into two or more smaller live ranges that may be colorable.

Large vs. Small Live Ranges

- A large live range has less spill code. Values are directly read from and written to a register.
  But, a large live range is harder to allocate, since it may conflict with many other register candidates.
- A small live range is easier to allocate since it competes with fewer register candidates.
  But, more spill code is needed to load and save register values across live ranges.
- In the limit a live range can shrink to a single definition or use of a register.
  But, then we really don't have an effective register allocation at all!

Terminology

In an Interference Graph:

- A node with fewer neighbors than colors is termed *unconstrained*. It is trivial to color.
- A node that is not unconstrained is termed *constrained*. It may need to be split or spilled.


```
PriorityRegAlloc(proc, regCount) {
  ig ← buildInterferenceGraph(proc)
  unconstrained ←
     { n ∈ nodes(ig) | neighborCount(n) < regCount }
  constrained ←
     { n ∈ nodes(ig) | neighborCount(n) ≥ regCount }

  while( constrained ≠ φ )  {
     for ( c ∈ constrained such that not colorable(c)
            and canSplit(c) ) {
        c1, c2 ← split(c)
        constrained ← constrained - {c}
        if ( neighborCount(c1) < regCount )
              unconstrained ← unconstrained U { c1}
        else  constrained ← constrained U {c1}
        if ( neighborCount(c2) < regCount )
              unconstrained ← unconstrained U { c2}
        else  constrained ← constrained U {c2}
        for ( d ∈ neighbors(c) such that
             d ∈ unconstrained and
              neighborCount(d) ≥ regCount ){
               unconstrained ← unconstrained - {d}
              constrained ← constrained U {d}
     }     } // End of both for loops
```



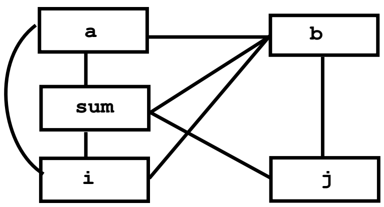
```
     /* At this point all nodes in constrained are
         colorable or can't be split */

     Select p ∈ constrained such that
                 priority(p) is maximized
      if ( colorable(p) )
              color(p)
      else  spill(p)
  } // End of While
 color all nodes ∈ unconstrained
}
```


How to Split a Constrained Node

- There are many possible partitions of a live range; too many to fully explore.
- Heuristics are used instead. One simple heuristic is:
    1. Remove the first basic block (or instruction) of the live range. Put it into a new live range, NR.
    2. Move successor blocks (or instructions) from the original live range into NR, as long as NR remains colorable.
    3. Single Basic Blocks (or instructions) that can't be colored are spilled.

Example

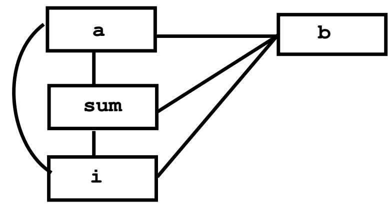
```
int sum(int a[], int b[]) {
  int sum = 0;
  for (int i=0; i<1000; i++)
     sum += a[i];
  for (int j=0; j<1000; j++)
     sum += b[j];
  return sum;
}
```



Assume we want a 3-coloring.

We first simplify the graph by removing unconstrained nodes (those with < 3 neighbors).

Node `j` is removed. We now have:

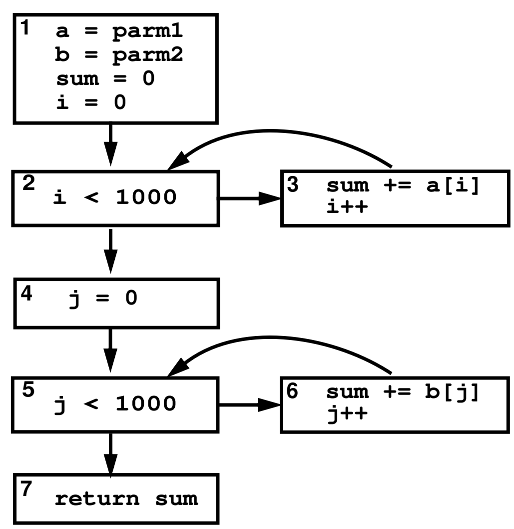


At this point, each node has 3 neighbors, so either spilling or splitting is necessary.

A spill really isn't attractive as each of the 4 register candidates is used within a loop, magnifying the costs of accessing memory.

Coloring by Priorities

We'll color constrained nodes by priority values, with preference given to large priority values.



	a	b	sum	i
Cost	11	11	42	41
Cost/Size	11/3	11/6	42/7	41/3

Variables `i`, `sum` and `a` are assigned colors `R1`, `R2` and `R3`.

Variable `b` can't be colored, so we will try to split it. `b`'s live range is blocks 1 to 6, with 1 as `b`'s entry point.

Blocks 1 to 3 can't be colored, so `b` is spilled in block 1. However, blocks 4 to 6 form a split live range that can be colored (using `R3`).

We will reload `b` into `R3` in block 4, and it will be register-allocated throughout the second loop. The added cost due to the split is minor—a store in block 1 and a reload in block 4.

Choice of Spill Heuristics

We have seen a number of heuristics used to choose the live ranges to be spilled (or colored).

These heuristics are typically chosen using one's intuition of what register candidates are most (or least) important. Then a heuristic is tested and "fine tuned" using a variety of test programs.

Recently, researchers have suggested using machine learning techniques to automatically determine effective heuristics.

In "Meta Optimization: Improving Compiler Heuristics with Machine Learning," Stephenson, Amarasinghe, et al, suggest using *genetic programming* techniques in which

priority functions (like choice of spill candidates) are mutated and allowed to "evolve."

Although the approach seems rather random and unfocused, it can be effective. Priority functions *better than* those used in real compilers have been reported, with research still ongoing.

Interprocedural Register Allocation

The goal of register allocation is to keep frequently used values in registers.

Ideally, we'd like to go to memory only to access values that may be aliased or pointed to.

For example, array elements and heap objects are routinely loaded from and stored to memory each time they are accessed.

With alias analysis, optimizations like Scalarization are possible.

```
for (i=0; i<1000; i++)
     for (j=0; j<1000; j++)
        a[i] += i * b[j];
```

is optimized to

```
for (i=0; i<1000; i++){
     int Ai = a[i];
     for (j=0; j<1000; j++)
        Ai += i * b[j];
     a[i] = Ai;
}
```


Attacking Call Overhead

- Even with good global register allocation calls are still a problem.
- In general, the caller and callee may use the same registers, requiring saves and restores across calls.
- Register windows help, but they are inflexible, forcing all subprograms to use the same number of registers.
- We'd prefer a register allocator that is sensitive to the calling structure of a program.

Reading Assignment

- Read "Minimum Cost Interprocedural Register Allocation," by S. Kurlander et al. (linked from class Web page).
- Get Handout #4 from DoIt.

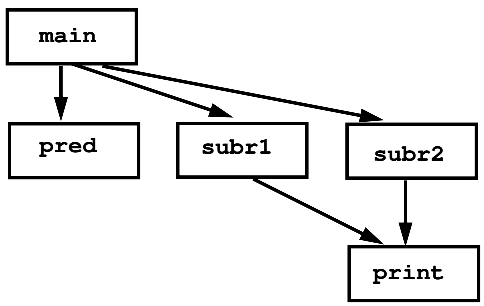
Call Graphs

A *Call Graph* represents the calling structure of a program.

- Nodes are subprograms (procedures and functions).
- Arcs represent calls between subprograms. An arc from A to B denotes that a call to B appears within A.
- For an indirect call (a function parameter or a function pointer) an arc is added to all potential callees.

Example

```
main() {
  if (pred(a,b))
       subr1(a)
  else subr2(b);}

bool pred(int a, int b){
   return a==b; }

subr1(int a){ print(a);}

subr2(int x){ print(2*x);}
```



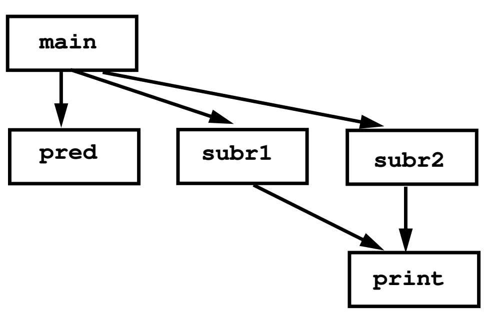
Wall's Interprocedural Register Allocator

Operates in two phases:

1. Register candidates are identified at the subprogram level.
   Each candidate (a single variable *or* a set of non-interfering live ranges) is compiled as if it *won't* get a register.
   At link-time unnecessary loads and stores are edited away *if* the candidate *is* allocated a register.
2. At link-time, when all subprograms are known and available, register candidates are allocated registers.

Identifying Interprocedural Register Sharing

If two subprograms are not connected in the call graph, a register candidate in each can share the same register without any saving or restoring across calls.



A register candidate from **`pred`**, **`subr1`** and **`subr2`** can all share one register.

At the interprocedural level we must answer 2 questions:

1. A local candidate of one subprogram can share a register with candidates of what other subprograms?
2. Which local register candidates will yield the greatest benefit if given a register?

Wall designed his allocator for a machine with 52 registers. This is enough to divide all the registers among the subprograms without any saves or restores at call sites.

With fewer registers, spills, saves and restores will often be needed (if registers are used aggressively within a subprogram).

Restrictions on the Call Graph

Wall limited calls graphs to DAGs since cycles in a call graph imply recursion, which will force saves and restores (why?)

Cost Computations

Each register candidate is given a per-call cost, based on the number of saves and restores that can be removed, scaled by $10^{\text{loop\_depth}}$.

This benefit is then multiplied by the *expected* number of calls, obtained by summing the total number of call sites, scaled by loop nesting depth.

Grouping Register Candidates

We now have an estimate of the benefit of allocating a register to a candidate. Call this estimate
cost(candidate)

We estimate potential interprocedural sharing of register candidates by assigning each candidate to a *Group*.

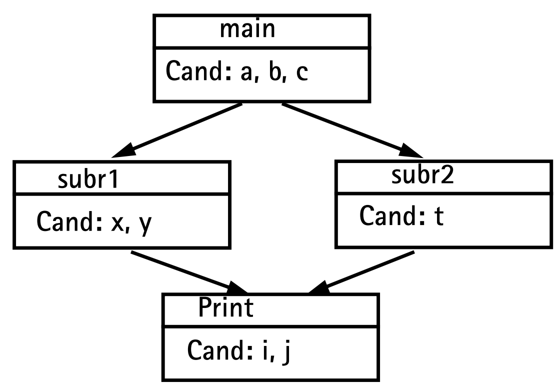
All candidates within a group can share a register. No two candidates in any subprogram are in the same group.

Groups are assigned during a reverse depth-first traversal of the call graph.

```
AsgGroup(node n) {
   if (n is a leaf node)
       grp = 0
   else { for (each c ∈ children(n)) {
              AsgGroup(c) }
       grp =
          1+ Max (Max group used in c)
           c ∈ children(n)
   }
   for (each r ∈ registerCandidates(n)){
         assign r to grp
         grp++  }
}
```

Global variables are assigned to a singleton group.

Example



At Print: grp(i)=0, grp(j)=1

At subr1: Max grp used in print is 1
grp(x)=2, grp(y)=3

At subr2: Max grp used in print is 1
grp(t)=2

At main: Max grp used in children is 3
grp(a)=4, grp(b)=5, grp(c)=6

If A calls B (directly or indirectly), then none of A's register candidates are in the same group as any of B's register candidates.

This *guarantees* that A and B will use different registers.

Thus no saves or restores are needed across a call from A to B.

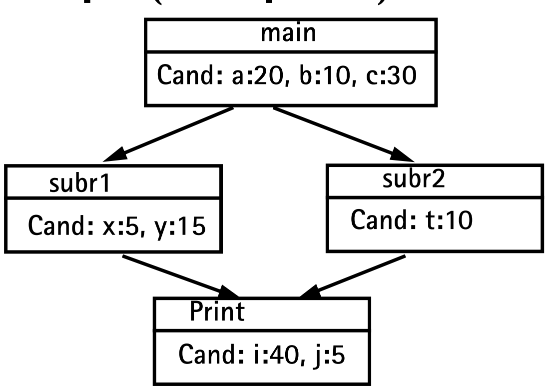
Assigning Registers to Groups

$$\text{Cost(group)} = \sum_{\text{candidates} \in \text{group}} \text{cost(candidates)}$$

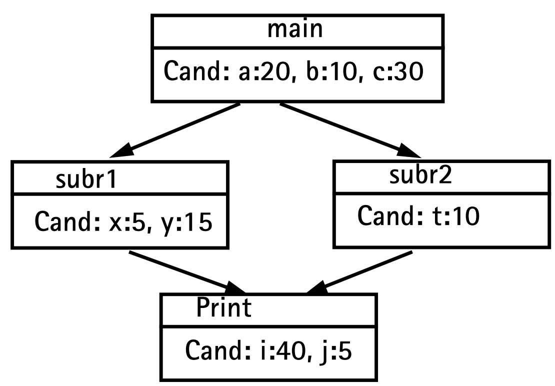
We assign registers to groups based on the cost of each group, using an "auction."

for (r=0; r < RegisterCount; r++) {
  Let G be the group with the
    greatest cost that has not yet
    been assigned a register.
  Assign r to G
}

Example (3 Registers)



Group	Members	Cost
0	i	40
1	j	5
2	x, t	15
3	y	15
4	a	20
5	b	10
6	c	30



The 3 registers are given to the groups with the highest weight,

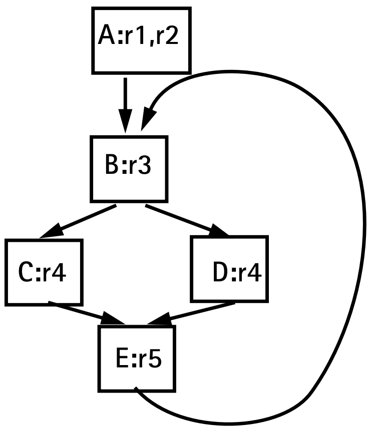
i (40), c(30), a(20).

Is this optimal?

No! If y and t are grouped together, y and t (cost=25) get the last register.

Recursion

To handle recursion, any call to a subprogram "up" in the call graph must save and restore all registers possibly in use between the caller and callee.



A call from E to B saves r3 to r5.

Performance

Wall found interprocedural register allocation to be very effective (given 52 Registers!).

Speedups of 10-28% were reported.

Even with only 8 registers, speedups of 5-20% were observed.

Optimal Interprocedural Register Allocation

Wall's approach to interprocedural register allocation isn't optimal because register candidates aren't grouped to achieve maximum benefit.

Moreover, the placement of save and restore code *if needed* isn't considered.

These limitations are addressed by Kurlander in "Minimum Cost Interprocedural Register Allocation."

Optimal Save-Free Interprocedural Register Allocation

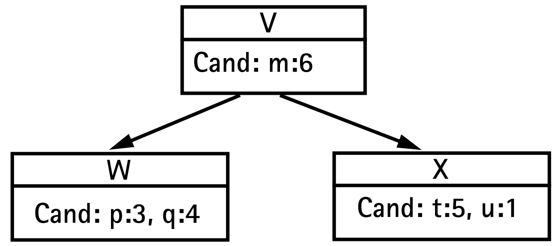
- Allocation is done on a cycle-free call graph.
- Each subprogram has one or more register candidates, c_i.
- Each register candidate, c_i, has a cost (or benefit), w_i, that is the improvement in performance if c_i is given a register. (This w_i value is scaled to include nested loops and expected call frequencies.)

Interference Between Register Candidates

The notion of interference is extended to include interprocedural register candidates:

- Two Candidates in the same subprogram always interfere.
  (Local non-interfering variables and values have already been grouped into interprocedural register candidates.)
- If subprogram P calls subprogram Q (directly or indirectly) then register candidates within P always interfere with register candidates within Q.

Example



The algorithm can group candidate p with either t or u (since they don't interfere). It can also group candidate q with either t or u.

If two registers are available, it must "discover" that assigning R1 to q&t, and R2 to m is optimal.

Non-interfering register candidates are grouped into registers so as to solve:

$$\text{Maximize} \sum_{c_j \in \bigcup_{i=1}^{k} R_i} w_j$$

That is, we wish to group sets of non-interfering register candidates into k registers such that the overall benefit is maximized.

But how do we solve this?

Certainly examining all possible groupings will be prohibitively expensive!

Kurlander solved this problem by mapping it to a known problem in Integer Programming:

the Dual Network Flow Problem.

Solution techniques for this problem are well known—libraries of standard solution algorithms exist.

Moreover, this problem can be solved in *polynomial time*.

That is, it is "easier" than optimal global (intraprocedural) register allocation, which is NP-complete!

Reading Assignment

- Read Section 15.4 (Code Scheduling) of Chapter 15.
- Read Gibbon's and Muchnick's paper, "Efficient Instruction Scheduling for a Pipelined Architecture."
- Read Kerns and Eggers' paper, "Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain." (Linked from the class Web page.)

Adding Saves & Restores

Wall designed his save-free interprocedural allocator for a machine with 52 registers.

Most computers have far fewer registers, and hence saving and restoring across calls, *when profitable*, should be allowed.

Kurlander's Technique can be extended to include save/restore costs. If the cost of saving and restoring is *less* than the benefit of allocating an extra register, saving is done. Moreover, saving is done where it is *cheapest* (not closest!).

Example

```
main() { ... p(); ...}

p(){ ...
     for (i=0; i<1000000; i++){
          q():
   }
}
```

We first allocate registers in a save-free mode. After all Registers have been allocated, `q` may need additional registers.

Most allocators would add save/ restore code at `q`'s call site (or `q`'s prologue and epilogue).

An optimal allocator will place save/ restore code at `p`'s call site, freeing a register that `p` doesn't even want (but that `q` does want!)

Extending the Cost Model

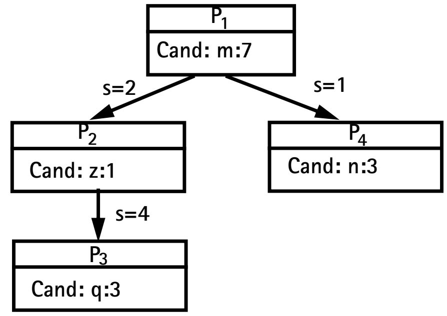
- As before, we group register candidates of different subprograms into registers.
- Now only candidates within the same subprogram automatically interfere.
- Saves are placed on the edges of the call graph.
- We aim to solve

$$\text{Maximize} \sum_{c_j \in \bigcup_{i=1}^{k} R_i} w_j - \sum_{e_m \in \text{ Edges}} s_m * \text{Saved}_m$$

where s_m is the per/register save/ restore cost and Saved_m is the number of registers saved on edge e_m.

- As registers are saved, they may be reused in child subprograms.
- This optimization problem can be solved as a Network Dual Flow Problem.
- Again, the solution algorithm is *polynomial*.

Example (One Register)



P_1 gets R1 save-free for m.

A save on $P_1 \rightarrow P_4$ costs 1 and gives a register to n (net profit =2), so we do it.

A save on $P_1 \rightarrow P_2$ for z costs 2, and yields 1, which isn't profitable.

A save on $P_2 \rightarrow P_3$ for q costs 4, and yields 3, which isn't profitable.

A save on $P_1 \rightarrow P_2$ for q costs 2, and yields 3, which *is* a net gain.

Handling Global Variables

- Wall's technique handled globals by assuming they interfere with all subprograms and all other globals.
- Kurlander's approach is incremental (and non-optimal):

  First, an optimal allocation for r registers is computed.

  Next, one register is "stolen" and assigned interprocedurally to the most beneficial global. (Subprograms that don't use the global can save and restore it locally, allowing local reuse).
  An optimal allocation using R-1 registers is computed. If this solution plus the shared global is more profitable than the R register

solution, the global allocation is "locked in."

Next, another register is "stolen" for a global, leaving R-2 for interprocedural allocation.

This process continues until stealing another register for a global isn't profitable.

Why is Optimal Interprocedural Register Allocation Easier than Optimal IntraProcedural Allocation?

This result seems counter-intuitive. How can allocating a whole program be *easier* (computationally) than allocating only one subprogram.

Two observations provide the answer:

- Interprocedural allocation assumes some form of local allocation has occurred (to identify register candidates).
- Interprocedural interference is *transitive* (if A interferes with B and B interferes with C then A interferes with B). But intraprocedural interference *isn't* transitive!

Code Scheduling

Modern processors are pipelined.

They give the impression that all instructions take unit time by executing instructions in *stages* (steps), as if on an assembly line.

Certain instructions though (loads, floating point divides and square roots, delayed branches) take more than one cycle to execute.

These instructions may *stall* (halt the processor) or require a nop (null operation) to execute properly.

A *Code Scheduling* phase may be needed in a compiler to avoid stalls or eliminate nops.

Scheduling Expression DAGs

After generating code for a DAG or basic block, we may wish to schedule (reorder) instructions to reduce or eliminate stalls.

A *Postpass Scheduler* is run after code selection and register allocation.

Postpass schedulers are very general and flexible, since they can be used with code generated by any compiler with any degree of optimization

But, since they can't modify register allocations, they can't always avoid stalls.

Dependency DAGs

Obviously, not all reorderings of generated instructions are acceptable.

Computation of a register value must precede all uses of that value.

A store of a value must precede all loads that might read that value.

A *Dependency Dag* reflects essential ordering constraints among instructions:

- Nodes are Instructions to be scheduled.
- An arc from Instruction i to Instruction j indicates that i must be executed before j may be executed.

Kinds of Dependencies

We can identify several kinds of dependencies:

- True Dependence:

  An operation that uses a value has a true dependence (also called a flow dependence) upon an earlier operation that computes the value. For example:

```
mov  1, %l2
add  %l2, 1, %l2
```

- Anti Dependence:

  An operation that writes a value has a anti dependence upon an earlier operation that reads the value. For example:

```
add  %l2, 1, %l0
mov  1, %l2
```


- Output Dependence:

  An operation that writes a value has a output dependence upon an earlier operation that writes the value. For example:

```
mov  1, %l2
mov  2, %l2
```

Collectively, true, anti and output dependencies are called data dependencies. Data dependencies constrain the order in which instructions may be executed.

Example

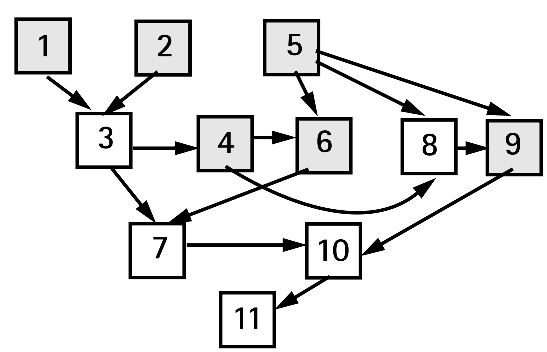
Consider the code that might be generated for

`a = ((a+b) + (c*d)) + ((c+d) * d);`

We'll assume 4 registers, the minimum possible, and we'll reuse already loaded values.

Assume a 1 cycle stall between a load and use of the loaded value and a 2 cycle stall between a multiplication and first use of the product.


```
1.  ld   [a], %r1
2.  ld   [b], %r2      <-- Stall
3.  add  %r1,%r2,%r1
4.  ld   [c], %r2
5.  ld   [d], %r3
6.  smul %r2,%r3,%r4   <-- Stall
7.  add  %r1,%r4,%r1   <-- Stall*2
8.  add  %r2,%r3,%r2
9.  smul %r2,%r3,%r2
10. add  %r1,%r2,%r1   <-- Stall*2
11. st   %r1,[a]       (6 Stalls Total)
```



Scheduling Requires Topological Traversal

Any valid code schedule is a *Topological Sort* of the dependency dag.

To create a code schedule you

(1) Pick any root of the Dag.

(2) Remove it from the Dag and schedule it.

(3) Iterate!

Choosing a *Minimum Delay* schedule is NP-Complete:

"Computers and Intractability," M. Garey and D. Johnson, W.H. Freeman, 1979.

Dynamically Scheduled (Out of Order) Processors

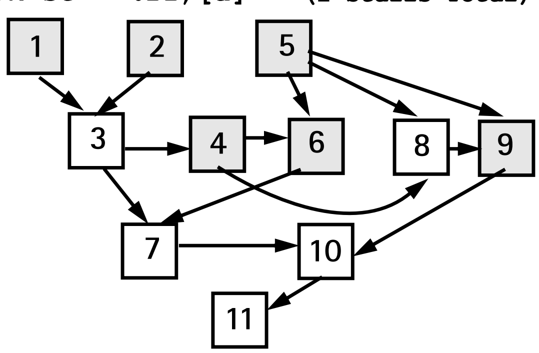
To avoid stalls, some processors can execute instructions *Out of Program Order*.

If an instruction can't execute because a previous instruction it depends upon hasn't completed yet, the instruction can be "held" and a successor instruction executed instead.

When needed predecessors have completed, the held instruction is released for execution.

Example

```
 1. ld   [a], %r1
 2. ld   [b], %r2
 5. ld   [d], %r3
 3. add  %r1,%r2,%r1
 4. ld   [c], %r2      <--- Stall
 6. smul %r2,%r3,%r4
 8. add  %r2,%r3,%r2
 9. smul %r2,%r3,%r2
 7. add  %r1,%r4,%r1   <--- Stall
10. add  %r1,%r2,%r1
11. st   %r1,[a]     (2 Stalls Total)
```



Limitations of Dynamic Scheduling

1. Extra processor complexity.
2. Register renaming (to avoid *False Dependencies*) may be needed.
3. Identifying instructions to be delayed takes time.
4. Instructions "late" in the program can't be started earlier.

Reading Assignment

- Read Goodman and Hsu's paper, "Code Scheduling and Register Allocation in Large Basic Blocks."
- Read Bernstein and Rodeh's paper, "Global Instruction Scheduling for Superscalar Machines."
  (Linked from the class Web page.)

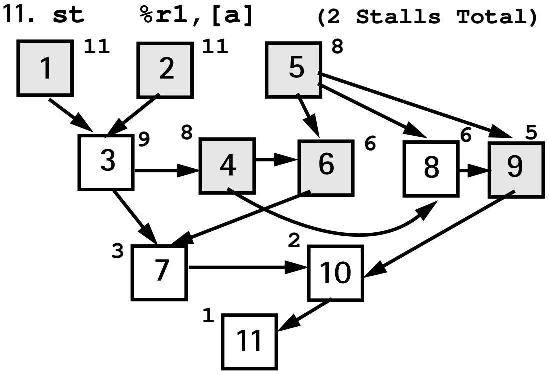
Gibbons & Muchnick Postpass Code Scheduler

1. If there is only one root, schedule it.
2. If there is more than one root, choose that root that won't be stalled by instructions already scheduled.
3. If more than one root can be scheduled without stalling, consider the following rules (in order);
   (a) Does this root stall any of its successors?
   (If so, schedule it immediately.)
   (b) How many new roots are exposed if this node is scheduled?
   (More is better.)

(c) Which root has the longest weighted path to a leaf (using instruction delays as the weight).
(The "critical path" in the DAG gets priority.)

Example

```
 1.  ld    [a], %r1  //Longest path
 2.  ld    [b], %r2  //Exposes a root
 5.  ld    [d], %r3  //Not delayed
 3.  add   %r1,%r2,%r1 //Only choice
 4.  ld    [c], %r2  //Only choice
 6.  smul  %r2,%r3,%r4 //Stalls succ.
 8.  add   %r2,%r3,%r2 //Not delayed
 9.  smul  %r2,%r3,%r2 //Not delayed
 7.  add   %r1,%r4,%r1 //Only choice
10.  add   %r1,%r2,%r1 //Only choice
11.  st    %r1,[a]     (2 Stalls Total)
```



False Dependencies

We still have delays in the schedule that was produced because of “false dependencies.”

Both `b` and `c` are loaded into `%r2`. This limits the ability to move the load of `c` prior to any use of `%r2` that uses `b`.

To improve our schedule we can use a processor that renames registers *or* allocate additional registers to remove false dependencies.

Register Renaming

Many out of order processors automatically rename distinct uses of the same architectural register to distinct internal registers.

Thus

```
ld [a],%r1
ld [b],%r2
add %r1,%r2,%r1
ld [c],%r2
```

is executed as if it were

```
ld [a],%r1
ld [b],%r2
add %r1,%r2,%r3
ld [c],%r4
```

Now the final load can be executed prior to the add, eliminating a stall.

Compiler Renaming

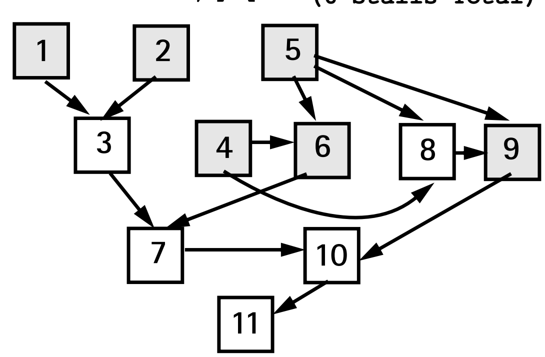
A compiler can also use the idea of renaming to avoid unnecessary stalls.

An extra register may be needed (as was the case for scheduling expression trees).

Also, a *round-robin* allocation policy is needed. Registers are reused in a *cyclic* fashion, so that the most recently freed register is reused last, not first.

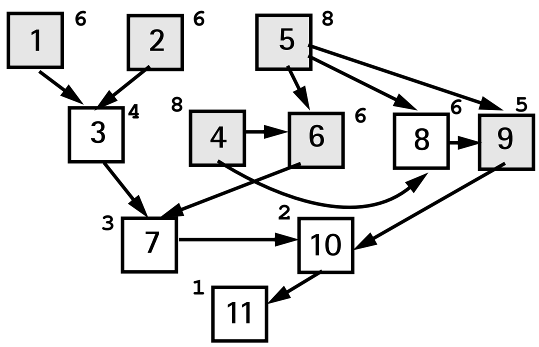
Example

```
1.  ld    [a], %r1
2.  ld    [b], %r2        <-- Stall
3.  add   %r1,%r2,%r1
4.  ld    [c], %r3
5.  ld    [d], %r4
6.  smul  %r3,%r4,%r5     <-- Stall
7.  add   %r1,%r5,%r2     <-- Stall*2
8.  add   %r3,%r4,%r3
9.  smul  %r3,%r4,%r3
10. add   %r2,%r3,%r2     <-- Stall*2
11. st    %r2,[a]         (6 Stalls Total)
```



After Scheduling:

```
4.  ld   [c],%r3    //Longest path
5.  ld   [d],%r4    //Exposes a root
1.  ld   [a],%r1    //Stalls succ.
2.  ld   [b],%r2    //Exposes a root
6.  smul %r3,%r4,%r5 //Stalls succ.
8.  add  %r3,%r4,%r3 //Longest path
9.  smul %r3,%r4,%r3 //Stalls succ.
3.  add  %r1,%r2,%r1 //Only choice
7.  add  %r1,%r5,%r2 //Only choice
10. add  %r2,%r3,%r2 //Only choice
11. st   %r2,[a]     (0 Stalls Total)
```



Balanced Scheduling

When scheduling a load, we normally anticipate the *best* case, a hit in the primary cache.

On older architectures this makes sense, since we stall execution on a cache miss.

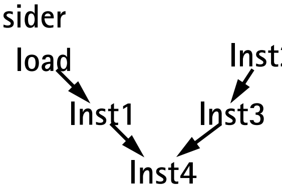
Many newer architectures are *non-blocking*. This means we can continue execution after a miss until the loaded value is used.

Assume a Cache miss takes N cycles (N is typically 10 or more).

Do we schedule a load anticipating a 1 cycle delay (a hit) or an N cycle delay (a miss)?

Neither *Optimistic Scheduling* (expect a hit) nor *Pessimistic Scheduling* (expect a miss) is *always* better.

Consider



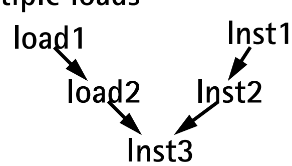
An Optimistic Schedule is

load Inst2 Inst1 Inst3 Inst4	Fine for a hit; inferior for a miss.

A Pessimistic Schedule is

load Inst2 Inst3 Inst1 Inst4	Fine for a hit; better for a miss.

But things become more complex with multiple loads



An Optimistic Schedule is

load1 Inst1 load2 Inst2 Inst3	Better for hits; same for misses.

A Pessimistic Schedule is

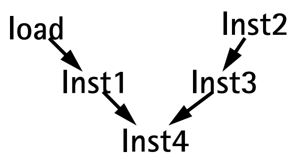
load1 Inst1 Inst2 load2 Inst3	Worse for hits; same for misses.

Balance Placement of Loads

Eggers suggests a *balanced scheduler* that spaces out loads, using available independent instructions as "filler."

The insight is that scheduling should not be driven by worst-case latencies but rather by available *Independent* Instructions.

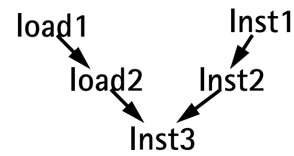
For



it produces

load Inst2 Inst3 Inst1 Inst4	Good; maximum distance between load and Inst1 in case of a miss.

For



balanced scheduling produces

load1 Inst1 load2 Inst2 Inst3	Good for hits; as good as possible for misses.

Idea of the Algorithm

Look at each Instruction, i, in the Dependency DAG.

Determine which loads can run in parallel with i and use all (or part) of i's execution time to cover the latency of these loads.

Compute available latency of each load:

Give each load instruction an initial latency of 1.

For (each instruction i in the Dependency DAG) do:

  Consider Instructions Independent of i:

    G_{ind} = DepDAG - (AllPred(i) U AllSucc(i) U {i})

  For (each connected subgraph c in G_{ind}) do:

    Find m = maximum number of load instructions on any path in c.

    For (each load d in c) do:

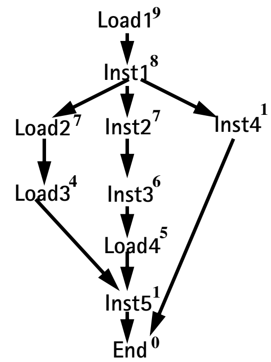
      add 1/m to d's latency.

Computing the Schedule Using Adjusted Latencies

Once latencies are assigned to each load (other instructions have a latency of 1), we annotate each instruction in the Dependency DAG with its critical path weight: the maximum latency (along any path) from the instruction to a Leaf of the DAG.

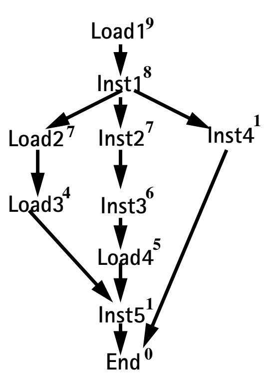
Instructions are scheduled using critical path values; the root with the highest critical path value is always scheduled next. In cases of ties (same critical path value), operations with the longest latency are scheduled first.

Example



	Ld 1	Ld 2	Ld 3	Ld 4	I1	I2	I3	I4	I5	Latency
Load1										1+0 = 1
Load2				1/2		1/2	1/2	1/2		1+2 = 3
Load3				1/2		1/2	1/2	1/2		1+2 = 3
Load4		1	1					1		1+3 = 4

Using the annotated Dependency Dag, instructions can be scheduled:



Load1 (0 latency; unavoidable)
Inst1
Load2 (3 instruction latency)
Inst2
Inst3
Load4 (2 instruction latency)
Load3 (1 instruction latency)
Inst4
Inst5

Goodman/Hsu Integrated Code Scheduler

Prepass Schedulers:
- Schedule code prior to register allocation.
- Can overuse registers—Always using a "fresh" register maximizes freedom to rearrange Instructions.

Postpass Schedulers:
- Schedule code after register allocation.
- Can be limited by "false dependencies" induced by register reuse.
- Example is Gibbons/Muchnick heuristic.

Integrated Schedulers

Capture best of both approaches.

When registers are plentiful, use additional registers to avoid stalls.

Goodman & Hsu call this *CSP*:
Code Scheduling for Pipelines.

When registers are scarce, switch to a policy that frees registers.

Goodman & Hsu call this *CSR*:
Code Scheduling to free Registers.

Assume code is generated in single assignment form, with a unique pseudo-register for each computed value.

We schedule from a DAG where nodes are operations (to be mapped to instructions), and arcs represent data dependencies.

Each node will have an associated Cost, that measures the execution and stall time of the instruction that the node represents.

Nodes are labeled with a critical path cost, used to select the "most critical" instructions to schedule.

Definitions

Leader Set:

Set of DAG nodes ready to be scheduled, possibly with an interlock.

Ready Set:

Subset of Leader Set; Nodes ready to be scheduled without an interlock.

AvailReg:

A count of currently unused registers.

MinThreshold:

Threshold at which heuristic will switch from avoiding interlocks to reducing registers in use.

Goodman/Hsu Heuristic:

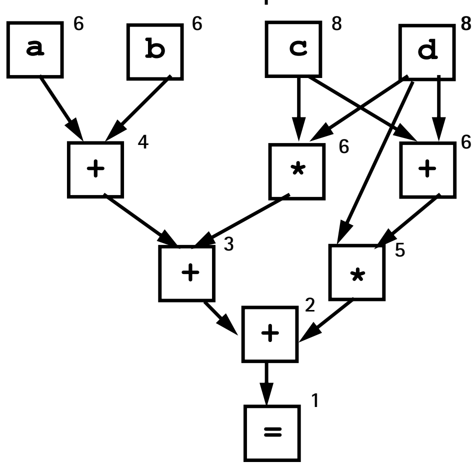
```
while (DAG ≠ φ) {
  if ( AvailReg > MinThreshold)
     if (ReadySet ≠ φ)
          Select Ready node with Max cost
     else Select Leader node with Max cost
  else  //  Reduce Registers in Use
     if (∃ node ∈ ReadySet that frees registers){
        Select node that frees most registers
        If (selected node isn't unique)
           Select node with Max cost   }
    elsif (∃ node ∈ LeaderSet that frees regs){
           Select node that frees most registers
          If (selected node isn't unique)
             Select node with fewest interlocks}
    else find a partially evaluated path and
          select a leader from this path
    else Select any node in ReadySet
    else Select any node in LeaderSet
Schedule Selected node
Update AvailReg count  }//end while
```

Example

We'll again consider

`a = ((a+b) + (c*d)) + ((c+d) * d);`

Again, assume a 1 cycle stall between a load and use of its value and a 2 cycle stall between a multiplication and first use of the product.



We'll try 4 registers (the minimum), then 5 registers.

Should MinThreshold be 0 or 1?

At MinThreshold = 1, we always have a register to hold a result, but we may force a register to be spilled too soon!

At MinThreshold = 0, we may be forced to spill a register to free a result register.

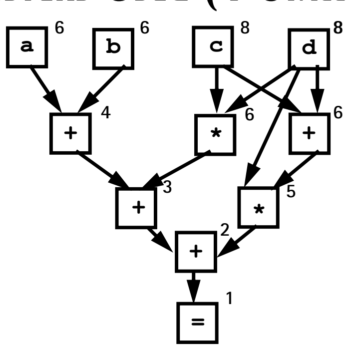
But, we'll also be able to schedule more aggressively.

Is a spill or stall worse?

Note that we may be able to "hide" a spill in a delay slot!

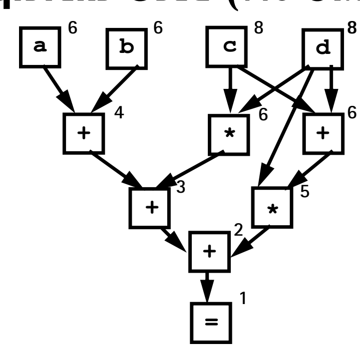
We'll be aggressive and use MinThreshold = 0.

4 Registers Used (1 Stall)



Instruction	Comment	Regs Used
`ld [c], %r1`	Choose ready, cost=8	1
`ld [d], %r2`	Choose ready, cost=8	2
`ld [a], %r3`	Choose ready, cost=6	3
`smul %r1,%r2,%r4`	Choose ready, cost=6	4
`add %r1,%r2,%r1`	Free a register	4
`smul %r1,%r2,%r1`	Free a register	3
`ld [b], %r2`	Choose ready, cost=6	4
`add %r3,%r2,%r3` ←	Choose a leader	3
`add %r3,%r4,%r3`	No choice	2
`add %r3,%r1,%r3`	No choice	1
`st %r3,[a]`	No choice	0

5 Registers Used (No Stalls)



Instruction	Comment	Regs Used
`ld [c], %r1`	Choose ready, cost=8	1
`ld [d], %r2`	Choose ready, cost=8	2
`ld [a], %r3`	Choose ready, cost=6	3
`smul %r1,%r2,%r4`	Choose ready, cost=6	4
`add %r1,%r2,%r1`	Choose ready, cost=6	4
`ld [b], %r5`	Choose ready, cost=6	5
`smul %r1,%r2,%r1`	Free a register	4
`add %r3,%r5,%r3`	Choose ready, cost=4	3
`add %r3,%r4,%r3`	No choice	2
`add %r3,%r1,%r3`	No choice	1
`st %r3,[a]`	No choice	0

Scheduling for Superscalar & Multiple Issue Machines

A number of computers have the ability to issue more than one instruction per cycle *if* the instructions are independent and satisfy constraints on available functional units.

Thus the instructions

```
add %r1,1,%r2
sub %r1,2,%r3
```

can issue and execute in parallel, but

```
add %r1,1,%r2
sub %r2,2,%r3
```

must execute sequentially.

Instructions that are linked by true or output dependencies must execute sequentially, but instructions that are linked by an anti dependence may execute concurrently.

For example,

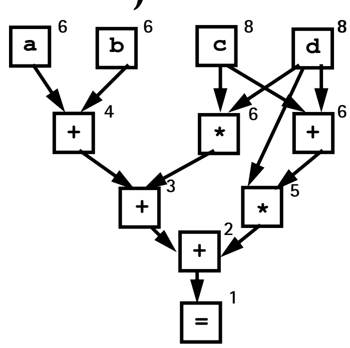
```
add %r1,1,%r2
sub %r3,2,%r1
```

can issue and execute in parallel.

The code scheduling techniques we've studied can be used to schedule machines that can issue 2 or more independent instructions simultaneously.

We select pairs (or triples or n-tuples), verifying (with the Dependence Dag) that they are independent or anti dependent.

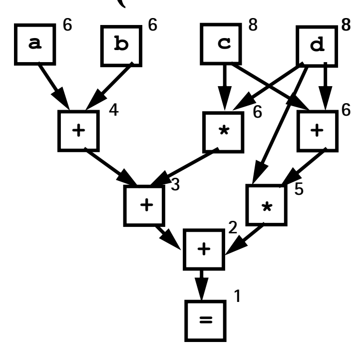
Example: 5 Registers (2 Wide Issue)



1	ld [c], %r1	ld [d], %r2
2	ld [a], %r3	ld [b], %r4
3	smul %r1,%r2,%r5	add %r1,%r2,%r1
4	add %r3,%r4,%r3	smul %r1,%r2,%r1
5	nop	nop
6	add %r3,%r5,%r3	nop
7	add %r3,%r1,%r3	nop
8	st %r3,[a]	nop

We need only 8 cycles rather than 11.

5 Registers (3 Wide Issue)



1	ld [c], %r1	ld [d], %r2	ld [a],%r3
2	ld [b], %r4	nop	nop
3	smul %r1,%r2,%r5	add %r1,%r2,%r1	nop
4	add %r3,%r4,%r3	smul %r1,%r2,%r1	nop
5	nop	nop	nop
6	add %r3,%r5,%r3	nop	nop
7	add %r3,%r1,%r3	nop	nop
8	st %r3,[a]	nop	nop

We still need 8 cycles!

Finding Additional Independent Instructions for Parallel Issue

We can extend the capabilities of processors:

- Out of order execution allows a processor to “search ahead” for independent instructions to launch.
- *But*, since basic blocks are often quite small, the processor may need to accurately predict branches, issuing instructions before the execution path is fully resolved.
- *But*, since branch predictions may be wrong, it will be necessary to “undo” instructions executed speculatively.

Compiler Support for Extended Scheduling

- Trace Scheduling

  Gather sequences of basic blocks together and schedule them as a unit.
- Global Scheduling

  Analyze the control flow graph and move instructions across basic block boundaries to improve scheduling.
- Software Pipelining

  Select instructions from several loop iterations and schedule them together.

Trace Scheduling

Reference:

J. Fisher, “Trace Scheduling: A Technique for Global Microcode Compaction,” IEEE Transactions on Computers, July 1981.

Idea:

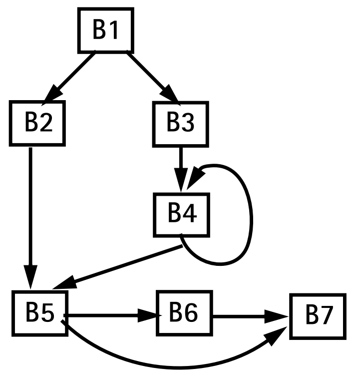
Since basic blocks are often too small to allow effective code scheduling, we will *profile* a program’s execution and identify the most frequently executed paths in a program.

Sequences of contiguous basic blocks on frequently executed paths will be gathered together into *traces*.

Trace

- A sequence of basic blocks (excluding loops) executed together can form a trace.
- A trace will be scheduled as a unit, allowing a larger span of instructions for scheduling.
- A loop can be unrolled or scheduled individually.
- *Compensation code* may need to be added when a branch into, or out of, a trace occurs.

Example



Assume profiling shows that
B1→B3→B4$^+$→B5→B7
is the most common execution path.

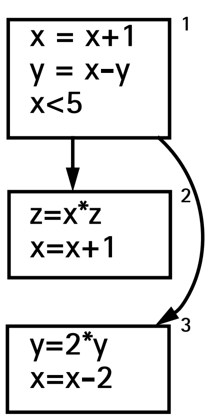
The traces extracted from this path are B1→B3, B4, and B5→B7.

Compensation Code

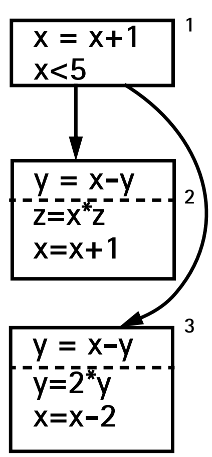
When we move instructions across basic block boundaries within a trace, we may need to add extra instructions that preserve program semantics on paths that enter or leave the trace.

Example

In the previous example, basic block B1 had B2 and B3 as successors, and B1→B3 formed a trace.



Before Scheduling



After Scheduling

Advantages & Disadvantages

- Trace scheduling allows scheduling to span multiple basic blocks. This can significantly increase the effectiveness of scheduling, especially in the context of superscalar processors (which need ILP to be effective).
- Trace Scheduling can also increase code size (because of compensation code). It is also sensitive to the accuracy of trace estimates.

Reading Assignment

- Read pp 367–386 of Allan et. al.'s paper, "Software Pipelining."
  (Linked from the class Web page.)

Global Code Scheduling

- Bernstein and Rodeh approach.
- A *prepass scheduler*
  (does scheduling before register allocation).
- Can move instructions across basic block boundaries.
- Prefers to move instructions that *must* eventually be executed.
- Can move Instructions *speculatively*, possibly executing instructions unnecessarily.

Data & Control Dependencies

When moving instructions across basic block boundaries, we must respect both data dependencies and control dependencies.

Data dependencies specify necessary orderings among instructions that produce a value and instructions that use that value.

Control dependencies determine when (and if) various instructions are executed. Thus an instruction is control dependent on expressions that affect flow of control to that instruction.

Definitions used in Global Scheduling

- Basic Block A *dominates* Basic Block B if and only if A appears on *all* paths to B.
- Basic Block B *postdominates* Basic Block A if and only if B appears on *all* paths from A to an exit point.
- Basic Blocks A and B are *equivalent* if and only if A dominates B and B postdominates A.
- Moving an Instruction from Basic Block B to Basic Block A is *useful* if and only if A and B are equivalent.
- Moving an Instruction from Basic Block B to Basic Block A is *speculative* if B does not postdominate A.

- Moving an Instruction from Basic Block B to Basic Block A requires *duplication* if A does not dominate B.

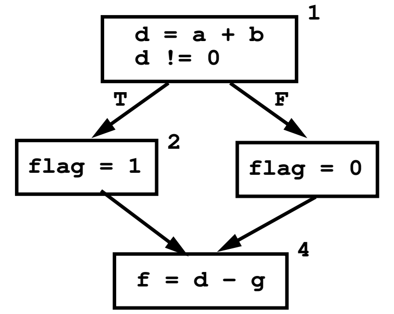
We prefer a move that does not require duplication. (Why?)

The degree of speculation in moving an instruction from one basic block to another can be quantified:

- Moving an Instruction from Basic Block B to Basic Block A is *n-branch* speculative if n conditional branches occur on a path from A to B.

Example

```
d = a + b;
if ( d != 0)
     flag = 1;
else flag = 0;
f = d - g;
```



Blocks 1 and 4 are equivalent.

Moving an Instruction from B2 to B1 (or B3 to B1) is 1-branch speculative.

Moving an Instruction from B4 to B2 (or B4 to B3) requires duplication.

Limits on Code Motion

Assume that pseudo registers are used in generated code (prior to register allocation).

To respect data dependencies:

- A use of a Pseudo Register can't be moved above its definition.
- Memory loads can't be moved ahead of Stores to the same location.
- Stores can't be moved ahead of either loads or stores to the same location.
- A load of a memory location *can* be moved ahead of another load of the same location (such a load may often optimized away by equivalencing the two pseudo registers).

Example (Revisited)

```
block1:
  ld   [a],Pr1
  ld   [b],Pr2
  add  Pr1,Pr2,Pr3  ← Stall
  st   Pr3,[d]
  cmp  Pr3,0
  be   block3
block2:
  mov  1,Pr4
  st   Pr4,[flag]
  b    block4
block3:
  st   0,[flag]
block4:
  ld   [d],Pr5
  ld   [g],Pr6
  sub  Pr5,Pr6,Pr7  ← Stall
  st   Pr7,[f]
```

In B1 and B4, the number of available registers is *irrelevant* in avoiding stalls. There are too few independent instructions in each block.

Global Scheduling Restrictions (in Bernstein/ Rodeh Heuristic)

1. Subprograms are divided into *Regions*. A region is a loop body or the subprogram body without enclosed loops.
2. Regions are scheduled inside-out.
3. Instructions never cross region boundaries.
4. All instructions move "upward" (to earlier positions in the instruction order).
5. The original order of branches is preserved.

Lesser (temporary) restrictions Include:

6. No code duplication.
7. Only 1-branch speculation.
8. No new basic blocks are created or added.

Scheduling Basic Blocks in a CFG

Basic blocks are visited and scheduled in *Topological Order*. Thus all of a block's predecessors are scheduled before it is.

Two levels of scheduling are possible (depending on whether speculative execution is allowed or not):

1. When Basic Block A is scheduled, only Instructions in A and blocks equivalent to A that A dominates are considered.
   (Only "useful" instructions are considered.)

2. Blocks that are immediate successors of those considered in (1) are also considered.
   (This allows 1-branch speculation.)

Candidate Instructions

We first compute the set of basic blocks that may contribute instructions when block A is scheduled. (Either blocks equivalent to A or blocks at most 1-branch speculative.)

An individual Instruction, Inst, in this set of basic blocks may be scheduled in A if:

1. It is located in A.
2. It is in a block equivalent to A and may be moved across block boundaries.
   (Some instructions, like calls, can't be moved.)
3. It is not in a block equivalent to A, but may be scheduled speculatively.
   (Some instructions, like stores, can't be executed speculatively.)

Selecting Instructions to Issue

- A list of "ready to issue" instructions in block A and in bocks equivalent to A (or 1-branch distant from A) is maintained.
- All data dependencies must be satisfied and stalls avoided (if possible).
- N independent instructions are selected, where N is the processor's issue-width.
- But what if more than N instructions are ready to issue?
- Selection is by *Priority*, using two *Scheduling Heuristics*.

Delay Heuristic

This value is computed on a per-basic block basis.

It estimates the worst-case delay (stalls) from an Instruction to the end of the basic block.

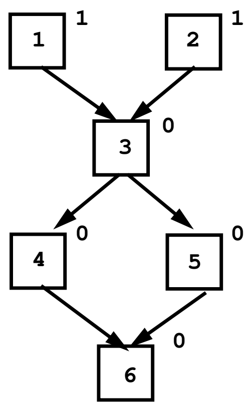
D(I) = 0 if I is a leaf.

Let d(I,J) be the delay if instruction J follows instruction I in the code schedule.

$$D(I) = \max_{J_i \in Succ(I)} (D(J_i)+d(I,J_i))$$

Example of Delay Values

```
block1:
1.  ld    [a],Pr1
2.  ld    [b],Pr2
3.  add   Pr1,Pr2,Pr3
4.  st    Pr3,[d]
5.  cmp   Pr3,0
6.  be    block3
```



(Assume only loads can stall.)

Critical Path Heuristic

This value is also computed on a per-basic block basis.

It estimates how long it will take to execute Instruction I, and all I's successors, assuming unlimited parallelism.

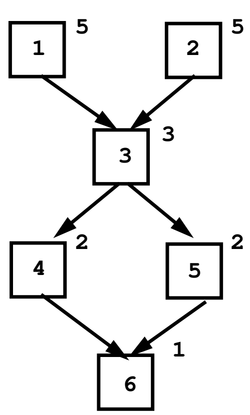
E(I) = Execution time for instruction I (normally 1 for pipelined machines)

CP(I) = E(I) if I is a leaf.

$$CP(I) = E(I) + \max_{J_i \in Succ(I)} (CP(J_i)+d(I,J_i))$$

Example of Critical Path Values

```
block1:
1.  ld    [a],Pr1
2.  ld    [b],Pr2
3.  add   Pr1,Pr2,Pr3
4.  st    Pr3,[d]
5.  cmp   Pr3,0
6.  be    block3
```



Selecting Instructions to Issue

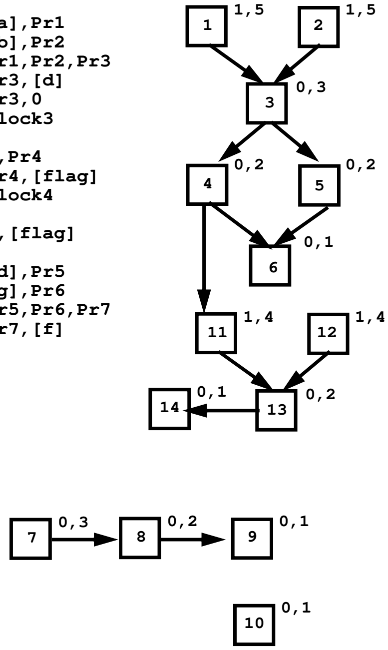
From the Ready Set (instructions with all dependencies satisfied, and which will not stall) use the following priority rules:

1. Instructions in block A and blocks equivalent to A have priority over other (speculative) blocks.
2. Instructions with the highest D values have priority.
3. Instructions with the highest CP values have priority.

These rules imply that we schedule useful instructions before speculative ones, instructions on paths with potentially many stalls over those with fewer stalls, and instructions on critical paths over those on non-critical paths.

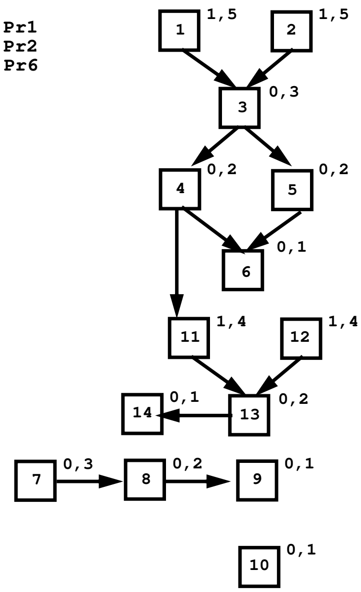
Example

```
block1:
1.   ld    [a],Pr1
2.   ld    [b],Pr2
3.   add   Pr1,Pr2,Pr3
4.   st    Pr3,[d]
5.   cmp   Pr3,0
6.   be    block3
block2:
7.   mov   1,Pr4
8.   st    Pr4,[flag]
9.   b     block4
block3:
10.  st    0,[flag]
block4:
11.  ld    [d],Pr5
12.  ld    [g],Pr6
13.  sub   Pr5,Pr6,Pr7
14.  st    Pr7,[f]
```



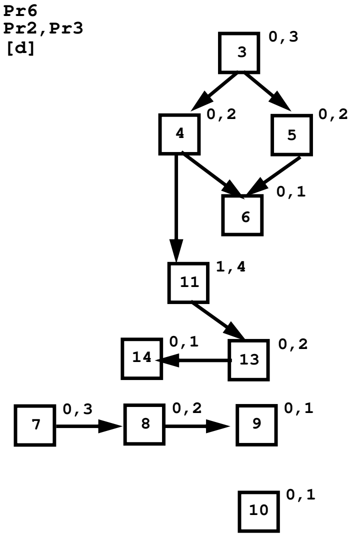
We'll schedule without speculation; highest D values first, then highest CP values.

```
block1:
1.   ld    [a],Pr1
2.   ld    [b],Pr2
12.  ld    [g],Pr6
```



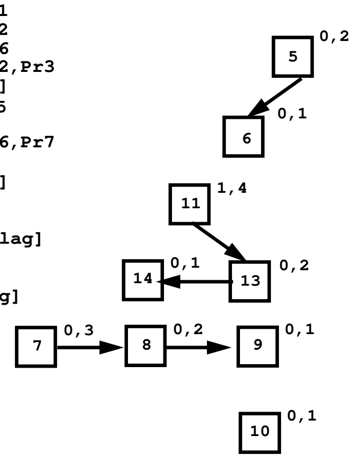
Next, come Instructions 3 and 4.

```
block1:
1.   ld    [a],Pr1
2.   ld    [b],Pr2
12.  ld    [g],Pr6
3.   add   Pr1,Pr2,Pr3
4.   st    Pr3,[d]
```



Now 11 can issue (D=1), followed by 5, 13, 6 and 14. Block B4 is now empty, so B3 and B4 are scheduled.

```
block1:
1.   ld    [a],Pr1
2.   ld    [b],Pr2
12.  ld    [g],Pr6
3.   add   Pr1,Pr2,Pr3
4.   st    Pr3,[d]
11.  ld    [d],Pr5
5.   cmp   Pr3,0
13.  sub   Pr5,Pr6,Pr7
6.   be    block3
14.  st    Pr7,[f]
block2:
7.   mov   1,Pr4
8.   st    Pr4,[flag]
9.   b     block4
block3:
10.  st    0,[flag]
block4:
```



There are no stalls. In fact, if we equivalence `Pr3` and `Pr5`, Instruction 11 can be removed.

Reading Assignment

- Read Section 15.5, "Automatic Instruction Selection," from Chapter 15.
- Read Pelegri-Llopart and Graham's paper, "Optimal Code Generation from Expression Trees."
  (Linked from the class Web page.)
- Read Fraser, Henry and Proebsting's paper, "BURG--Fast Optimal Instruction Selection and Tree Parsing."
  (Linked from the class Web page.)

Software Pipelining

Often loop bodies are too small to allow effective code scheduling. But loop bodies, being "hot spots," are exactly where scheduling is most important.

Consider

```
void f (int a[],int last) {
  for (p=&a[0];p!=&a[last];p++)
     (*p)++;
}
```

The body of the loop might be:

```
L: ld   [%g3],%g2
   nop
   add  %g2,1,%g2
   st   %g2,[%g3]
   add  %g3,4,%g3
   cmp  %g3,%g4
   bne  L
   nop
```


Scheduling this loop body in isolation is ineffective—each instruction depends upon its immediate predecessor.

So we have a loop body that takes 8 cycles to execute 6 "core" instructions.

We could unroll the loop body, but for how many iterations? What if the loop ends in the "middle" of an expanded loop body? Will extra registers be a problem?

In this case *software pipelining* offers a nice solution. We expand the loop body *symbolically*, intermixing instructions from several iterations. Instructions can overlap, increasing parallelism and forming a "tighter" loop body:

```
   ld   [%g3],%g2
   nop
   add  %g2,1,%g2
L: st   %g2,[%g3]
   add  %g3,4,%g3
   ld   [%g3],%g2
   cmp  %g3,%g4
   bne  L
   add  %g2,1,%g2
```

Now the loop body is ideal—exactly 6 instructions. Also, no extra registers are needed!

But, we do "overshoot" the end of the loop a bit, loading one element past the exit point. (How serious is this?)

Key Insight of Software Pipelining

Software pipelining exploits the fact that a loop of the form $\{A\ B\ C\}^n$, where **A**, **B** and **C** are individual instructions, and **n** is the iteration count, is equivalent to $A\ \{B\ C\ A\}^{n-1}\ B\ C$ and is also equivalent to $A\ B\ \{C\ A\ B\}^{n-1}\ C$.

Mixing instructions from several iterations may increase the effectiveness of code scheduling, and may perhaps allow for more parallel execution.

Software Pipelining is Hard

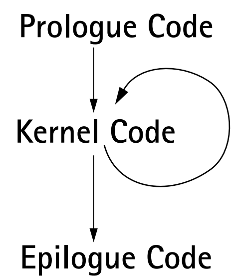
In fact, it is NP-complete:

Hsu and Davidson, "Highly concurrent scalar processing," 13th ISCA (1986).

The Iteration Interval

We seek to initiate the next iteration of a loop as soon as possible, squeezing each iteration of the loop body into as few machine cycles as possible.

The general form of a software pipelined loop is:



The prologue code "sets up" the main loop, and the epilogue code "cleans up" after loop termination. Neither the prolog nor the epilogue need be optimized, since they execute only once.

Optimizing the kernel is key in software pipelining. The kernel's execution time (in cycles) is called the *initiation interval (II)*; it measures how quickly the next iteration of a loop can start.

We want the smallest possible initiation interval. Determining the smallest viable II is itself NP-complete. Because of parallel issue and execution in superscalar and multiple issue processors, very small II values are possible (even less than 1!)

Factors that Limit the Size of the Initiation Interval

We want the initiation interval to be as small as possible. Two factors limit how small the II can become:

- Resource Constraints
- Dependency Constraints

Resource Constraints

A small II normally means that we are doing steps of several iterations simultaneously. The number of registers and functional units (that execute instructions) can become limiting factors of the size of II.

For example, if a loop body contains 4 floating point operations, and our processor can issue and execute no more than 2 floating point operations per cycle, then the loop's II can't be less than 2.

Dependency Constraints

A loop body can often contain a *loop-carried dependence*. This means one iteration of a loop depends on values computed in an earlier iteration. For example, in

```
void f (int a[]) {
  for (i=1;i<1000;i++)
     a[i]=(a[i-1]+a[i])/2;
}
```

there is a loop carried dependence from the use of `a[i-1]` to the computation of `a[i]` in the previous iteration. This means the computation of `a[i]` can't begin until the computation of `a[i-1]` is completed.

Let's look at the code that might be generated for this loop:


```
f:
    mov   %o0, %o2        !a in %o2
    mov   1, %o1          !i=1 in %o1
L:
    sll   %o1, 2, %o0     !i*4 in %o0
    add   %o0, %o2, %g2   !&a[i] in %g2
☛   ld    [%g2-4], %g2    !a[i-1] in %g2
    ld    [%o2+%o0], %g3  !a[i] in %g3
☛   add   %g2, %g3, %g2   !a[i-1]+a[i]
☛   srl   %g2, 31, %g3    !s=0 or 1=sign
☛   add   %g2, %g3, %g2   !a[i-1]+a[i]+s
☛   sra   %g2, 1, %g2     !a[i-1]+a[i]/2
    add   %o1, 1, %o1     !i++
    cmp   %o1, 999
    ble   L
☛   st    %g2, [%o2+%o0]  !store a[i]
    retl
    nop
```

The 6 marked instructions form a cyclic dependency chain from a use of `a[i-1]` to its computation (as `a[i]`) in the previous cycle. This cycle means that the loop's II can never be less than 6.

Modulo Scheduling

There are many approaches to software pipelining. One of the simplest, and best known, is *modulo scheduling*. Modulo scheduling builds upon the postpass basic block schedulers we've already studied.

First, we estimate the II of the loop we will create. How?

We can compute the minimum II based on resource considerations (II_{res}) and the minimum II based on cyclic loop-carried dependencies (II_{dep}). Then $\max(II_{res}, II_{dep})$ is a reasonable estimate of the best possible II. We'll try to build a loop with a kernel size of II. If this fails, we'll try II+1, II+2, etc.

In modulo scheduling we'll schedule instructions one by one, using the dependency dag and whatever heuristic we prefer to choose among multiple roots.

Now though, if we place an instruction at cycle c (many independent instructions may execute in the same cycle), then we'll place additional copies of the instruction at cycle c+II, c+2*II, etc.

Placement must respect dependency constraints and resource limits at all positions. We consider placements only until a kernel (of size II) forms. The kernel must begin before cycle s-1, where s is the size of the loop body (in instructions). The loop's conditional branch is placed *after* the kernel is formed.

If we can't form a kernel of size II (because of dependency or resource conflicts), we increase II by 1 and try again. At worst, we get a kernel equal in size to the original loop body, which guarantees that the modulo scheduler eventually terminates.

Depending on how many iterations are intermixed in the kernel, the loop termination condition may need to be adjusted (since the initial and final iterations may appear as part of the loop prologue and epilogue).

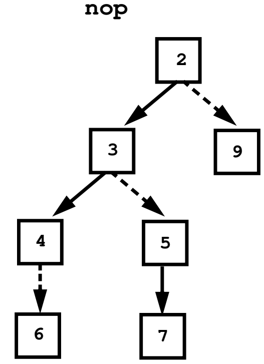
Example

Consider the following simple function which adds an array index to each element of an array and copies the results into a second array:

```
void f (int a[],int b[]) {
  t1 = &a[0];
  t2 = &b[0];
  for (i=0;i<1000;i++,t1++,t2++)
     *t1 = *t2 + i;
}
```

The code for `f` (compiled as a leaf procedure) is:


```
1.   f:   mov    0, %g3
2.   L:   ld     [%o1], %g2
3.        add    %g3, %g2, %g4
4.        st     %g4, [%o0]
5.        add    %g3, 1, %g3
6.        add    %o0, 4, %o0
7.        cmp    %g3, 999
8.        ble    L
9.        add    %o1, 4, %o1
10.       retl
11.       nop
```



Dashed arcs are anti dependencies.

We'll software pipeline the loop body, excluding the conditional branch (which is placed after the loop kernel is formed).

This loop body contains 2 loads/ stores, 5 arithmetic and logical operations (including the compare) and one conditional branch.

Let's assume the processor we are compiling for has 1 load/store unit, 3 arithmetic/logic units, and 1 branch unit. That means the processor can (ideally) issue and execute simultaneously 1 load or store, 3 arithmetic and logic instructions, and 1 branch. Thus its maximum issue width is 5. (Current superscalars have roughly this capability.)

Considering resource requirements, we will need at least two cycles to process the contents of the loop body. There are no loop-carried dependencies.

Thus we will estimate this loop's best possible Initiation Interval to be 2.

Since the only instruction that can stall is the root of the dependency dag, we'll schedule using estimated critical path length, which is just the node's height in the tree. Hence we'll schedule the nodes in the order: 2,3,4,5,6,7,9.

We'll schedule all instructions in a legal execution order (respecting dependencies), and we'll try to choose as many instructions as possible to execute in the same cycle.

Starting with the root, instruction 2, we schedule it at cycles 1, 3 (=1+II), 5 (=1+2*II):

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          ld    [%o1], %g2
4.
5.          ld    [%o1], %g2
```

No conflicts so far, since each of the loads starts an independent iteration.

We'll schedule instruction 3 next. It must be placed at cycles 3, 5 and 7 since it uses the result of the load.

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
4.
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
6.
7.          add   %g3, %g2, %g4
```

Note that in cycles 3 and 5 we use the current value of `%g2` *and* initiate a load into `%g2`.

Instruction 4 is next. It uses the result of the add we just scheduled, so it is placed at cycles 4 and 6.

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
4.          st    %g4, [%o0]
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
6.          st    %g4, [%o0]
7.          add   %g3, %g2, %g4
```


Instruction 5 is next. It is anti dependent on instruction 3, so we can place it in the same cycles that 3 uses (3, 5 and 7).

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
3.          add   %g3, 1, %g3
4.          st    %g4, [%o0]
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
5.          add   %g3, 1, %g3
6.          st    %g4, [%o0]
7.          add   %g3, %g2, %g4
7.          add   %g3, 1, %g3
```


Instruction 6 is next. It is anti dependent on instruction 4, so we can place it in the same cycles that 4 uses (4 and 6).

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
3.          add   %g3, 1, %g3
4.          st    %g4, [%o0]
4.          add   %o0, 4, %o0
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
5.          add   %g3, 1, %g3
6.          st    %g4, [%o0]
6.          add   %o0, 4, %o0
7.          add   %g3, %g2, %g4
7.          add   %g3, 1, %g3
```


Next we place instruction 7. It uses the result of instruction 5 (`%g3`), so it is placed in the cycles following instruction 5 (4 and 6).

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
3.          add   %g3, 1, %g3
4.          st    %g4, [%o0]
4.          add   %o0, 4, %o0
4.          cmp   %g3, 999
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
5.          add   %g3, 1, %g3
6.          st    %g4, [%o0]
6.          add   %o0, 4, %o0
7.          add   %g3, %g2, %g4
7.          add   %g3, 1, %g3
7.          cmp   %g3, 999
```

Finally we place instruction 9. It is anti dependent on instruction 2 so it is placed in the same cycles as instruction 3 (1, 3 and 5).

```
cycle       instruction
1.           ld    [%o1], %g2
1.           add   %o1, 4, %o1
3.           add   %g3, %g2, %g4
3.           ld    [%o1], %g2
3.           add   %o1, 4, %o1
3.           add   %g3, 1, %g3
4.           st    %g4, [%o0]
4.           add   %o0, 4, %o0
4.           cmp   %g3, 999
5.           add   %g3, %g2, %g4
5.           ld    [%o1], %g2
5.           add   %o1, 4, %o1
5.           add   %g3, 1, %g3
6.           st    %g4, [%o0]
6.           add   %o0, 4, %o0
7.           add   %g3, %g2, %g4
7.           add   %g3, 1, %g3
7.           cmp   %g3, 999
```


We look for a 2 cycles kernel that contains all 7 instructions of the loop body that we have scheduled. We also want a kernel that sets the condition code (via the cmp) during its first cycle so that it can be tested during its second (and final) cycle. Cycles 4 and 5 meet these criteria, and will form our kernel.

We place the conditional branch just before the last instruction in cycle 5 (to give the conditional branch a useful instruction for its delay slot).

We now have:

```
cycle       instruction
1.           ld    [%o1], %g2
1.           add   %o1, 4, %o1
3.           add   %g3, %g2, %g4
3.           ld    [%o1], %g2
3.           add   %o1, 4, %o1
3.           add   %g3, 1, %g3
4.      L:   st    %g4, [%o0]
4.           add   %o0, 4, %o0
4.           cmp   %g3, 999
5.           add   %g3, %g2, %g4
5.           ld    [%o1], %g2
5.           add   %o1, 4, %o1
5.           ble   L
5.           add   %g3, 1, %g3
6.           st    %g4, [%o0]
6.           add   %o0, 4, %o0
7.           add   %g3, %g2, %g4
7.           add   %g3, 1, %g3
7.           cmp   %g3, 999
```


A couple of final issues must be dealt with:

- Does the iteration count need to be changed?
  In this case no, since the final valid value of `i`, 999, is used to compute `%g4` in cycle 5, before the loop exits.
- What instructions do we keep as the loop's epilogue?
  None! Instructions past the kernel aren't needed since they are part of future iterations (past `i==999`)which aren't needed or wanted.
- Note that `b[1000]` and `b[1001]` are "touched" even though they are never used. This is probably OK as long as arrays aren't placed at the very end of a page or segment.

Our final loop is:

```
cycle       instruction
1.          ld    [%o1], %g2      !N0
1.          add   %o1, 4, %o1     !N0
3.          add   %g3, %g2, %g4   !N0
3.          ld    [%o1], %g2      !N1
3.          add   %o1, 4, %o1     !N1
3.          add   %g3, 1, %g3     !N0
4.      L:  st    %g4, [%o0]      !N0
4.          add   %o0, 4, %o0     !N0
4.          cmp   %g3, 999        !N0
5.          add   %g3, %g2, %g4   !N1
5.          ld    [%o1], %g2      !N2
5.          add   %o1, 4, %o1     !N2
5.          ble   L               !N0
5.          add   %g3, 1, %g3     !N1
```

This is very efficient code—we use the full parallelism of the processor, executing 5 instructions in cycle 5 and 8 instructions in just 2 cycles. All resource limitations are respected.

False Dependencies & Loop Unrolling

A limiting factor in how “tightly” we can software pipeline a loop is reuse of registers and the false dependencies reuse induces.

Consider the following simple function that copies array elements:

```
void f (int a[],int b[], int lim) {
  for (i=0;i<lim;i++)
     a[i]=b[i];
}
```

The loop that is generated takes 3 cycles:

```
cycle       instruction
1.      L:  ld     [%g3+%o1], %g2
1.          addcc  %o2, -1, %o2
3.          st     %g2, [%g3+%o0]
3.          bne    L
3.          add    %g3, 4, %g3
```


We'd like to tighten the iteration interval to 2 or less. One cycle is unlikely, since doing a load and a store in the same cycle is problematic (due to a possible dependence through memory).

If we try to use modulo scheduling, we can't put a second copy of the load in cycle 2 because it would overwrite the contents of the first load. A load in cycle 3 will clash with the store.

The solution is to unroll the loop into two copies, using different registers to hold the contents of the load and the current offset into the arrays.

The use of a “count down” register to test for loop termination is helpful,

since it allows an easy exit from the middle of the loop.

With the renaming of the registers used in the two expanded iterations, scheduling to “tighten” the loop is effective.

After expansion we have:

```
cycle       instruction
1.      L:  ld     [%g3+%o1], %g2
1.          addcc  %o2, -1, %o2
3.          st     %g2, [%g3+%o0]
3.          beq    L2
3.          add    %g3, 4, %g4
4.          ld     [%g4+%o1], %g5
4.          addcc  %o2, -1, %o2
6.          st     %g5, [%g4+%o0]
6.          bne    L
6.          add    %g4, 4, %g3
        L2:
```

We still have 3 cycles per iteration, because we haven't scheduled yet.

Now we can move the increment of **%g3** (into **%g4**) above other uses of **%g3**. Moreover, we can move the load into **%g5** *above* the store from **%g2** (*if the load and store are independent*):

```
cycle       instruction
1.     L:    ld    [%g3+%o1], %g2
1.           addcc %o2, -1, %o2
1.           add   %g3, 4, %g4
2.           ld    [%g4+%o1], %g5
3.           st    %g2, [%g3+%o0]
3.           beq   L2
3.           addcc %o2, -1, %o2
4.           st    %g5, [%g4+%o0]
4.           bne   L
4.           add   %g4, 4, %g3
      L2:
```

We can normally test whether **%g4+%o1** and **%g3+%o0** can be equal at compile-time, by looking at the actual array parameters. (Can **&a[0] == &b[1]**?)

Automatic Instruction Selection

Besides register allocation and code scheduling, a code generator must also do *Instruction Selection*.

For CISC (Complex Instruction Set Computer) Architectures, like the Intel x86, DEC Vax, and many special purpose processors (like Digital Signal Processors), instruction selection is often *challenging* because so many choices exist.

In the Vax, for example, one, two and three address instructions exist. Each address may be a register, memory location (with or without indexing), or an immediate operand.

For RISC (Reduced Instruction Set Computer) Processors, instruction formats and addressing modes are far more limited.

Still, it is necessary to handle immediate operands, commutative operands and special case null operands (add of 0 or multiply of 1).

Moreover, automatic instruction selection supports *automatic retargeting* of a compiler to a new or extended instruction set.

Tree-Structured Intermediate Representations

For purposes of automatic code generation, it is convenient to translate a source program into a *Low-level, Tree-Structured IR*.

This representation exposes translation details (how locals are accessed, how conditionals are translated, etc.) without assuming a particular instruction set.

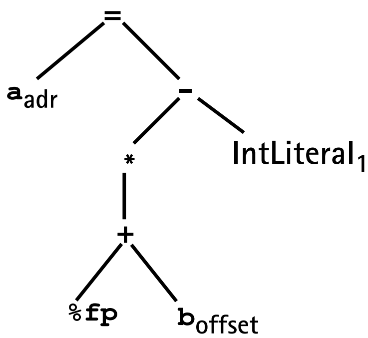
In a low-level, tree-structured IR, leaves are registers or bit-patterns and internal nodes are machine-level primitives, like load, store, add, etc.

Example

Let's look at how

`a = b - 1;`

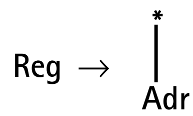
is represented, where `a` is a global integer variable and `b` is a local (frame allocated) integer variable.



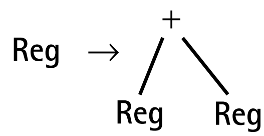
Representation of Instructions

Individual instructions can be represented as trees, rooted by the operation they implement.

For example:

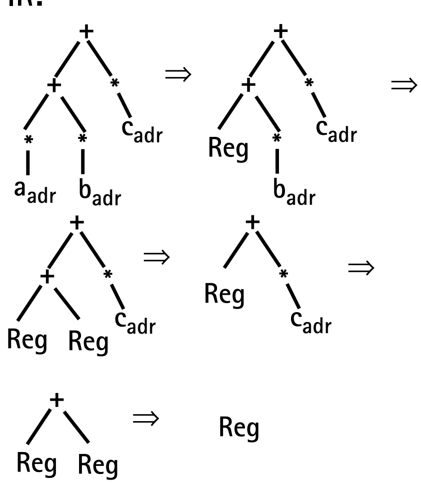


This is an instruction that loads a register with the value at an absolute address.



This is an instruction that adds the contents of two registers and stores the sum into a third register.

Using the above pair of instruction definitions, we can repeatedly match instructions in the following program IR:



Each match of an instruction pattern can have the side-effect of generating an instruction:

```
ld   [a],%R1
ld   [b],%R2
add  %R1,%R2,%R3
ld   [c],%R4
add  %R3,%R4,%R5
```

Registers can be allocated on-the-fly as Instructions are generated or instructions can be generated using pseudo-registers, with a subsequent register allocation phase.

Using this view of instruction selection, choosing instructions involves finding a *cover* for an IR tree using Instruction Patterns.

Any cover is a *valid* translation.

Tree Parsing vs. String Parsing

This process of selecting instructions by matching instruction patterns is very similar to how strings are parsed using Context-free Grammars.

We repeatedly identify a sub-tree that corresponds to an instruction, and simplify the IR-tree by replacing the instruction sub-tree with a nonterminal symbol. The process is repeated until the IR-tree is reduced to a single nonterminal.

The theory of reducing an IR-tree using rewrite rules has been studied as part of BURS (Bottom-Up Rewrite Systems) Theory by Pelegri-Llopart and Graham.

Automatic Instruction Selection Tools

Just as tools like Yacc and Bison automatically generate a string parser from a specification of a Context-free Grammar, there exist tools that will automatically generate a tree-parser from a specification of tree productions.

Two such tools are BURG (Bottom Up Rewrite Generator) and IBURG (Interpreted BURG). Both automatically generate parsers for tree grammars using BURS theory.

Least-Cost Tree Parsing

BURG (and IBURG) *guarantee* to find a cover for an input tree (if one exists).

But tree grammars are usually *very* ambiguous.

Why?—Because there is usually more than one code sequence that can correctly implement a given IR-tree.

To deal with ambiguity, BURG and IBURG allow each instruction pattern (tree production) to have a *cost*.

This cost is typically the size or execution time for the corresponding target-machine instructions.

Using costs, BURG (and IBURG) not only guarantee to find a cover, but also a *least-cost cover*.

This means that when a generated tree-parser is used to cover (and thereby translate) an IR-Tree, the *best possible* code sequence is guaranteed.

If more than one least-cost cover exists, an arbitrary choice is made.

Using BURG to Specify Instruction Selection

We'll need a tree grammar to specify possible partial covers of a tree.

For simplicity, BURG requires that all tree productions be of the form

$A \rightarrow b$

(where b is a single terminal symbol)

or

$A \rightarrow Op(B,C, ...)$

(where Op is a terminal that is a subtree root and B,C, ... are non-terminals)

$A \rightarrow Op(B,C, ...)$ denotes

```
    Op
   / | \
  B  C  ...
```


All tree grammars can be put into this form by adding new nonterminals and productions as needed.

We must specify terminal symbols (leaves and operators in the IR-Tree) and nonterminals that are used in tree productions.

Example

A subset of a SPARC instruction selector.

Terminals

Leaf Nodes

int32	(32 bit integer)
s13	(13 bit signed integer)
r	(0-31, a register name)

Operator Nodes

*	(unary indirection)
-	(binary minus)
+	(binary addition)
=	(binary assignment)

Nonterminals

UInt	(32 bit unsigned integer)
Reg	(Loaded register value)
Imm	(Immediate operand)
Adr	(Address expression)
Void	(Null value)

Productions

Rule #	Production	Cost	SPARC Code
R0	UInt → Int32	0	
R1	Reg → r	0	
R2	Adr → r	0	
R3	Adr → + (Reg, Imm)	0	
R4	Imm → s13	0	
R5	Reg → s13	1	`mov s13,Reg`
R6	Reg → int32	2	`sethi %hi(int32),%g1` `or %g1, %lo(int32),Reg`
R7	Reg → − (Reg, Reg)	1	`sub Reg,Reg,Reg`

Rule #	Production	Cost	SPARC Code
R8	Reg → − (Reg, Imm)	1	`sub Reg,Imm,Reg`
R9	Reg → * (Adr)	1	`ld [Adr],Reg`
R10	Void → = (UInt, Reg)	2	`sethi %hi(UInt),%g1` `st Reg, [%g1+%lo(Uint)]`

Reading Assignment

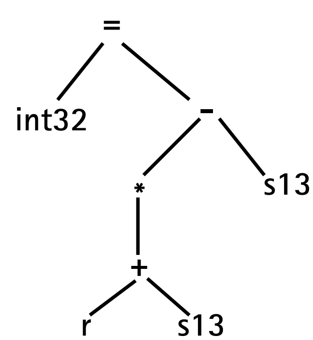
- Read "Optimal Spilling for CISC Machines with Few Registers," by Appel and George. (Linked from the class Web page.)

Example

Let's look at instruction selection for

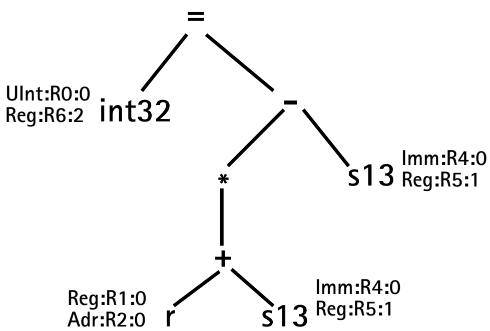
```
a = b - 1;
```

where `a` is a global int, accessed with a 32 bit address and `b` is a local int, accessed as an offset from the frame pointer.



We match tree nodes *bottom-up*. Each node is labeled with the nonterminals it can be reduced to, the production used to produce the nonterminal, and the cost to generate the node (and its children) from the nonterminal.

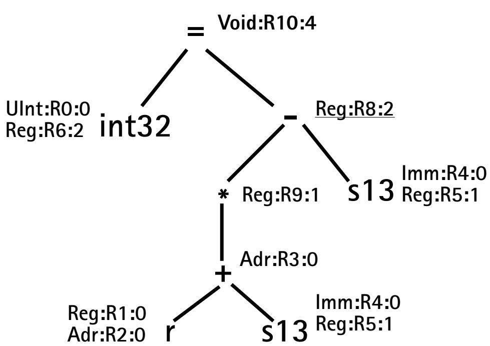
We match leaves first:



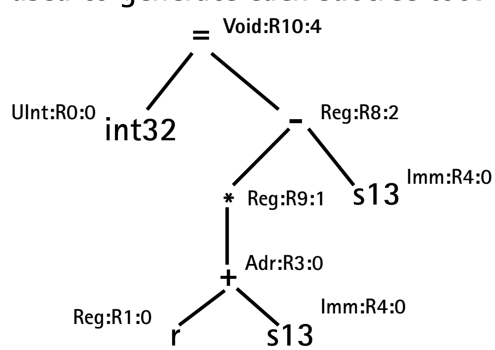
We now work *upward*, considering operators whose children have been labeled. Again, if an operator can be generated by a nonterminal, we mark the operator with the nonterminal, the production used to generate the operator, and the total cost (including the cost to generate all children).

If a nonterminal can generate the operator using more than one production, the *least-cost* derivation is chosen.

When we reach the root, the nonterminal with the lowest overall cost is used to generate the tree.

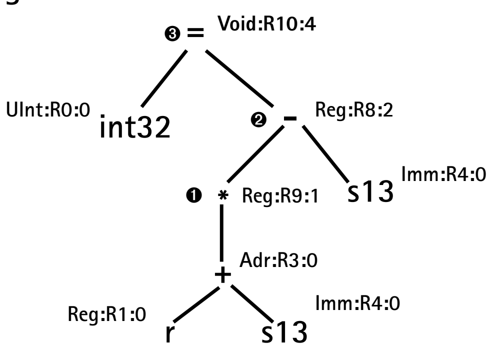


Note that once we know the production used to generate the root of the tree, we know the productions used to generate each subtree too:



We generate code by doing a depth-first traversal, generating code for a production after all the production's children have been processed.

We need to do register allocation too; for our example, a simple on-the-fly generator will suffice.




```
❶  ld     [%fp+b],%l0
❷  sub    %l0,1,%l0
❸  sethi  %hi(a),%g1
   st     %l0,[%g1+%lo(a)]
```


Had we translated a slightly difference expression,

```
a = b - 1000000;
```

we would *automatically* get a different code sequence (because 1000000 is an int32 rather than an s13):

```
ld    [%fp+b],%l0
sethi %hi(1000000),%g1
or    %g1,%lo(1000000),%l1
sub   %l0,%l1,%l0
sethi %hi(a),%g1
st    %l0,[%g1+%lo(a)]
```


Adding New Rules

Since instruction selectors can be automatically generated, it's easy to add "extra" rules that handle optimizations or special cases.

For example, we might add the following to handle addition of a left immediate operand or subtraction of 0 from a register:

Rule #	Production	Cost	SPARC Code
R11	Reg → + (Imm Reg)	1	`add Reg,Imm,Reg`
R12	Reg → − (Reg Zero)	0	

Improving the Speed of Instruction Selection

As we have presented it, instruction selection looks rather slow—for each node in the IR tree, we must match productions, compare costs, and select least-cost productions.

Since compilers routinely generate program with tens or hundreds of thousands of instructions, doing a lot of computation to select one instruction (even if it's the *best* instruction) could be too slow.

Fortunately, this need not be the case.

Instruction selection using BURS can be made *very* fast.

Adding States to BURG

We can *precompute* a set of *states* that represent possible labelings on IR tree nodes. A table of node names and subtree states then is used to select a node's state. Thus labeling becomes nothing more than repeated table lookup.

For example, we might create a state s0 that corresponds to the labeling {Reg:R1:0, Adr:R2:0}.

A state selection function, *label*, defines label(r) = s0. That is, whenever r is matched as a leaf, it is to be labeled with s0.

If a node is an operator, label uses the name of the operator and the labeling

assigned to its children to choose the operator's label. For example,

label(+,s0,s1)=s2

says that a + with children labeled as s0 and s1 is to be labeled as s2.

In theory, that's all there is to building a fast instruction selector.

We generate possible labelings, encode them as states, and table all combinations of labelings.

But,

how do we know the set of possible labelings is even finite?

In fact, it isn't!

Normalizing Costs

It is possible to generate states that are identical except for their costs.

For example, we might have

s1 = {Reg:R1:0, Adr:R2:0},

s2 = {Reg:R1:1, Adr:R2:1},

s3 = {Reg:R1:2, Adr:R2:2}, etc.

Here an important insight is needed— the *absolute* costs included in states aren't really essential. Rather *relative* costs are what is important. In s1, s2, and s3, Reg and Adr have the same cost. Hence the same decision in choosing between Reg and Adr will be made in all three states.

We can limit the number of states needed by *normalizing* costs within states so that the lowest cost choice is always 0, and other costs are differences (deltas) from the lowest cost choice.

This observation keeps costs bounded within states (except for pathologic cases).

Using additional techniques to further reduce the number of states needed, and the time needed to generate them, fast and compact BURS instruction selectors are achievable. See

"Simple and Efficient BURS Table Generation," T. Proebsting, 1992 PLDI Conference.

Example

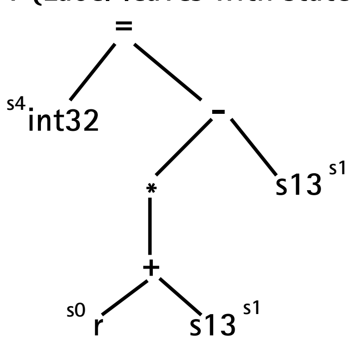
State	Meaning
s0	{Reg:R1:0, Adr:R2:0}
s1	{Imm:R4:0, Reg:R5:1}
s2	{adr:R3:0}
s3	{Reg:R9:0}
s4	{UInt:R0:0}
s5	{Reg:R8:0}
s6	{Void:R10:0}
s7	{Reg:R7:0}

Node	Left Child	Right Child	Result
r			s0
s13			s1
int32			s4
+	s0	s1	s2
*	s2		s3
-	s3	s1	s5
-	s1	s3	s7
=	s4	s5	s6

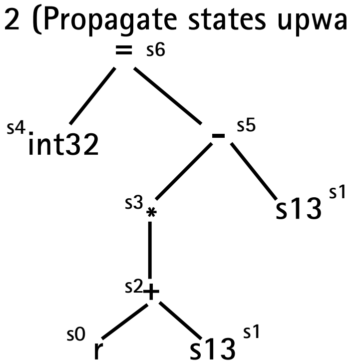
We start by looking up the state assigned to each leaf. We then work upward, choosing the state of a parent based on the parent's kind and the states assigned to the children. These are all table lookups, and hence very fast.

At the root, we select the nonterminal and production based on the state assigned to the root (any entry with 0 cost). Knowing the production used at the root tells us the nonterminal used at each child. Each state has only one entry per nonterminal, so knowing a node's state and the nonterminal used to generate it immediately tells us the production used. Hence identifying the production used for each node is again very fast.

Step 1 (Label leaves with states):

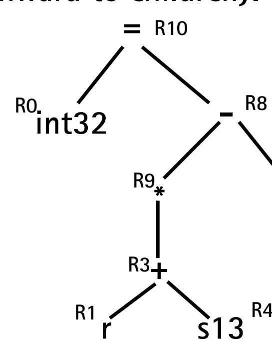


Step 2 (Propagate states upward):



Step 3 (Choose production used at root): R10.

Step 4 (Propagate productions used downward to children):



Code Generation for x86 Machines

The x86 presents several special difficulties when generating code.

- There are only 8 architecturally visible registers, and only 6 of these are allocatable. Deciding what values to keep in registers, and for how long, is a difficult, but crucial, decision.
- Operands may be addressed directly from memory in some instructions. Such instructions avoid using a register, but are longer and add to I-cache pressure.

In "Optimal Spilling for CISC Machines with Few Registers," Appel and George address both of these difficulties.

They use Integer Programming techniques to directly and optimally solve the crucial problem of deciding which live ranges are to be register-resident at each program point. Stores and loads are automatically added to split long live ranges.

Then a variant of Chaitin-style register allocation is used to assign registers to live ranges chosen to be register-resident.

The presentation of this paper, at the 2001 PLDI Conference, is at

`www.cs.wisc.edu/~fischer/cs701/cisc.spilling.pdf`

Reading Assignment

- Read pages 1-30 of "Automatic Program Optimization," by Ron Cytron. (Linked from the class Web page.)

Optimistic Coalescing

Given R allocatable registers, Appel and George guarantee that no more than R live ranges are marked as register resident.

This doesn't always guarantee that an R coloring is possible.

Consider the following program fragment:

```
x=0;
while (...) {
  y = x+1;
  print(x);
  z = y+1;
  print(y);
  x = z+1;
  print(z);
}
```

At any given point in the loop body only 2 variables are live, but 3 registers are needed (`x` interferes with `y`, `y` interferes with `z` and `z` interferes with `x`).

We know that we have enough registers to handle all live ranges marked as register-resident, but we may need to “shuffle” register allocations at certain points.

Thus at one point x might be allocated R1 and at some other point it might be placed in R2. Such shuffling implies register to register copies, so we'd like to minimize their added cost.

Appel and George suggest allowing changes in register assignments between program points. This is done by creating multiple variable names for a live range (x_1, x_2, x_3, ...), one for each program point. Variables are connected by assignments between points. Using coalescing, it is expected that most of the assignments will be optimized away.

Using our earlier example, we have the following code with each variable expanded into 3 segments (one for each assignment). Copies of dead variables are removed to simplify the example:


```
x3=0;
while (...) {
  x1 = x3;
  y1 = x1+1;
  print(x1);
  y2 = y1;
  z2 = y2+1;
  print(y2);
  z3 = z2;
  x3 = z3+1;
  print(z3);
}
```

Now a 2 coloring is possible:

x_1: R1, y_1: R2

z_2: R1, y_2: R2

z_3: R1, x_3: R2

(and only $x_1 = x_3$ is retained).

Appel and George found that iterated coalescing wasn't effective (too many copies, most of which are useless).

Instead they recommend *Optimistic Coalescing*. The idea is to first do Chaitin-style reckless coalescing of all copies, even if colorability is impaired.

Then we do graph coloring register allocation, using the cost of copies as the “spill cost.” As we select colors, a coalesced node that can't be colored is simply split back to the original source and target variables. Since we always limit the number of live ranges to the number of colors, we know the live ranges must be colorable (with register to register copies sometimes needed).

Using our earlier example, we initially merge x_1 and x_3, y_1 and y_2, z_2 and z_3. We already know this can't be colored with two registers. All three pairs have the same costs, so we arbitrarily stack x_1-x_3, then y_1-y_2 and finally z_2-z_3.

When we unstack, z_2-z_3 gets R1, and y_1-y_2 gets R2. x_1-x_3 must be split back into x_1 and x_3. x_1 interferes with y_1-y_2 so it gets R1. x_3 interferes with z_2-z_3 so it gets R2, and coloring is done.

x_1: R1, y_1: R2

z_2: R1, y_2: R2

z_3: R1, x_3: R2

Data Flow Frameworks

- Data Flow Graph:

  Nodes of the graph are basic blocks or individual instructions.

  Arcs represent flow of control.

  Forward Analysis:

  Information flow is the same direction as control flow.

  Backward Analysis:

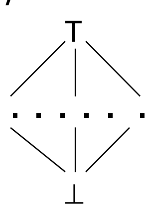
  Information flow is the opposite direction as control flow.

  Bi-directional Analysis:

  Information flow is in both directions. (Not too common.)

- Meet Lattice

  Represents solution space for the data flow analysis.



- Meet operation
  (And, Or, Union, Intersection, etc.)

  Combines solutions from predecessors or successors in the control flow graph.

- Transfer Function

  Maps a solution at the top of a node to a solution at the end of the node (forward flow)
  or
  Maps a solution at the end of a node to a solution at the top of the node (backward flow).

Example: Available Expressions

This data flow analysis determines whether an expression that has been previously computed may be reused.

Available expression analysis is a forward flow problem—computed expression values flow forward to points of possible reuse.

The best solution is True—the expression may be reused.

The worst solution is False—the expression may not be reused.

The Meet Lattice is:

T (Expression is Available)

|

F (Expression is Not Available)

As initial values, at the top of the start node, nothing is available. Hence, for a given expression,

$\text{AvailIn}(b_0) = F$

We choose an expression, and consider all the variables that contribute to its evaluation.

Thus for e_1=a+b−c, a, b and c are e_1's *operands*.

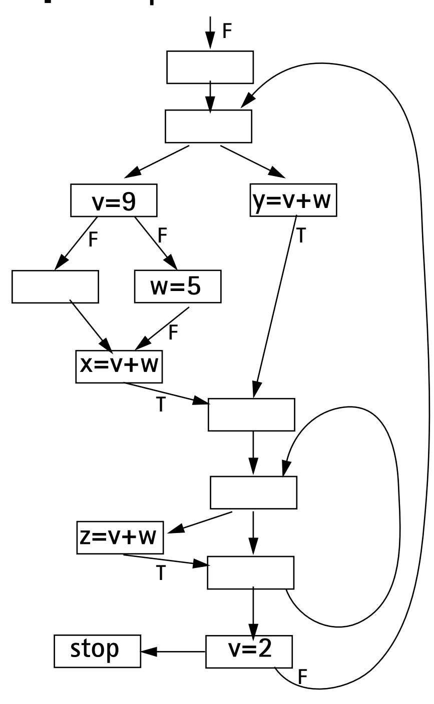
The transfer function for e_1 in block b is defined as:

If e_1 is computed in b after any assignments to e_1's operands in b
Then AvailOut(b) = T
Elsif any of e_1's operands are changed after the last computation of e_1 or e_1's operands are changed without any computation of e_1
Then AvailOut(b) = F
Else AvailOut(b) = AvailIn(b)

The meet operation (to combine solutions) is:

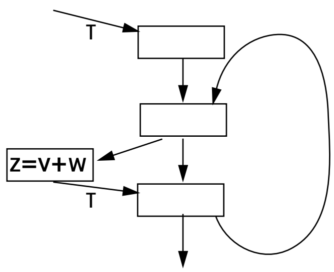
$$\text{AvailIn}(b) = \underset{p \in \text{Pred}(b)}{\text{AND}} \text{AvailOut}(p)$$

Example: e_1=v+w



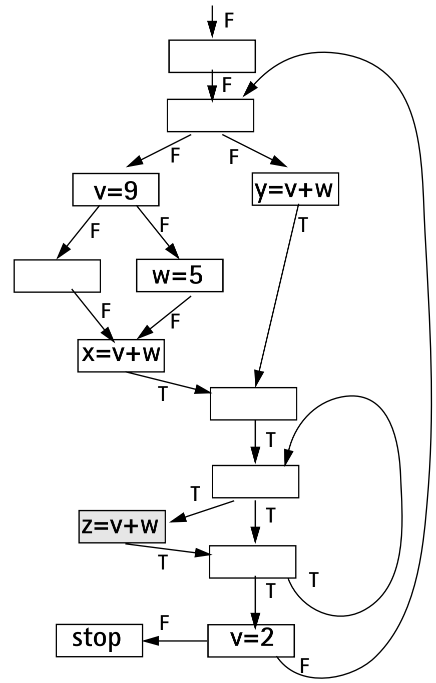
Circularities Require Care

Since data flow values can depend on themselves (because of loops), care is required in assigning initial "guesses" to unknown values. Consider



If the flow value on the loop backedge is initially set to false, it can never become true. (Why?)

Instead we should use True, the *identity* for the AND operation.



Reading Assignment

- Read pages 31–62 of "Automatic Program Optimization," by Ron Cytron. (Linked from the class Web page.)

Very Busy Expressions

This is an interesting variant of available expression analysis.

An expression is *very busy* at a point if it is *guaranteed* that the expression will be computed at some time in the future.

Thus starting at the point in question, the expression must be reached before its value changes.

Very busy expression analysis is a backward flow analysis, since it propagates information about future evaluations backward to "earlier" points in the computation.

The meet lattice is:

T (Expression is Very Busy)

|

F (Expression is Not Very Busy)

As initial values, at the end of all exit nodes, nothing is very busy. Hence, for a given expression,

$\text{VeryBusyOut}(b_{last}) = F$

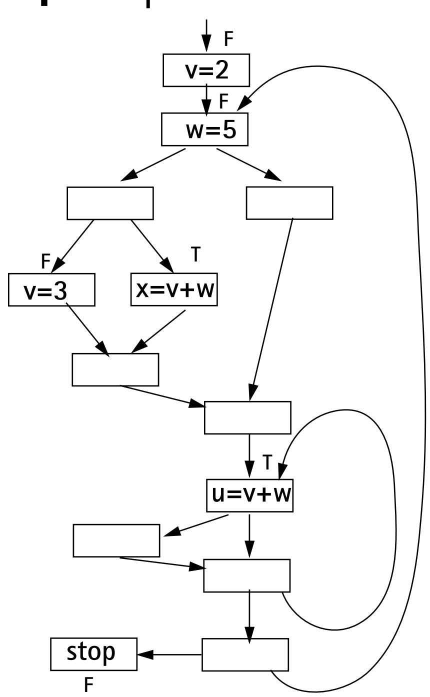
The transfer function for e_1 in block b is defined as:

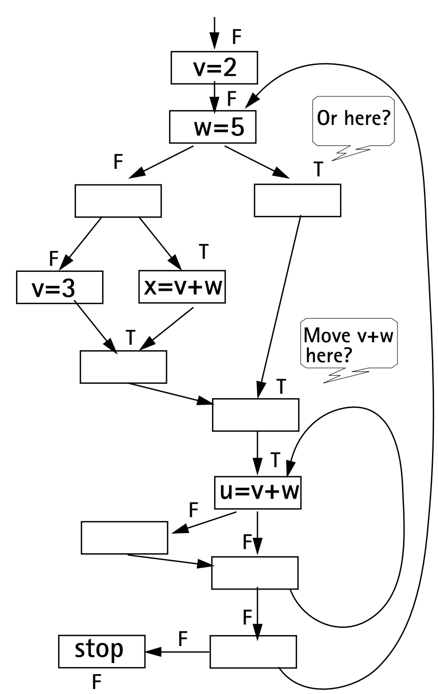
If e_1 is computed in b before any of its operands
Then VeryBusyIn(b) = T
Elsif any of e_1's operands are changed before e_1 is computed
Then VeryBusyIn(b) = F
Else VeryBusyIn(b) = VeryBusyOut(b)

The meet operation (to combine solutions) is:

$$\text{VeryBusyOut}(b) = \underset{s \in \text{Succ}(b)}{\text{AND}} \text{VeryBusyIn}(s)$$

Example: e_1=v+w





Identifying Identical Expressions

We can hash expressions, based on hash values assigned to operands and operators. This makes recognizing potentially redundant expressions straightforward.

For example, if H(a) = 10, H(b) = 21 and H(+) = 5, then (using a simple product hash),
H(a+b) = 10×21×5 Mod TableSize

Effects of Aliasing and Calls

When looking for assignments to operands, we must consider the effects of pointers, formal parameters and calls.

An assignment through a pointer (e.g, `*p = val`) *kills* all expressions dependent on variables `p` might point too. Similarly, an assignment to a formal parameter kills all expressions dependent on variables the formal might be bound to.

A call kills all expressions dependent on a variable changeable during the call.

Lacking careful alias analysis, pointers, formal parameters and calls can kill all (or most) expressions.

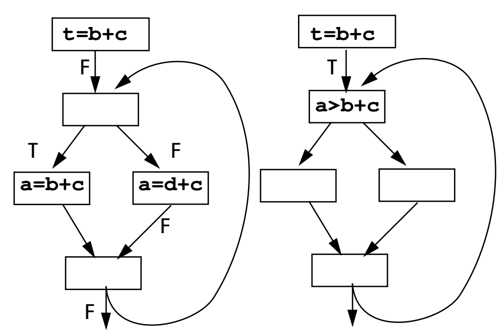
Very Busy Expressions and Loop Invariants

Very busy expressions are ideal candidates for invariant loop motion.

If an expression, invariant in a loop, is also very busy, we know it must be used in the future, and hence evaluation outside the loop must be worthwhile.


```
for (...) {
 if (...)
     a=b+c;
 else a=d+c;}
```

```
for (...) {
 if (a>b+c)
     x=1;
 else x=0;}
```



`b+c` is not very busy at loop entrance

`b+c` is very busy at loop entrance

Reaching Definitions

We have seen reaching definition analysis formulated as a set-valued problem. It can also be formulated on a per-definition basis.

That is, we ask "What blocks does a particular definition to v reach?"

This is a boolean-valued, forward flow data flow problem.

Initially, $DefIn(b_0)$ = false.

For basic block b:

DefOut(b) =
  If the definition being analyzed is the last definition to v in b
  Then True
  Elsif any other definition to v occurs in b
  Then False
  Else DefIn(b)

The meet operation (to combine solutions) is:

$$DefIn(b) = \underset{p \in Pred(b)}{OR} DefOut(p)$$

To get all reaching definition, we do a series of single definition analyses.

Live Variable Analysis

This is a boolean-valued, backward flow data flow problem.

Initially, $LiveOut(b_{last})$ = false.

For basic block b:

LiveIn(b) =
  If the variable is used before it is defined in b
  Then True
  Elsif it is defined before it is used in b
  Then False
  Else LiveOut(b)

The meet operation (to combine solutions) is:

$$LiveOut(b) = \underset{s \in Succ(b)}{OR} LiveIn(s)$$

Bit Vectoring Data Flow Problems

The four data flow problems we have just reviewed all fit within a *single* framework.

Their solution values are Booleans (bits).

The meet operation is And or OR.

The transfer function is of the general form

$Out(b) = (In(b) - Kill_b) \cup Gen_b$

or

$In(b) = (Out(b) - Kill_b) \cup Gen_b$

where $Kill_b$ is true if a value is "killed" within b and Gen_b is true if a value is "generated" within b.

In Boolean terms:

$Out(b) = (In(b) \text{ AND Not } Kill_b) \text{ OR } Gen_b$

or

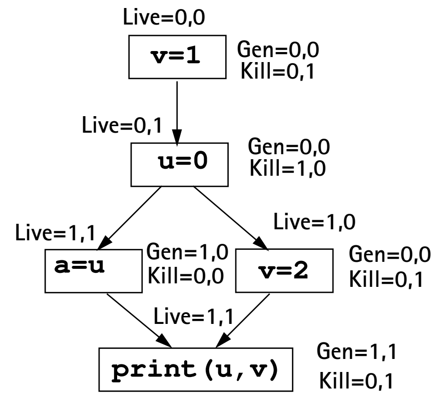
$In(b) = (Out(b) \text{ AND Not } Kill_b) \text{ OR } Gen_b$

An advantage of a bit vectoring data flow problem is that we can do a series of data flow problems "in parallel" using a bit vector.

Hence using ordinary word-level ANDs, ORs, and NOTs, we can solve 32 (or 64) problems simultaneously.

Example

Do live variable analysis for u and v, using a 2 bit vector:



We expect no variable to be live at the start of b_0. (Why?)

Depth-First Spanning Trees

Sometimes we want to "cover" the nodes of a control flow graph with an acyclic structure.

This allows us to visit nodes once, without worrying about cycles or infinite loops.

Also, a careful visitation order can approximate forward control flow (very useful in solving forward data flow problems).

A Depth-First Spanning Tree (DFST) is a tree structure that covers the nodes of a control flow graph, with the start node serving as root of the DFST.

Building a DFST

We will visit CFG nodes in depth-first order, keeping arcs if the visited node hasn't be reached before.

To create a DFST, T, from a CFG, G:

1. T $\leftarrow$ empty tree
2. Mark all nodes in G as "unvisited."
3. Call DF(start node)

DF (node) {

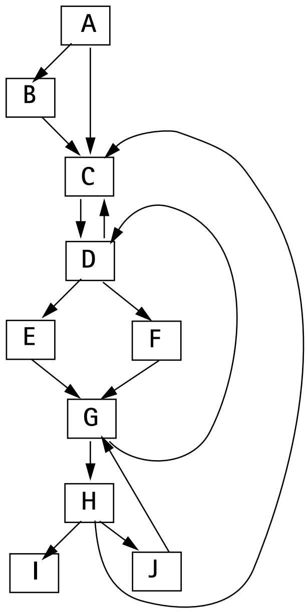
1. Mark node as visited.
2. For each successor, s, of node in G:

   If s is unvisited

   (a) Add node $\rightarrow$ s to T

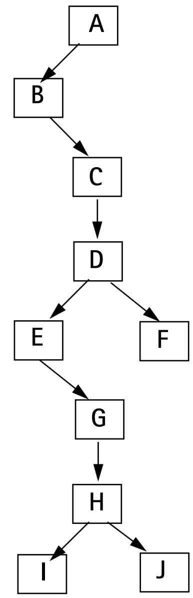
   (b) Call DF(s)

Example



Visit order is A, B, C, D, E, G, H, I, J, F

The DFST is



Categorizing Arcs using a DFST

Arcs in a CFG can be categorized by examining the corresponding DFST.

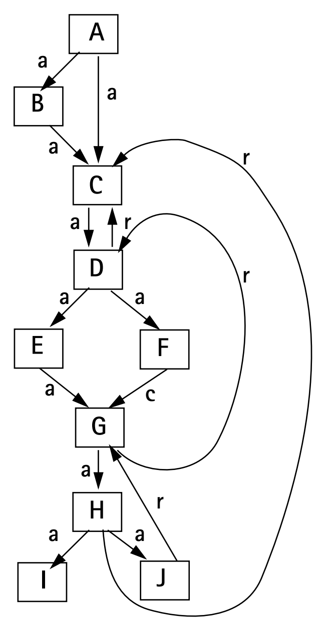
An arc A→B in a CFG is

(a) An *Advancing Edge* if B is a proper descendent of A in the DFST.

(b) A *Retreating Edge* if B is an ancestor of A in the DFST. (This includes the A→A case.)

(c) A *Cross Edge* if B is neither a descendent nor an ancestor of A in the DFST.

Example



Depth-First Order

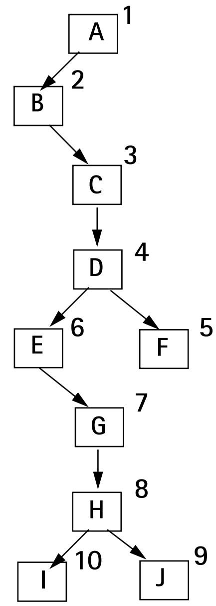
Once we have a DFST, we can label nodes with a *Depth-First Ordering* (DFO).

Let i = the number of nodes in a CFG (= the number of nodes in its DFST).

```
DFO(node) {
   For (each successor s of node) do
         DFO(s);
   Mark node with i;
   i--;
}
```


Example

The number of nodes = 10.



Application of Depth-First Ordering

- *Retreating edges* (a necessary component of loops) are easy to identify:
  a→b is a retreating edge if and only if dfo(b) ≤ dfo(a)
- A depth-first ordering in an excellent *visit order* for solving forward data flow problems. We want to visit nodes in essentially topological order, so that all predecessors of a node are visited (and evaluated) before the node itself is.

Dominators

A CFG node M *dominates* N (M dom N) if and only if *all* paths from the start node to N *must* pass through M.

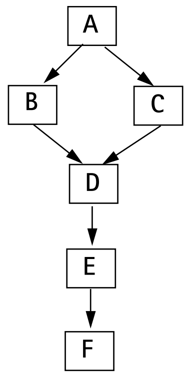
A node trivially dominates itself.

Thus (N dom N) is always true.

A CFG node M *strictly dominates* N (M sdom N) if and only if (M dom N) and M ≠ N.
A node can't strictly dominates itself.

Thus (N sdom N) is never true.

A CFG node may have many dominators.



Node F is dominated by F, E, D and A.

Immediate Dominators

If a CFG node has more than one dominator (which is common), there is always a unique "closest" dominator called its *immediate dominator*.

(M idom N) if and only if
(M sdom N) and
(P sdom N) $\Rightarrow$ (P dom M)

To see that an immediate dominator always exists (except for the start node) and is unique, assume that node N is strictly dominated by M_1, M_2, ..., M_p, $P \geq 2$.

By definition, M_1, ..., M_p must appear on *all* paths to N, including acyclic paths.

Look at the relative ordering among M_1 to M_p on some arbitrary acyclic path from the start node to N.

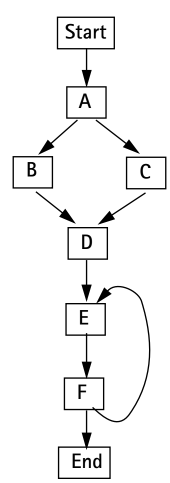
Assume that M_i is "last" on that path (and hence "nearest" to N).

If, on some other acyclic path, $M_j \neq M_i$ is last, then we can shorten this second path by going directly from M_i to N without touching any more of the M_1 to M_p nodes.

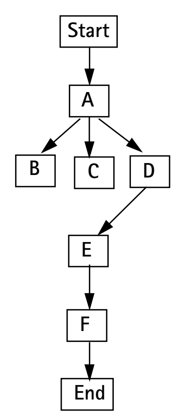
But, this totally removes M_j from the path, contradicting the assumption that (M_j sdom N).

Dominator Trees

Using immediate dominators, we can create a *dominator tree* in which A$\rightarrow$B in the dominator tree if and only if (A idom B).

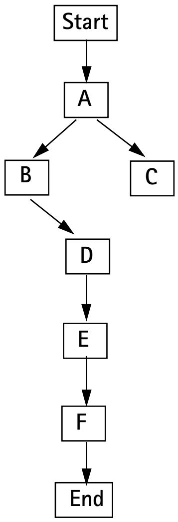


Control Flow Graph

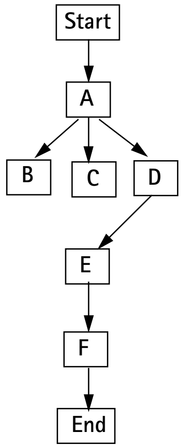


Dominator Tree

Note that the Dominator Tree of a CFG and its DFST are distinct trees (though they have the same nodes).



Depth-First Spanning Tree



Dominator Tree

A Dominator Tree is a compact and convenient representation of both the dom and idom relations.

A node in a Dominator Tree dominates all its descendents in the tree, and immediately dominates all its children.

Computing Dominators

Dominators can be computed as a Set-valued Forward Data Flow Problem.

If a node N dominates all of node M's predecessors, then N appears on all paths to M. Hence (N dom M).

Similarly, if M *doesn't* dominate all of M's predecessors, then there is a path to M that doesn't include M. Hence $\neg$(N dom M).

These observations give us a "data flow equation" for dominator sets:

$$\text{dom}(N) = \{N\} \cup \bigcap_{M \in \text{Pred}(N)} \text{dom}(M)$$

The analysis domain is the lattice of all subsets of nodes. Top is the set of all nodes; bottom is the empty set. The ordering relation is subset.

The meet operation is intersection.

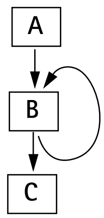
The Initial Condition is that $\text{DomIn}(b_0) = \phi$

$$\text{DomIn}(b) = \bigcap_{c \in \text{Pred}(b)} \text{DomOut}(c)$$

$$\text{DomOut}(b) = \text{DomIn}(b) \cup \{b\}$$

Loops Require Care

Loops in the Control Flow Graph induce circularities in the Data Flow equations for Dominators. In



we have the rule dom(B) =
DomOut(B) =
DomIn(B) U $\{B\}$ =
$\{B\}$ U ($\underline{\text{DomOut(B)}}$ $\cap$ DomOut(A))

If we choose DomOut(B) = ϕ initially, we get DomOut(B) =
$\{B\}$ U ($\phi \cap$ DomOut(A)) = $\{B\}$
which is *wrong*.

Instead, we should use the Universal Set (of all nodes) which is the identity for $\cap$.

Then we get DomOut(B) =
$\{B\}$ U ($\{$all nodes$\}$ $\cap$ DomOut(A)) =
$\{B\}$ U DomOut(A)
which is correct.

A Worklist Algorithm for Dominators

The data flow equations we have developed for dominators can be evaluated using a simple Worklist Algorithm.

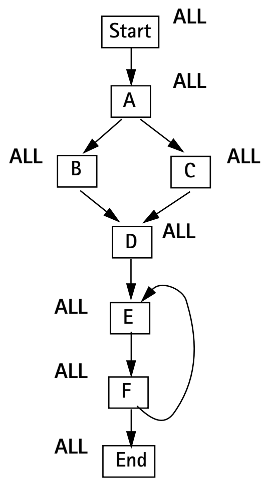
Initially, each node's dominator set is set to the set of all nodes. We add the start node to our worklist.

For each node on the worklist, we reevaluate its dominator set. If the set changes, the updated dominator set is used, and all the node's successors are added to the worklist (so that the updated dominator set can be propagated).

The algorithm terminates when the worklist becomes empty, indicating that a stable solution has been found.

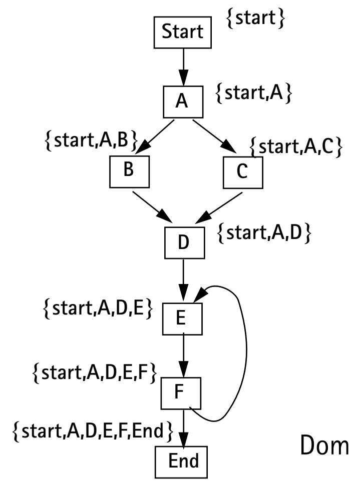
```
Compute Dominators(){
  For (each n ∈ NodeSet)
      Dom(n) = NodeSet
  WorkList = {StartNode}
  While (WorkList ≠ φ) {
    Remove any node Y from WorkList
    New = {Y} U   ∩          Dom(X)
               X ∈ Pred(Y)
    If New ≠ Dom(Y) {
       Dom(Y) = New
       For (each Z ∈ Succ(Y))
          WorkList = WorkList U {Z}
}}}
```

Example

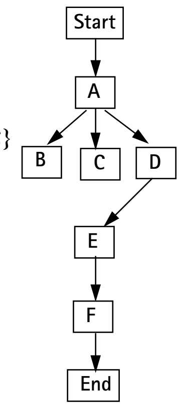


Initially the WorkList = {Start}.

Be careful when Pred(Node) = ϕ.



Control Flow Graph

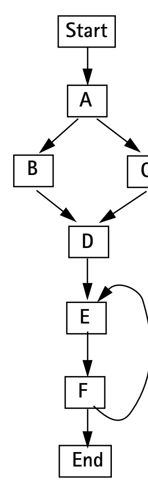


Dominator Tree

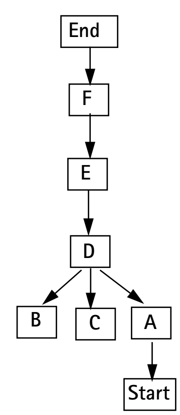
Postdominance

A block Z *postdominates* a block Y (Z pdom Y) if and only if all paths from Y to an exit block must pass through Z. Notions of immediate postdominance and a postdominator tree carry over.

Note that if a CFG has a single exit node, then postdominance is equivalent to dominance if flow is reversed (going from the exit node to the start node).



Control Flow Graph



Postdominator Tree

Dominance Frontiers

Dominators and postdominators tell us which basic block must be executed prior to, of after, a block N.

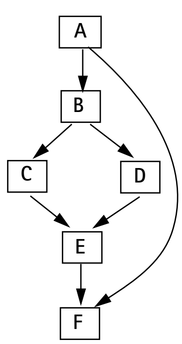
It is interesting to consider blocks "just before" or "just after" blocks we're dominated by, or blocks we dominate.

The Dominance Frontier of a basic block N, DF(N), is the set of all blocks that are immediate successors to blocks dominated by N, but which aren't themselves strictly dominated by N.

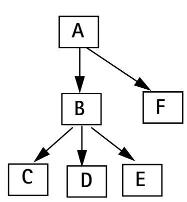
DF(N) =
$\{Z \mid M \to Z \ \&\ (N \text{ dom } M)\ \&$
$\neg(N \text{ sdom } Z)\}$

The dominance frontier of N is the set of blocks that are not dominated N and which are "first reached" on paths from N.

Example



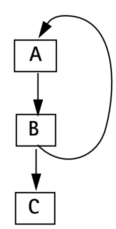
Control Flow Graph



Dominator Tree

Block	A	B	C	D	E	F
Dominance Frontier	ϕ	{F}	{E}	{E}	{F}	ϕ

A block can be in its own Dominance Frontier:



Here, DF(A) = {A}

Why? Reconsider the definition:

DF(N) =
$\{Z \mid M \to Z \ \&\ (N \text{ dom } M)\ \&$
$\neg(N \text{ sdom } Z)\}$

Now B is dominated by A and $B \to A$.

Moreover, A does not *strictly* dominate itself. So, it meets the definition.

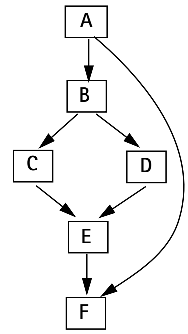
Postdominance Frontiers

The Postdominance Frontier of a basic block N, PDF(N), is the set of all blocks that are immediate predecessors to blocks postdominated by N, but which aren't themselves postdominated by N.

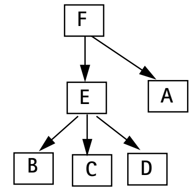
$$PDF(N) = \{Z \mid Z \rightarrow M \ \& \ (N \text{ pdom } M) \ \& \ \neg(N \text{ pdom } Z)\}$$

The postdominance frontier of N is the set of blocks closest to N where a choice was made of whether to reach N or not.

Example



Control Flow Graph



Postominator Tree

Block	A	B	C	D	E	F
Postdominance Frontier	ϕ	{A}	{B}	{B}	{A}	ϕ

Control Dependence

Since CFGs model flow of control, it is useful to identify those basic blocks whose execution is controlled by a branch decision made by a predecessor.

We say Y is *control dependent* on X if, reaching X, choosing one out arc will force Y to be reached, while choosing another arc out of X allows Y to be avoided.

Formally, Y is control dependent on X if and only if,

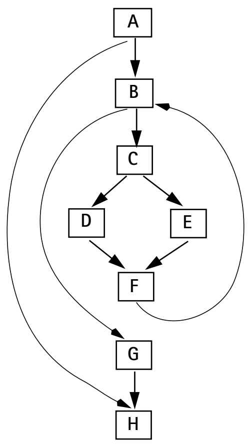
(a) Y postdominates a successor of X.

(b) Y does not postdominate all successors of X.

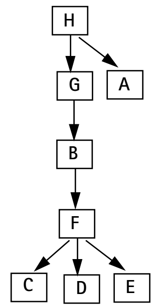
X is the most recent block where a choice was made to reach Y or not.

Control Dependence Graph

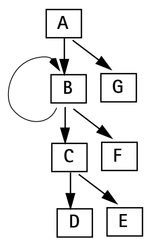
We can build a *Control Dependence Graph* that shows (in graphical form) all Control Dependence relations.
(A Block *can be* Control Dependent on itself.)



Control Flow Graph



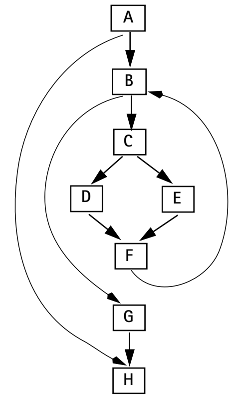
Postominator Tree



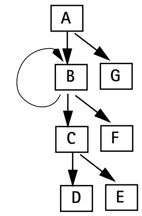
Control Dependence Graph

What happened to H in the CD Graph?

Let's reconsider the CD Graph:



Control Flow Graph



Control Dependence Graph

Blocks C and F, as well as D and E, seem to have the same control dependence relations with their parent. But this isn't so!

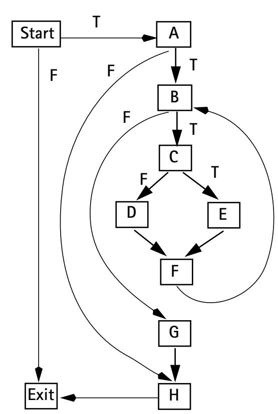
C and F *are* control equivalent, but D and E are *mutually exclusive*!

Improving the Representation of Control Dependence

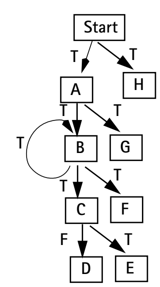
We can label arcs in the CFG and the CD Graph with the condition (T or F or some switch value) that caused the arc to be selected for execution.

This labeling then shows the conditions that lead to the execution of a given block.

To allow the exit block to appear in the CD Graph, we can also add "artificial" start and exit blocks, linked together.



Control Flow Graph



Control Dependence Graph

Now C and F have the same Control Dependence relations—they are part of the same extended basic block.

But D and E aren't identically control dependent. Similarly, A and H are control equivalent, as are B and G.

Data Flow Frameworks Revisited

Recall that a Data Flow problem is characterized as:

(a) A Control Flow Graph

(b) A Lattice of Data Flow values

(c) A Meet operator to join solutions from Predecessors or Successors

(d) A Transfer Function
Out = f_b(In) or In = f_b(Out)

Value Lattice

The lattice of values is usually a *meet semilattice* defined by:

A: a set of values

T and $\perp$ ("top" and "bottom"): distinguished values in the lattice

$\le$: A reflexive partial order relating values in the lattice

$\wedge$: An associative and commutative meet operator on lattice values

Lattice Axioms

The following axioms apply to the lattice defined by A, T, $\perp$, $\le$ and $\wedge$:

$a \le b \iff a \wedge b = a$

$a \wedge a = a$

$(a \wedge b) \le a$

$(a \wedge b) \le b$

$(a \wedge T) = a$

$(a \wedge \perp) = \perp$

Monotone Transfer Function

Transfer Functions, f_b:L $\to$ L (where L is the Data Flow Lattice) are normally required to be monotone.

That is $x \le y \Rightarrow f_b(x) \le f_b(y)$.

This rule states that a "worse" input can't produce a "better" output.

Monotone transfer functions allow us to guarantee that data flow solutions are stable.

If we had $f_b(T) = \perp$ and $f_b(\perp)=T$, then solutions might oscillate between T and $\perp$ indefinitely.

Since $\perp \le T$, $f_b(\perp)$ should be $\le f_b(T)$. But $f_b(\perp) = T$ which is not $\le f_b(T) = \perp$. Thus f_b isn't monotone.

Dominators fit the Data Flow Framework

Given a set of Basic Blocks, N, we have:

A is 2^N (all subsets of Basic Blocks).

T is N.

$\perp$ is ϕ.

$a \le b \equiv a \subseteq b$.

$f_Z(in) = In \cup \{Z\}$

$\wedge$ is $\cap$ (set intersection).

The required axioms are satisfied:

$a \subseteq b \ \Leftrightarrow a \cap b = a$

$a \cap a = a$

$(a \cap b) \subseteq a$

$(a \cap b) \subseteq b$

$(a \cap N) = a$

$(a \cap \phi) = \phi$

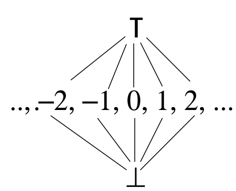
Also f_Z is monotone since

$a \subseteq b \Rightarrow a \cup \{Z\} \subseteq \ b \cup \{Z\} \Rightarrow f_Z(a) \subseteq \ f_Z(b)$

Constant Propagation

We can model Constant Propagation as a Data Flow Problem. For each scalar integer variable, we will determine whether it is known to hold a particular constant value at a particular basic block.

The value lattice is



T represents a variable holding a constant, whose value is not yet known.

i represents a variable holding a known constant value.

$\perp$ represents a variable whose value is non-constant.

This analysis is complicated by the fact that variables interact, so we can't just do a series of independent one variable analyses.

Instead, the solution lattice will contain functions (or vectors) that map each variable in the program to its constant status (T, $\perp$, or some integer).

Let V be the set of all variables in a program.

Let $t : V \rightarrow N \cup \{T,\perp\}$

t is the set of all total mappings from V (the set of variables) to $N \cup \{T,\perp\}$ (the lattice of "constant status" values).

For example, $t_1=(T,6,\perp)$ is a mapping for three variables (call them A, B and C) into their constant status. t_1 says A is considered a constant, with value as yet undetermined. B holds the value 6, and C is non-constant.

We can create a lattice composed of t functions:

$t_T(V) = T$ ($\forall$ V) ($t_T=(T,T,T, \ldots)$

$t_\perp(V) = \perp$ ($\forall$ V) ($t_\perp=(\perp,\perp,\perp, \ldots)$

$t_a \le t_b \Leftrightarrow \forall v\ t_a(v) \le t_b(v)$

Thus $(1,\perp) \le (T,3)$
since $1 \le T$ and $\perp \le 3$.

The meet operator $\wedge$ is applied *componentwise*:

$t_a \wedge t_b = t_c$
where $\forall v\ t_c(v) = t_a(v) \wedge t_b(b)$

Thus $(1,\perp) \wedge (T,3) = (1,\perp)$
since $1 \wedge T = 1$ and $\perp \wedge 3 = \perp$.

The lattice axioms hold:

$t_a \le t_b \Leftrightarrow t_a \wedge t_b = t_a$ (since this axiom holds for each component)

$t_a \wedge t_a = t_a$ (trivially holds)

$(t_a \wedge t_b) \le t_a$ (per variable def of $\wedge$)

$(t_a \wedge t_b) \le t_b$ (per variable def of $\wedge$)

$(t_a \wedge t_T) = t_a$ (true for all components)

$(t_a \wedge t_\perp) = t_\perp$ (true for all components)

The Transfer Function

Constant propagation is a forward flow problem, so Cout = f_b(Cin)

Cin is a function, t(v), that maps variables to $T,\perp$, or an integer value

$f_b(t(v))$ is defined as:

(1) Initially, let $t'(v)=t(v)$ ($\forall$v)

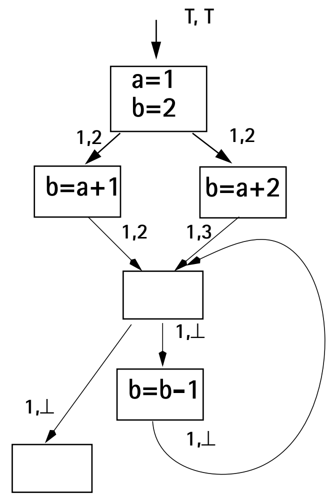
(2) For each assignment statement
$v = e(w_1,w_2,\ldots,w_n)$
in b, in order of execution, do:
If any $t'(w_i) = \perp$ ($1 \le i \le n$)
Then set $t'(v) = \perp$ (strictness)
Elsif any $t'(w_i) = T$ ($1 \le i \le n$)
Then set $t'(v) = T$ (delay eval of v)
Else $t'(v) = e(t'(w_1),t'(w_2),\ldots)$

(3) Cout = t'(v)

Note that in valid programs, we don't use uninitialized variables, so variables mapped to T should only occur prior to initialization.

Initially, all variables are mapped to T, indicating that initially their constant status is unknown.

Example



Distributive Functions

From the properties of $\wedge$ and f's monotone property, we can show that

$f(a \wedge b) \le f(a) \wedge f(b)$

To see this note that

$a \wedge b \le a,\ a \wedge b \le b \Rightarrow$

$f(a \wedge b) \le f(a),\ f(a \wedge b) \le f(b)$ (*)

Now we can establish that

$x \le y,\ x \le z \Rightarrow x \le y \wedge z$ (**)

To see that (**) holds, note that

$x \le y \Rightarrow x \wedge y = x$

$x \le z \Rightarrow x \wedge z = x$

$(y \wedge z) \wedge x \le y \wedge z$

$(y \wedge z) \wedge x = (y \wedge z) \wedge (x \wedge x) =$

$(y \wedge x) \wedge (z \wedge x) = x \wedge x = x$

Thus $x \le y \wedge z$, establishing (**).

Now substituting $f(a \wedge b)$ for x, f(a) for y and f(b) for z in (**) and using (*) we get

$f(a \wedge b) \le f(a) \wedge f(b)$.

Many Data Flow problems have flow equations that satisfy the *distributive property*:

$f(a \wedge b) = f(a) \wedge f(b)$

For example, in our formulation of dominators:

$Out = f_b(In) = In \cup \{b\}$

where

$$In = \bigcap_{p \in Pred(b)} Out(p)$$

In this case, $\wedge = \cap$.

Now $f_b(S_1 \cap S_2) = (S_1 \cap S_2) \cup \{b\}$

Also, $f_b(S_1) \cap f_b(S_2) =$

$(S_1 \cup \{b\}) \cap (S_2 \cup \{b\}) =$

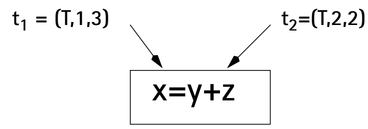
$(S_1 \cap S_2) \cup \{b\}$

So dominators are distributive.

Not all Data Flow Problems are Distributive

Constant propagation is *not* distributive.

Consider the following (with variables (x,y,z)):



Now f(t)=t' where

t'(y) = t(y), t'(z) = t(z),

t'(x) = if t(y)=$\perp$ or t(z) = $\perp$
then $\perp$
elseif t(y)=T or t(z) =T
then T
else t(y)+t(z)

Now $f(t_1 \wedge t_2) = f(T,\perp,\perp) = (\perp,\perp,\perp)$

$f(t_1) = (4,1,3)$

$f(t_2) = (4,2,2)$

$f(t_1) \wedge f(t_2) = (4,\perp,\perp) \geq (\perp,\perp,\perp)$

Why does it Matter if a Data Flow Problem isn't Distributive?

Consider actual program execution paths from b_0 to (say) b_k.

One path might be $b_0, b_{i_1}, b_{i_2}, \ldots, b_{i_n}$ where $b_{i_n} = b_k$.

At b_k the Data Flow information we want is

$f_{i_n}(\ldots f_{i_2}(f_{i_1}(f_0(T)))\ldots) \equiv f(b_0, b_1, \ldots, b_{i_n})$

On a different path to b_k, say $b_0, b_{j_1}, b_{j_2}, \ldots, b_{j_m}$, where $b_{j_m} = b_k$

the Data Flow result we get is
$f_{j_m}(\ldots f_{j_2}(f_{j_1}(f_0(T)))\ldots) \equiv$
$f(b_0, b_{j_1}, \ldots, b_{j_m})$.

Since we can't know at compile time which path will be taken, we must *combine* all possible paths:

$$\bigwedge_{p \in \text{ all paths to } b_k} f(p)$$

This is the *meet over all paths* (MOP) solution. It is the *best possible* static solution. (Why?)

As we shall see, the meet over all paths solution can be computed efficiently, using standard Data Flow techniques, if the problem is Distributive.

Other, non-distributive problems (like Constant Propagation) can't be solved as precisely.

Explicitly computing and meeting all paths is prohibitively expensive.

Conditional Constant Propagation

We can extend our Constant Propagation Analysis to determine that some paths in a CFG aren't executable. This is *Conditional Constant Propagation*.

Consider

```
i = 1;
if (i > 0)
      j = 1;
else j = 2;
```

Conditional Constant Propagation can determine that the else part of the if is unreachable, and hence `j` must be 1.

The idea behind Conditional Constant Propagation is simple. Initially, we mark all edges out of conditionals as "not reachable."

Starting at b_0, we propagate constant information *only* along edges considered reachable.

When a boolean expression $b(v_1,v_2,...)$ controls a conditional branch, we evaluate $b(v_1,v_2,...)$ using the $t(v)$ mapping that identifies the "constant status" of variables.

If $t(v_i)$=T for any v_i, we consider all out edges unreachable (for now).

Otherwise, we evaluate $b(v_1,v_2,...)$ using $t(v)$, getting true, false or $\bot$.

Note that the short-circuit properties of boolean operators may yield true or false even if $t(v_i)=\bot$ for some v_i.

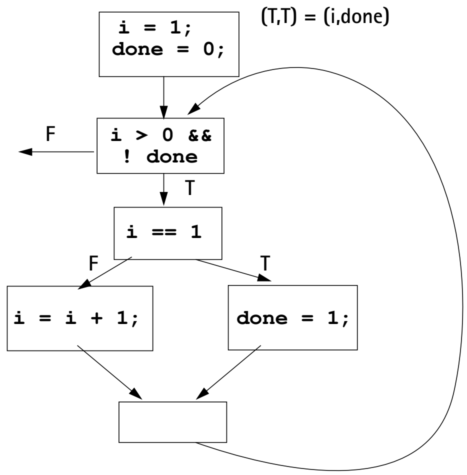
If $b(v_1,v_2,...)$ is true or false, we mark only one out edge as reachable.

Otherwise, if $b(v_1,v_2,...)$ evaluates to $\bot$, we mark all out edges as reachable.

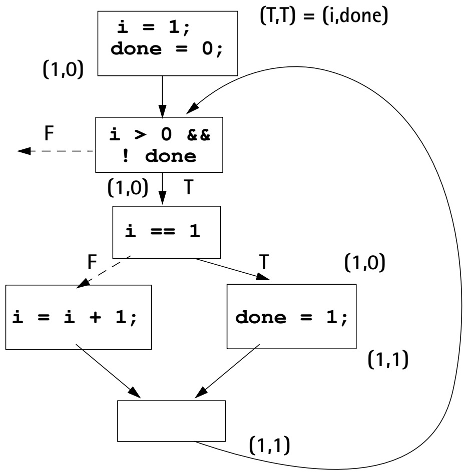
We propagate constant information only along reachable edges.

Example

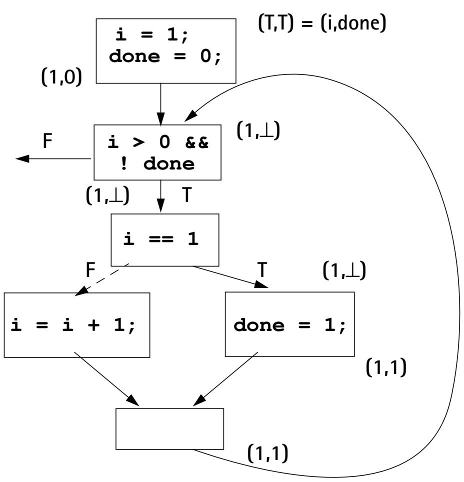
```
i = 1;
done = 0;
while ( i > 0 && ! done) {
  if (i == 1)
       done = 1;
  else i = i + 1;
}
```



Pass 1:



Pass 2:



Reading Assignment

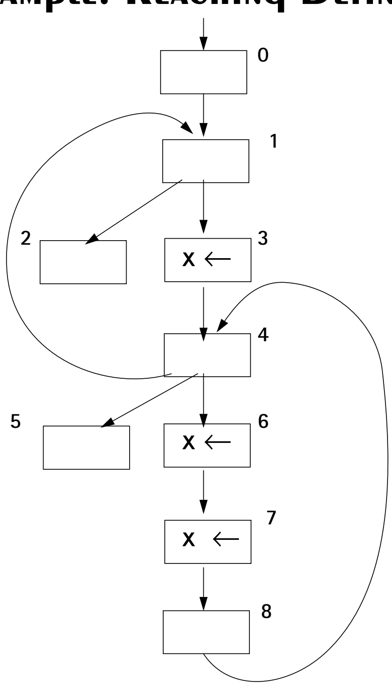
- Read pages 63-end of "Automatic Program Optimization," by Ron Cytron. (Linked from the class Web page.)

Iterative Solution of Data Flow Problems

This algorithm will use DFO numbering to determine the order in which blocks are visited for evaluation. We iterate over the nodes until convergence.


```
EvalDF{
  For (all n ∈ CFG) {
     soln(n) = T
     ReEval(n) = true }
  Repeat
     LoopAgain = false
     For (all n ∈ CFG in DFO order){
        If (ReEval(n)) {
             ReEval(n) = false
             OldSoln = soln(n)
             In =   ∧        soln(p)
                 p ∈ Pred(n)
             soln(n) = f_n(In)
             If (soln(n) ≠ OldSoln) {
                For (all s ∈ Succ(n)) {
                   ReEval(s) = true
                   LoopAgain = LoopAgain OR
                        IsBackEdge(n,s)
     } }    } }
  Until (! LoopAgain)
}
```


Example: Reaching Definitions



We'll do this as a set-valued problem (though it really is just three bit-valued analyses, since each analysis is independent).

L is the power set of Basic Blocks

$\wedge$ is set union

T is ϕ; $\perp$ is the set of all blocks

$a \le b \equiv b \subseteq a$

$f_3(in) = \{3\}$

$f_6(in) = \{6\}$

$f_7(in) = \{7\}$

For all other blocks, $f_b(in) = in$

We'll track soln and ReEval across multiple passes

	0	1	2	3	4	5	6	7	8	Loop-Again
Initial	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	**true**
	true	**true**	**true**	**true**	**true**	**true**	**true**	**true**	**true**	
Pass 1	ϕ	ϕ	ϕ	{3}	{3}	{3}	{6}	{7}	{7}	**true**
	false	**true**	**false**	**false**	**true**	**false**	**false**	**false**	**false**	
Pass 2	ϕ	{3}	{3}	{3}	{3,7}	{3,7}	{6}	{7}	{7}	**true**
	false	**true**	**false**	**false**	**false**	**false**	**false**	**false**	**false**	
Pass 3	ϕ	{3,7}	{3,7}	{3}	{3,7}	{3,7}	{6}	{7}	{7}	**false**
	false	**false**	**false**	**false**	**false**	**false**	**false**	**false**	**false**	

Properties of Iterative Data Flow Analysis

- If the height of the lattice (the maximum distance from T to $\perp$) is finite, then termination is *guaranteed*.
  Why?
  Recall that transfer functions are assumed monotone ($a \le b \Rightarrow f(a) \le f(b)$).
  Also, $\wedge$ has the property that
  $a \wedge b \le a$ and $a \wedge b \le b$.
  At each iteration, some solution value must change, else we halt. If something changes it must "move down" the lattice (we start at T). If the lattice has finite height, each block's value can change only a bounded number of times. Hence termination is guaranteed.

- If the iterative data flow algorithm terminates, a valid solution *must* have been computed. (This is because data flow values flow forward, and any change along a backedge forces another iteration.)

How Many Iterations are Needed?

Can we bound the number of iterations needed to compute a data flow solution?

In our example, 3 passes were needed, but why?

In an "ideal" CFG, with no loops or backedges, only 1 pass is needed.

With backedges, it can take several passes for a value computed in one block to reach a block that depends upon the value.

Let p be the maximum number of backedges in any acyclic path in the CFG.

Then (p+1) passes suffice to propagate a data flow value to any other block that uses it.

Recall that any block's value can change only a bounded number of times. In fact, the height of the lattice (maximum distance from top to bottom) is that bound.

Thus the maximum number of passes in our iterative data flow evaluator =

(p+1) * Height of Lattice

In our example, p = 2 and lattice height really was 1 (we did 3 independent bit valued problems).

So passes needed = (2+1)*1 = 3.

Rapid Data Flow Frameworks

We still have the concern that it may take many passes to traverse a Solution Lattice that has a significant height.

Many data flow problems are *rapid*. For rapid data flow problems, extra passes to feed back values along cyclic paths aren't needed.

For a data flow problem to be rapid we require that:

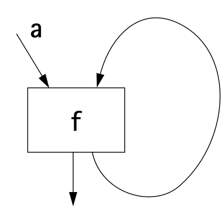
$(\forall a \in A)(\forall f \in F) \quad a \wedge f(T) \leq f(a)$

This is an odd requirement that states that using f(T) as a very crude approximation to a value computed by F is OK when joined using the $\wedge$ operator. In effect the term "a" rather than f(T) is dominant).

(Recall that $a \wedge f(a) \leq f(a)$ always holds.)

How does the Rapid Data Flow Property Help?

Consider a direct feedback loop (the idea holds for indirect loops too):



a is an input from outside the loop.

Our concern is how often we'll need to reevaluate f, as new values are computed and fed back into f.

Initially, we'll use T to model the value on the backedge.

Iteration 1: Input = $a \wedge T = a$

Output = $f(a)$

Iteration 2: Input = $a \wedge f(a)$

Output = $f(a \wedge f(a))$

Iteration 3: Input = $a \wedge f(a \wedge f(a))$

Now we'll exploit the rapid data flow property: $b \wedge f(T) \le f(b)$

Let $b \equiv a \wedge f(a)$

Then $a \wedge f(a) \wedge f(T) \le f(a \wedge f(a))$ (*)

Note that $x \le y \Rightarrow a \wedge x \le a \wedge y$ (**)

To prove this, recall that

(1) $p \wedge q = p \Rightarrow p \le q$

(2) $x \le y \Rightarrow x \wedge y = x$

Thus $(a \wedge x) \wedge (a \wedge y) = a \wedge (x \wedge y) = (a \wedge x)$ (by 2) $\Rightarrow (a \wedge x) \le (a \wedge y)$ (by 1).

From (*) and (**) we get

$a \wedge a \wedge f(a) \wedge f(T) \le f(a \wedge f(a)) \wedge a$ (***)

Now $a \le T \Rightarrow f(a) \le f(T) \Rightarrow$

$f(a) \wedge f(T) = f(a)$.

Using this on (***) we get

$a \wedge f(a) \le f(a \wedge f(a)) \wedge a$

That is, $Input_2 \le Input_3$

Note too that

$a \wedge f(a) \le a \Rightarrow f(a \wedge f(a)) \le f(a) \Rightarrow$
$a \wedge f(a \wedge f(a)) \le a \wedge f(a)$

That is, $Input_3 \le Input_2$

Thus we conclude $Input_2 = Input_3$, which means we can stop after two passes *independent* of lattice height!

(One initial visit plus one reevaluation via the backedge.)

Many Important Data Flow Problems are Rapid

Consider reaching definitions, done as sets. We may have many definitions to the same variable, so the height of the lattice may be large.

L is the power set of Basic Blocks

$\wedge$ is set union

T is ϕ; $\perp$ is the set of all blocks

$a \le b \equiv a \supseteq b$

$f_b(in) = (In - Kill_b)\ U\ Gen_b$

where Gen_b is the last definition to a variable in b,

$Kill_b$ is all defs to a variable except the last one in b,

$Kill_b$ is empty if there is no def to a variable in b.

The Rapid Data Flow Property is

$a \wedge f(T) \le f(a)$

In terms of Reaching Definitions this is

$a\ U\ f(\phi) \supseteq f(a) \equiv$

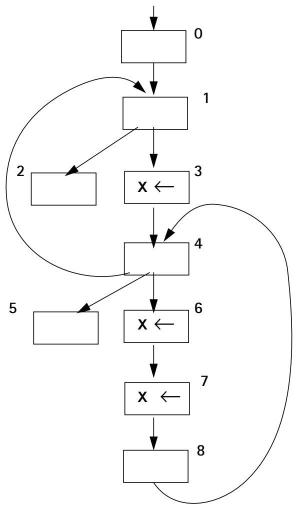
$a\ U\ (\phi - Kill)\ U\ Gen \supseteq (a - Kill)\ U\ Gen$

Simplifying,

$a\ U\ Gen \supseteq (a - Kill)\ U\ Gen$

which always holds.

Recall



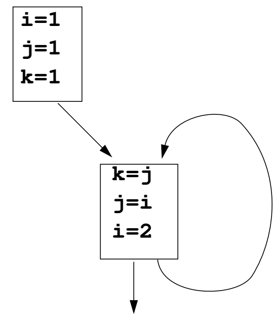
Here it took two passes to transmit the def in b7 to b1, so we expect 3 passes to evaluate *independent* of the lattice height.

Constant Propagation isn't Rapid

We require that

$a \wedge f(T) \leq f(a)$

Consider



Look at the transfer function for the second (bottom) block.

f(t) = t' where

t'(v) = case(v){

k: t(j);

j: t(i);

i: 2; }

Let a = $(\perp,1,1)$.

f(T) = (2,T,T)

$a \wedge f(T) = (\perp,1,1) \wedge (2,T,T) = (\perp,1,1)$

$f(a) = f(\perp,1,1) = (2,\perp,1)$.

Now $(\perp,1,1)$ is not $\leq (2,\perp,1)$

so this problem isn't rapid.

Let's follow the iterations:

Pass 1: In = $(1,1,1) \wedge (T,T,T) = (1,1,1)$

Out = (2,1,1)

Pass 2: In = $(1,1,1) \wedge (2,1,1) = (\perp,1,1)$

Out = $(2,\perp,1)$

Pass 3: In = $(1,1,1) \wedge (2,\perp,1) = (\perp,\perp,1)$

Out = $(2,\perp,\perp)$

This took 3 passes. In general, if we had N variables, we could require N passes, with each pass resolving the constant status of one variable.

How Good Is Iterative Data Flow Analysis?

A single execution of a program will follow some path
$b_0, b_{i_1}, b_{i_2}, \ldots, b_{i_n}$.

The Data Flow solution along this path is

$f_{i_n}(\ldots f_{i_2}(f_{i_1}(f_0(T)))\ldots) \equiv f(b_0, b_1, \ldots, b_{i_n})$

The best possible static data flow solution at some block b is computed over all possible paths from b_0 to b.

Let P_b = The set of all paths from b_0 to b.

$$MOP(b) = \bigwedge_{p \in P_b} f(p)$$

Any particular path p_i from b_0 to b is included in P_b.

Thus $MOP(b) \wedge f(p_i) = MOP(b) \le f(p_i)$.

This means MOP(b) is *always* a safe approximation to the "true" solution $f(p_i)$.

If we have the distributive property for transfer functions,

$f(a \wedge b) = f(a) \wedge f(b)$

then our iterative algorithm *always* computes the MOP solution, the best static solution possible.

To prove this, note that for trivial path of length 1, containing only the start block, b_0, the algorithm computes $f_0(T)$ which is $MOP(b_0)$ (trivially).

Now assume that the iterative algorithm for paths of length n or less to block c *does* compute MOP(c).

We'll show that for paths to block b of length n+1, MOP(b) is computed.

Let P be the set of all paths to b of length n+1 or less.

The paths in P end with b.

$MOP(b) = f_b(f(P_1)) \wedge f_b(f(P_2) \wedge \ldots$

where P_1, P_2, ... are the prefixes (of length n or less) of paths in P with b removed.

Using the distributive property,

$f_b(f(P_1)) \wedge f_b(f(P_2) \wedge \ldots =$

$f_b(f(P_1) \wedge f(P_2) \wedge \ldots)$.

But note that $f(P_1) \wedge f(P_2) \wedge \ldots$ is just the input to f_b in our iterative algorithm, which then applies f_b.

Thus MOP(b) for paths of length n+1 is computed.

For data flow problems that aren't distributive (like constant propagation), the iterative solution is $\leq$ the MOP solution.

This means that the solution is a safe approximation, but perhaps not as "sharp" as we might wish.

Reading Assignment

Read "An Efficient Method of Computing Static Single Assignment Form."
(Linked from the class Web page.)

Exploiting Structure in Data Flow Analysis

So far we haven't utilized the fact that CFGs are constructed from standard programming language constructs like IFs, Fors, and Whiles.

Instead of iterating across a given CFG, we can isolate, and solve symbolically, subgraphs that correspond to "standard" programming language constructs.

We can then progressively simplify the CFG until we reach a single node, or until we reach a CFG structure that matches no standard pattern.

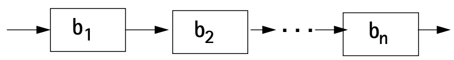
In the latter case, we can solve the residual graph using our iterative evaluator.

Three Program-Building Operations

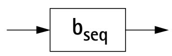
1. Sequential Execution (";")
2. Conditional Execution (If, Switch)
3. Iterative Execution
   (While, For, Repeat)

Sequential Execution

We can reduce a sequential "chain" of basic blocks:



into a single composite block:



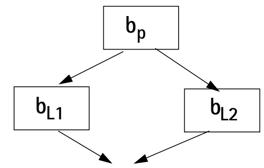
The transfer function of b_{seq} is

$f_{seq} = f_n \circ f_{n-1} \circ \ldots f_1$

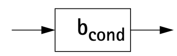
where $\circ$ is functional composition.

Conditional Execution

Given the basic blocks:



we create a single composite block:



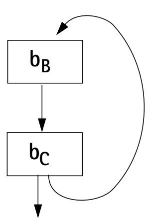
The transfer function of b_{cond} is

$f_{cond} = f_{L1} \circ f_p \wedge f_{L2} \circ f_p$

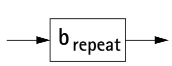
Iterative Execution

Repeat Loop

Given the basic blocks:



we create a single composite block:



Here b_B is the loop body, and b_C is the loop control.

If the loop iterates once, the transfer function is $f_C \circ f_B$.

If the loop iterates twice, the transfer function is $(f_C \circ f_B) \circ (f_C \circ f_B)$.

Considering all paths, the transfer function is $(f_C \circ f_B) \wedge (f_C \circ f_B)^2 \wedge \ldots$

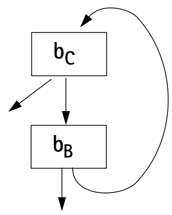
Define fix $f \equiv f \wedge f^2 \wedge f^3 \wedge \ldots$

The transfer function of repeat is then

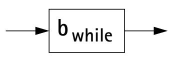
$f_{repeat} = \text{fix}(f_C \circ f_B)$

While Loop.

Given the basic blocks:



we create a single composite block:



Here again b_B is the loop body, and b_C is the loop control.

The loop always executes b_C at least once, and always executes b_C as the last block before exiting.

The transfer function of a while is therefore

$$f_{while} = f_C \wedge fix(f_C \circ f_B) \circ f_C$$

Evaluating Fixed Points

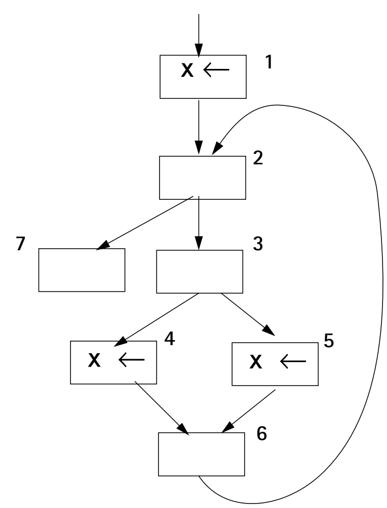
For lattices of height H, and monotone transfer functions, fix f needs to look at no more than H terms.

In practice, we can give fix f an operational definition, suitable for implementation:

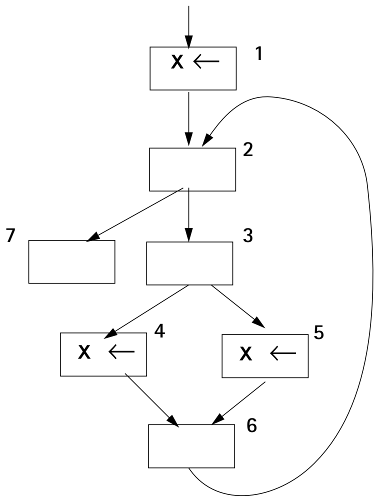
Evaluate

```
(fix f)(x) {
  prev = soln = f(x);
  while (prev ≠ new = f(prev)){
    prev = new;
    soln = soln ∧ new;
  }
  return soln;
}
```


Example—Reaching Definitions



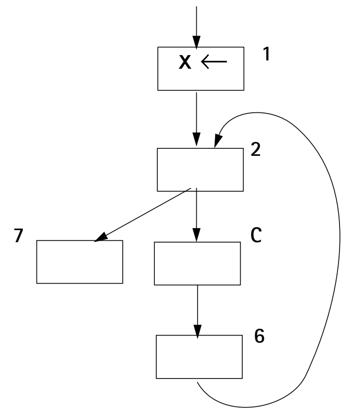
The transfer functions are either constant-valued (f_1={b1}, f_4={b4}, f_5={b5}) or identity functions (f_2=f_3=f_6=f_7=Id).



First we isolate and reduce the conditional:

$f_C = f_4 \circ f_3 \wedge f_5 \circ f_3 =$
$\{b4\} \circ Id \cup \{b5\} \circ Id = \{b4,b5\}$

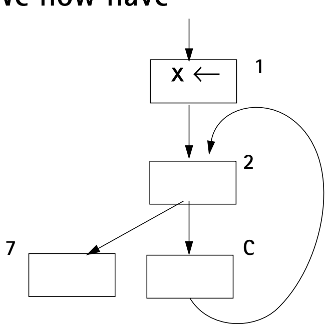
Substituting, we get



We can combine b_C and b_6, to get a block equivalent to b_C. That is,

$f_6 \circ f_C = Id \circ f_C = f_C$

We now have



We isolate and reduce the while loop formed by b_2 and b_C, creating b_W. The transfer function is

$f_W = f_2 \wedge (fix(f_2 \circ f_C) \circ f_2 =$

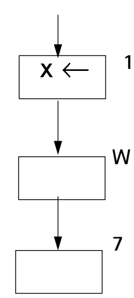
$Id \cup (fix(Id \circ f_C\ ) \circ Id =$

$Id \cup (fix(f_C)) =$

$Id \cup (f_C \wedge f_C^2 \wedge f_C^3 \wedge ...) =$

$Id \cup \{b4,b5\}$

We now have



We compose these three sequential blocks to get the whole solution, f_P.

$f_P = Id \circ (Id \cup \{f4,f5\}) \circ \{b1\} =$

$\{b1,b4,b5\}$.

These are the definitions that reach the end of the program.

We can expand subgraphs to get the solutions at interior blocks.

Thus at the beginning of the while, the solution is {b1}.

At the head if the If, the solution is

$(Id \cup (Id \circ f_C \circ Id) \cup$
$(Id \circ f_C \circ Id \circ f_C \circ Id) \cup \ldots)(\{b1\}) =$
$\{b1\} \cup \{b4,b5\} \cup \{b4,b5\} \cup \ldots =$
$\{b1,b4,b5\}$

At the head of the then part of the If, the solution is Id({b1,b4,b5}) = {b1,b4,b5}.

Static Single Assignment Form

Many of the complexities of optimization and code generation arise from the fact that a given variable may be assigned to in *many* different places.

Thus reaching definition analysis gives us the *set* of assignments that *may* reach a given use of a variable.

Live range analysis must track *all* assignments that may reach a use of a variable and merge them into the same live range.

Available expression analysis must look at *all* places a variable may be assigned to and decide if any kill an already computed expression.

What If

each variable is assigned to in only one place?
(Much like a named constant).

Then for a given use, we can find a single *unique* definition point.

But this seems *impossible* for most programs—or is it?

In *Static Single Assignment* (SSA) Form each assignment to a variable, v, is changed into a unique assignment to new variable, v_i.

If variable v has n assignments to it throughout the program, then (at least) n new variables, v_1 to v_n, are created to replace v. All uses of v are replaced by a use of some v_i.

Phi Functions

Control flow can't be predicted in advance, so we can't always know which definition of a variable reached a particular use.

To handle this uncertainty, we create *phi functions*.

As illustrated below, if v_i and v_j both reach the top of the same block, we add the assignment

$v_k \leftarrow \phi(v_i,v_j)$

to the top of the block.

Within the block, all uses of v become uses of v_k (until the next assignment to v).

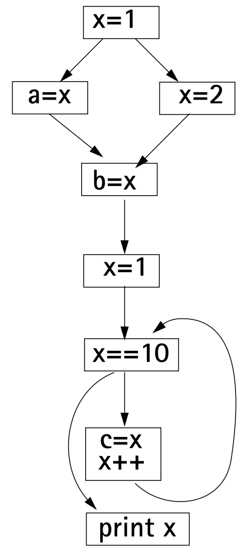
What does $\phi(v_i,v_j)$ Mean?

One way to read $\phi(v_i,v_j)$ is that if control reaches the phi function via the path on which v_i is defined, ϕ "selects" v_i; otherwise it "selects" v_j.

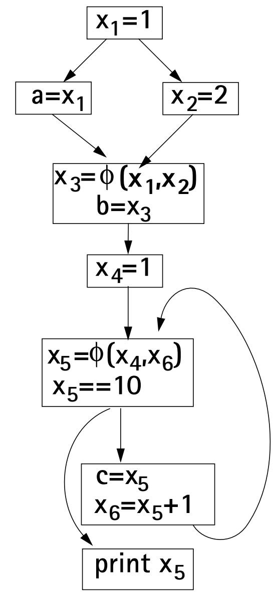
Phi functions may take more than 2 arguments if more than 2 definitions might reach the same block.

Through phi functions we have simple links to all the places where v receives a value, directly or indirectly.

Example



Original CFG



CFG in SSA Form

In SSA form computing live ranges is almost trivial. For each x_i include all x_j variables involved in phi functions that define x_i.

Initially, assume x_1 to x_6 (in our example) are independent. We then union into equivalence classes x_i values involved in the same phi function or assignment.

Thus x_1 to x_3 are unioned together (forming a live range). Similarly, x_4 to x_6 are unioned to form a live range.

Constant Propagation in SSA

In SSA form, constant propagation is simplified since values flow directly from assignments to uses, and phi functions represent natural "meet points" where values are combined (into a constant or $\perp$).

Even conditional constant propagation fits in. As long as a path is considered unreachable, it variables are set to T (and therefore ignored at phi functions, which meet values together).

Example

```
i=6
j=1
k=1
repeat
  if (i==6)
     k=0
  else
     i=i+1
  i=i+k
  j=j+1
until (i==j)
```

```
i_1=6
j_1=1
k_1=1
repeat
  i_2=φ(i_1,i_5)
  j_2=φ(j_1,j_3)
  k_2=φ(k_1,k_4)
  if (i_2==6)
     k_3=0
  else
     i_3=i_2+1
  i_4=φ(i_2,i_3)
  k_4=φ(k_3,k_2)
  i_5=i_4+k_4
  j_3=j_2+1
until (i_5==j_3)
```

	i_1	j_1	k_1	i_2	j_2	k_2	k_3	i_3	i_4	k_4	i_5	j_3
Pass1	6	1	1	$6\wedge T$	$1\wedge T$	$1\wedge T$	0	$\boxed{T}$	$6\wedge T$	$\boxed{0}$	$\boxed{6}$	2
Pass2	6	1	1	$6\wedge 6$	$\perp$	$\perp$	0	T	6	0	6	$\perp$

We have determined that i=6 everywhere.

Putting Programs into SSA Form

Assume we have the CFG for a program, which we want to put into SSA form. We must:

- Rename all definitions and uses of variables
- Decide where to add phi functions

Renaming variable definitions is trivial—each assignment is to a new, unique variable.

After phi functions are added (at the heads of selected basic blocks), only one variable definition (the most recent in the block) can reach any use. Thus renaming uses of variables is easy.

Placing Phi Functions

Let b be a block with a definition to some variable, v. If b contains more than one definition to v, the last (or most recent) applies.

What is the first basic block following b where some other definition to v *as well as* b's definition can reach?

In blocks dominated by b, b's definition *must* have been executed, though other later definitions may have overwritten b's definition.

Domination Frontiers (Again)

Recall that the Domination Frontier of a block b, is defined as

$$DF(N) = \{Z \mid M \rightarrow Z \ \& \ (N \text{ dom } M) \ \& \ \neg(N \text{ sdom } Z)\}$$

The Dominance Frontier of a basic block N, DF(N), is the set of all blocks that are immediate successors to blocks dominated by N, but which aren't themselves strictly dominated by N.

Assume that an initial assignment to all variables occurs in b_0 (possibly of some special "uninitialized value.")

We will need to place a phi function at the start of all blocks in b's Domination Frontier.

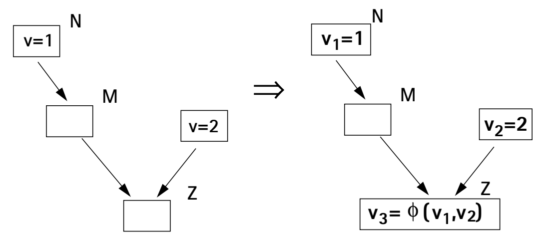
The phi functions will join the definition to v that occurred in b (or in a block dominated by b) with definitions occurring on paths that don't include b.

After phi functions are added to blocks in DF(b), the domination frontier of blocks with newly added phi's will need to be computed (since phi functions imply assignment to a new v_i variable).

Examples of How Domination Frontiers Guide Phi Placement

DF(N) =
$\{Z \mid M \to Z$ & (N dom M) & $\neg$(N sdom Z)$\}$

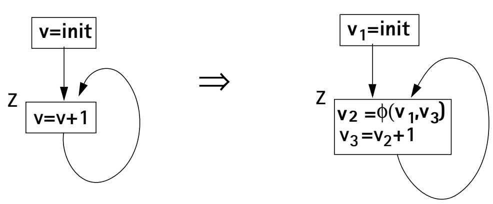
Simple Case:



Here, (N dom M) but $\neg$(N sdom Z),

so a phi function is needed in Z.

Loop:



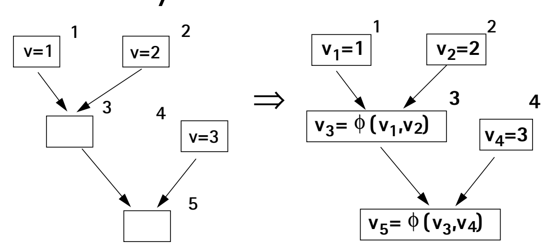
Here, let M = Z = N. M$\to$Z,

(N dom M) but $\neg$(N sdom Z),

so a phi function *is* needed in Z.

DF(N) =
$\{Z \mid M \to Z$ & (N dom M) & $\neg$(N sdom Z)$\}$

Sometimes Phi's must be Placed Iteratively



Now, DF(b1) = $\{$b3$\}$, so we add a phi function in b3. This adds an assignment into b3. We then look at DF(b3) = $\{$b5$\}$, so another phi function must be added to b5.

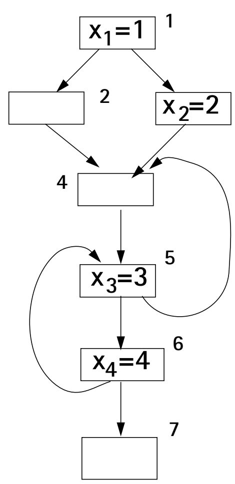
Phi Placement Algorithm

To decide what blocks require a phi function to join a definition to a variable v in block b:

1. Compute $D_1 = DF(b)$.
   Place Phi functions at the head of all members of D_1.
2. Compute $D_2 = DF(D_1)$.
   Place Phi functions at the head of all members of $D_2 - D_1$.
3. Compute $D_3 = DF(D_2)$.
   Place Phi functions at the head of all members of $D_3 - D_2 - D_1$.
4. Repeat until no additional Phi functions can be added.


```
PlacePhi{
  For (each variable v ∈ program) {
     For (each block b ∈ CFG ){
        PhiInserted(b) = false
        Added(b) = false }
     List = φ
     For (each b ∈ CFG that assigns to V ){
        Added(b) = true
        List = List U {b}   }
     While (List ≠ φ) {
         Remove any b from List
         For (each d ∈ DF(b)){
             If (! PhiInserted(d)) {
                 Add a Phi Function to d
                 PhiInserted(d) = true
                 If (! Added(d)) {
                       Added(d) = true
                       List = List U {d}
                 }
             }
         }
     }
  }
}
```


Example



Initially, List={1,3,5,6}

Process 1: DF(1) = φ

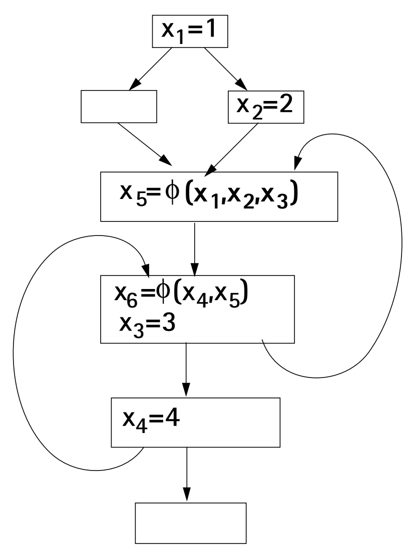
Process 3: DF(3) = 4, so add 4 to List and add phi fct to 4.

Process 5: DF(5)={4,5} so add phi fct to 5.

Process 5: DF(6) = {5}

Process 4: DF(4) = {4}

We will add Phi's into blocks 4 and 5. The arity of each phi is the number of in-arcs to its block. To find the args to a phi, follow each arc "backwards" to the sole reaching def on that path.



SSA and Value Numbering

We already know how to do available expression analysis to determine if a previous computation of an expression can be reused.

A limitation of this analysis is that it can't recognize that two expressions that aren't syntactically identical may actually still be equivalent.

For example, given

```
t1 = a + b
c = a
t2 = c + b
```

Available expression analysis won't recognize that `t1` and `t2` must be equivalent, since it doesn't track the fact that `a` = `c` at `t2`.

Value Numbering

An early expression analysis technique called *value numbering* worked only at the level of basic blocks. The analysis was in terms of "values" rather than variable or temporary names.

Each non-trivial (non-copy) computation is given a number, called its *value number*.

Two expressions, using the same operators and operands with the same value numbers, must be equivalent.

For example,

```
t1 = a + b
c = a
t2 = c + b
```

is analyzed as

```
v1 = a
v2 = b
t1 = v1 + v2
c = v1
t2 = v1 + v2
```

Clearly `t2` is equivalent to `t1` (and hence need not be computed).

In contrast, given

```
t1 = a + b
a = 2
t2 = a + b
```

the analysis creates

```
v1 = a
v2 = b
t1 = v1 + v2
v3 = 2
t2 = v3 + v2
```

Clearly `t2` is not equivalent to `t1` (and hence will need to be recomputed).

Extending Value Numbering to Entire CFGs

The problem with a global version of value numbering is how to reconcile values produced on different flow paths. But this is exactly what SSA is designed to do!

In particular, we know that an ordinary assignment

`x = y`

does *not* imply that all references to x can be replaced by y after the assignment. That is, an assignment *is not* an assertion of value equivalence.

But,

in SSA form

$x_i = y_j$

does mean the two values are *always* equivalent after the assignment. If y_j reaches a use of x_i, that use of x_i *can* be replaced with y_j.

Thus in SSA form, an assignment *is* an assertion of value equivalence.

We will assume that simple variable to variable copies are removed by substituting equivalent SSA names.

This alone is enough to recognize some simple value equivalences.

As we saw,

$t_1 = a_1 + b_1$

$c_1 = a_1$

$t_2 = c_1 + b_1$

becomes

$t_1 = a_1 + b_1$

$t_2 = a_1 + b_1$

Partitioning SSA Variables

Initially, all SSA variables will be partitioned by the *form* of the expression assigned to them.

Expressions involving different constants or operators won't (in general) be equivalent, even if their operands happen to be equivalent.

Thus

$v_1 = 2$ and $w_1 = a_2 + 1$

are always considered inequivalent.

But,

$v_3 = a_1 + b_2$ and $w_1 = d_1 + e_2$

may *possibly* be equivalent since both involve the same operator.

Phi functions are potentially equivalent only if they are in the same basic block.

All variables are initially considered equivalent (since they all initially are considered uninitialized until explicit initialization).

After SSA variables are grouped by assignment form, groups are split.

If a_i op b_y and c_k op d_l
are in the same group (because they both have the same operator, op)
and $a_i \not\equiv c_k$ or $b_j \not\equiv d_l$
then we split the two expressions apart into different groups.

We continue splitting based on operand inequivalence, until no more splits are possible. Values still grouped are equivalent.

Example

```
if (...) {
  a1=0
  if (...)
    b1=0
  else {
    a2=x0
    b2=x0 }
  a3=φ(a1,a2)
  b3=φ(b1,b2)
  c2=*a3
  d2=*b3 }
else {
  b4=10 }
a5=φ(a0,a3)
b5=φ(b3,b4)
c3=*a5
d3=*b5
e3=*a5
```

Initial Groupings:

G_1=[a_0,b_0,c_0,d_0,e_0,x_0]
G_2=[a_1=0, b_1=0]
G_3=[a_2=x_0, b_2=x_0]
G_4=[b_4=10]
G_5=[a_3=ϕ(a_1,a_2),
b_3=ϕ(b_1,b_2)]
G_6=[a_5=ϕ(a_0,a_3),
b_5=ϕ(b_3,b_4)]
G_7=[c_2=*a_3,
d_2=*b_3,
d_3=*b_5,
c_3=*a_5,
e_3=*a_5]

Now b_4 isn't equivalent to anything, so split a_5 and b_5. In G_7 split operands b_3, a_5 and b_5. We now have


```
if (...) {
  a1=0
  if (...)
    b1=0
  else {
    a2=x0
    b2=x0 }
  a3=φ(a1,a2)
  b3=φ(b1,b2)
  c2=*a3
  d2=*b3 }
else {
  b4=10 }
a5=φ(a0,a3)
b5=φ(b3,b4)
c3=*a5
d3=*b5
e3=*a5
```

Final Groupings:

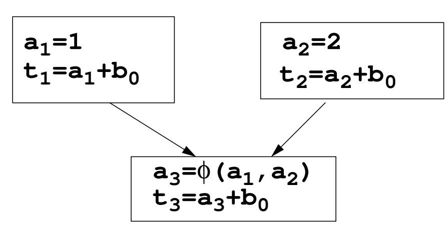
G_1=[a_0,b_0,c_0,d_0,e_0,x_0]
G_2=[a_1=0, b_1=0]
G_3=[a_2=x_0, b_2=x_0]
G_4=[b_4=10]
G_5=[a_3=ϕ(a_1,a_2),
b_3=ϕ(b_1,b_2)]
G_{6a}=[a_5=ϕ(a_0,a_3)]
G_{6b}=[b_5=ϕ(b_3,b_4)]
G_{7a}=[c_2=*a_3,
d_2=*b_3]
G_{7b}=[d_3=*b_5]
G_{7c}=[c_3=*a_5,
e_3=*a_5]

Variable e_3 can use c_3's value and d_2 can use c_2's value.

Limitations of Global Value Numbering

As presented, our global value numbering technique doesn't recognize (or handle) computations of the same expression that produce different values along different paths.

Thus in



variable a_3 isn't equivalent to either a_1 or a_2.

But,
we can still remove a redundant computation of **a+b** by moving the computation of t_3 to each of its predecessors:

$a_1=1$
$t_1=a_1+b_0$
$e_1=a_1+b_0$

$a_2=2$
$t_2=a_2+b_0$
$e_2=a_2+b_0$

$e_3=\phi(e_1,e_2)$
$t_3=e_3$

Now a redundant computation of **a+b** is evident in each predecessor block. Note too that this has a nice register targeting effect—e_1, e_2 and e_3 can be readily mapped to the same live range.

The notion of moving expression computations above phi functions also meshes nicely with notion of partial redundancy elimination. Given

$a_1=1$
$t_1=a_1+b_0$

$a_2=2$

$a_3=\phi(a_1,a_2)$
$t_3=a_3+b_0$

moving **a+b** above the phi produces

$a_1=1$
$t1=a_1+b_0$

$a_2=2$
$t_2=a_2+b_0$

$t_3=\phi(t_1,t_2)$

Now **a+b** is computed only once on each path, an improvement.

Reading Assignment

- Read "Global Optimization by Suppression of Partial Redundancies," Morel and Renvoise.
  (Linked from the class Web page.)
- Read “Profile Guided Code Positioning,” Pettis and Hansen.
  (Linked from the class Web page.)

Partial Redundancy Analysis

Partial Redundancy Analysis is a boolean-valued data flow analysis that generalizes available expression analysis.

Ordinary available expression analysis tells us if an expression must already have been evaluated (and not killed) along *all* execution paths.

Partial redundancy analysis, originally developed by Morel & Renvoise, determines if an expression has been computed along *some* paths. Moreover, it tells us where to add new computations of the expression to change a partial redundancy into a full redundancy.

This technique *never* adds computations to paths where the computation isn't needed. It strives to avoid having any redundant computation on any path.

In fact, this approach includes movement of a loop invariant expression into a preheader. This loop invariant code movement is just a special case of partial redundancy elimination.

Basic Definition & Notation

For a Basic Block i and a particular expression, e:

$Transp_i$ is true if and only if e's operands aren't assigned to in i.

$Transp_i \equiv \neg Kill_i$

$Comp_i$ is true if and only if e is computed in block i and is not killed in the block after computation.

$Comp_i \equiv Gen_i$

$AntLoc_i$ (Anticipated Locally in i) is true if and only if e is computed in i and there are no assignments to e's operands prior to e's computation.

If $AntLoc_i$ is true, computation of e in block i will be redundant if e is available on entrance to i.

We'll need some standard data flow analyses we've seen before:

$AvIn_i$ = Available In for block i

$= 0$ (false) for b_0

$= \underset{p \in Pred(i)}{AND} AvOut_p$

$AvOut_i = Comp_i$ OR $(AvIn_i$ AND $Transp_i)$

$\equiv Gen_i$ OR $(AvIn_i$ AND $\neg Kill_i)$

We *anticipate* an expression if it is very busy:

$\text{AntOut}_i = \text{VeryBusyOut}_i$

$= 0$ (false) if i is an exit block

$= \underset{s \in \text{Succ}(i)}{\text{AND}} \text{AntIn}_s$

$\text{AntIn}_i = \text{VeryBusyIn}_i$

$= \text{AntLoc}_i \text{ OR } (\text{Transp}_i \text{ AND } \text{AntOut}_i)$

Partial Availability

Partial availability is similar to available expression analysis except that an expression must be computed (and not killed) along *some* (not necessarily *all*) paths:

PavIn_i

$= 0$ (false) for b_0

$= \underset{p \in \text{Pred}(i)}{\text{OR}} \text{PavOut}_p$

$\text{PavOut}_i = \text{Comp}_i \text{ OR } (\text{PavIn}_i \text{ AND } \text{Transp}_i)$

Where are Computations Added?

The key to partial redundancy elimination is deciding where to add computations of an expression to change partial redundancies into full redundancies (which may then be optimized away).

We'll start with an "enabling term."

$\text{Const}_i = \text{AntIn}_i \text{ AND } [\text{PavIn}_i \text{ OR } (\text{Transp}_i \text{ AND } \neg\, \text{AntLoc}_i)]$

This term say that we require the expression to be:

(1) Anticipated at the start of block i (somebody wants the expression)

and

(2a) The expression must be partially available (to perhaps transform into full availability)

or

(2b) The block neither kills nor computes the expression.

Next, we compute $PPIn_i$ and $PPOut_i$.

PP means "possible placement" of a computation at the start ($PPIn_i$) or end ($PPOut_i$) of a block.

These values determine whether a computation of the expression would be "useful" at the start or end of a basic block.

$PPOut_i$

= 0 (false) for all exit blocks

$$= \underset{s \in Succ(i)}{\text{AND}} \; PPIn_s$$

We try to move computations "up" (nearer the start block).

It makes sense to compute an expression at the end of a block if it makes sense to compute at the start of all the block's successors.

$PPIn_i$ = 0 (false) for b_0.

$$= Const_i \text{ AND } (AntLoc_i \text{ OR } (Transp_i \text{ AND } PPOut_i)) \underset{p \in Pred(i)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p)$$

To determine if $PPIn_i$ is true, we first check the enabling term. It makes sense to consider a computation of the expression at the start of block i if the expression is anticipated (wanted) and partially available or if the expression is anticipated (wanted) and it is neither computed nor killed in the block.

We then check that the expression is anticipated locally or that it is unchanged within the block and possibly positioned at the end of the block.

Finally, we check that all the block's predecessors either have the expression available at their ends or are willing to position a computation at their end.

Note also, the bi-directional nature of this equation.

Inserting New Computations

After $PPIn_i$ and $PPOut_i$ are computed, we decide where computations will be inserted:

$$Insert_i = PPOut_i \text{ AND } (\neg AvOut_i) \text{ AND } (\neg PPIn_i \text{ OR } \neg Transp_i)$$

This rule states that we really will compute the expression at the end of block i if this is a possible placement point and the expression is not already computed and available and moving the computation still earlier doesn't work because the start of the block isn't a possible placement point or because the block kills the expression.

Removing Existing Computations

We've added computations of the expression to change partial redundancies into full redundancies. Once this is done, expressions that are fully redundant can be removed.

But where?

$\text{Remove}_i = \text{AntLoc}_i \text{ and } \text{PPIn}_i$

This rule states that we remove computation of the expression in blocks where it is computed locally and might be moved to the block's beginning.

Partial Redundancy Subsumes Available Expression Analysis

Using partial redundancy analysis, we can find (and remove) ordinary fully redundant available expressions.

Consider a block, b, in which:

(1) The expression is computed (anticipated) locally

and

(2) The expression is available on entrance

Point (1) tells us that AntLoc_b is true

Moreover, recall that

$$\text{PPIn}_b = \text{Const}_b \text{ AND } (\text{AntLoc}_b \text{ OR} \ldots) \text{ AND}_{p \in \text{Pred(i)}} (\text{AvOut}_p \text{ OR} \ldots)$$

$\text{Const}_b = \text{AntIn}_b$ AND $[\text{PavIn}_b$ OR ...$]$

We know AntLoc_b is true $\Rightarrow \text{AntIn}_b =$ true.

Moreover, $\text{AvIn}_b = \text{true} \Rightarrow \text{PavIn}_b = \text{true}$.

Thus $\text{Const}_b = \text{true}$.

If AvIn_b is true, AvOut_p is true for all $p \in \text{Pred}(b)$.

Thus PPIn_b AND AntLoc_b = true = Remove_b

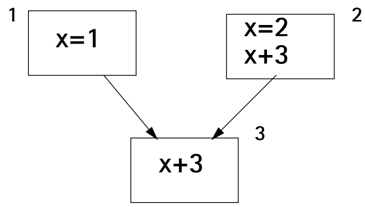
Are any computations added earlier (to any of b's ancestors)?

No:

$$\text{Insert}_i = \text{PPOut}_i \text{ AND } (\neg\, \text{AvOut}_i) \text{ AND } (\neg\, \text{PPIn}_i \text{ OR } \neg\, \text{Transp}_i)$$

But for any ancestor, i, between the computation of the expression and b, AvOut_i is true, so Insert_i must be false.

Examples of Partial Redundancy Elimination



At block 3, x+3 is partially, but not fully, redundant.

$PPIn_3 = Const_3$ AND ($AntLoc_3$ OR ...)

$\underset{p \in Pred(3)}{\text{AND}}$ ($PPOut_p$ OR $AvOut_p$)

$Const_3 = AntIn_3$ AND [$PavIn_3$ OR ...]
Now $AntIn_3$ = true and $PavIn_3$ = true.

$Const_3$ = true AND true = true

$PPout_1 = PPIn_3$

Default initialization of PPIn and PPOut terms is true, since we AND terms together.

$AntLoc_3$ = true.

$PPIn_3$ = true AND true

$\underset{p \in Pred(3)}{\text{AND}}$ ($PPOut_p$ OR $AvOut_p$) =

$PPOut_1$ AND $AvOut_2$= true AND true = $PPIn_3 = PPOut_1$.

$Insert_1 = PPOut_1$ AND ($\neg$ $AvOut_1$) AND ($\neg$ $PPIn_1$ OR $\neg$ $Transp_1$) =

$PPOut_1$ AND ($\neg$ $AvOut_1$) AND ($\neg$ $Transp_1$) = true,

so x+3 is inserted at the end of block 3.

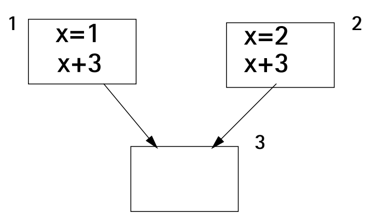
$Remove_3 = AntLoc_3$ and $PPIn_3$ = true AND true = true, so x+3 is removed from block 3.

Is x+3 inserted at the end of block 2? (It shouldn't be).

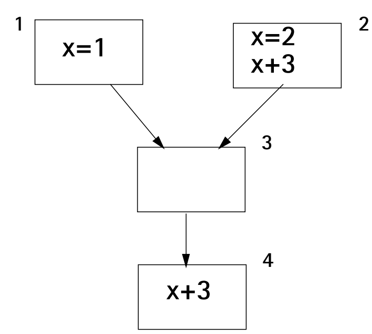
$Insert_2 = PPOut_2$ AND ($\neg$ $AvOut_2$) AND ($\neg$ $PPIn_2$ OR $\neg$ $Transp_2$) =

$PPOut_2$ AND false AND ($\neg$ $PPIn_2$ OR $\neg$ $Transp_2$) = false.

We now have



Computations May Move Up Several Blocks



Again, at block 4, x+3 is partially, but not fully, redundant.

$PPIn_4 = Const_4$ AND ($AntLoc_4$ OR ...)

$\underset{p \in Pred(4)}{\text{AND}}$ ($PPOut_p$ OR $AvOut_p$)

$Const_4 = AntIn_4$ AND [$PavIn_4$ OR ...]
Now $AntIn_4$ = true and $PavIn_4$ = true.

$Const_4$ = true AND true = true

$PPout_3 = PPIn_4$.

$AntLoc_4$ = true.

$PPIn_4$ = true AND true

$$\underset{p \in Pred(4)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p) =$$

$PPOut_3$ = true.

$PPIn_3 = Const_3$ AND
((${Transp}_3$ AND $PPOut_3$) OR ...)

$$\underset{p \in Pred(3)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p)$$

$Const_3 = AntIn_3$ AND [$PavIn_3$ OR ...]
$AntIn_3$ = true and $PavIn_3$ = true.

$Const_3$ = true AND true = true

$PPOut_1 = PPIn_3$

$Transp_3$ = true.

$PPIn_3$ = true AND (true AND true)

$$\underset{p \in Pred(3)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p) =$$

$PPOut_1$ AND $AvOut_2$= true AND true
= $PPIn_3 = PPOut_1$.

Where Do We Insert Computations?

$Insert_3 = PPOut_3$ AND ($\neg$ $AvOut_3$)
AND ($\neg$ $PPIn_3$ OR $\neg$ $Transp_3$) =

true AND (true) AND
(false OR false) = false

so x+3 is *not* inserted at the end of block 3.

$Insert_2 = PPOut_2$ AND ($\neg$ $AvOut_2$)
AND ($\neg$ $PPIn_2$ OR $\neg$ $Transp_2$) =

$PPOut_2$ AND (false)
AND ($\neg$ $PPIn_2$ OR $\neg$ $Transp_2$)=false,
so x+3 is *not* inserted at the end of block 2.

$Insert_1 = PPOut_1$ AND ($\neg$ $AvOut_1$)
AND ($\neg$ $PPIn_1$ OR $\neg$ $Transp_1$) =

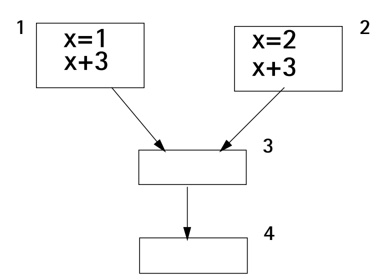
true AND (true) AND
($\neg$ $PPIn_1$ OR true) = true

so x+3 *is* inserted at the end of block 3.

$Remove_4 = AntLoc_4$ and $PPIn_4$

= true AND true = true, so x+3 is removed from block 4.

We finally have



Code Movement is Never Speculative

Partial redundancy analysis has the attractive property that it never adds a computation to an execution path that doesn't use the computation.

That is, we never *speculatively* add computations.

How do we know this is so?

Assume we are about to insert a computation of an expression at the end of block b, but there is a path from b that doesn't later compute and use the expression.

Say the path goes from b to c (a successor of b), and then eventually to an end node.

Looking at the rules for insertion of an expression:

$Insert_b = PPOut_b$ AND ...

$PPOut_b = PPIn_c$ AND ...

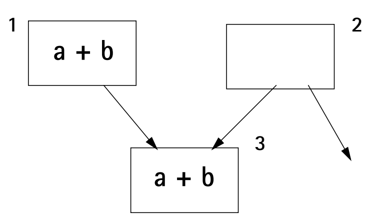
$PPIn_c = Const_c$ AND ...

$Const_c = AntIn_c$ AND ...

But if the expression isn't computed and used on the path through c, then $AntIn_c$ = False, forcing $Insert_b$ = false, a contradiction.

Can Computations Always be Moved Up?

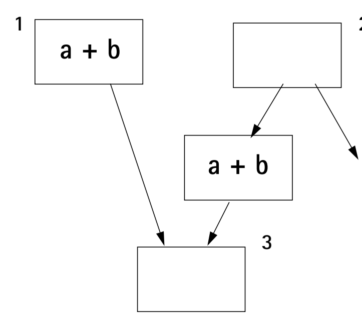
Sometimes an attempt to move a computation earlier in the CFG can be *blocked*. Consider



We'd like to move a+b into block 2, but this may be impossible if a+b isn't anticipated on all paths out of block 2.

The solution to this difficulty is no notice that we really want a+b computed on the *edge* from 2 to 3.

If we add an *artificial* block between blocks 2 and 3, movement of a+b out of block 3 is no longer blocked:



Loop Invariant Code Motion

Partial redundancy elimination subsumes loop invariant code motion.

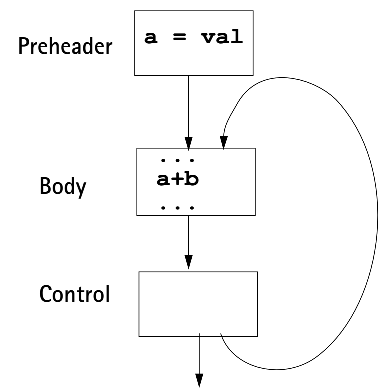
Why?

The iteration of the loop makes the invariant expression partially redundant on a path from the expression to itself.

If we're guaranteed the loop will iterate at least once (do-while or repeat-until loops), then evaluation of the expression can be anticipated in the loop's preheader.

Consider

```
a = val
do
     ...
     a+b
     ...
while (...)
```



$PPIn_B$ = $Const_B$ AND ($AntLoc_B$ OR ...) AND ($PPOut_p$ AND $AvOut_C$)

$Const_B$ = $AntIn_B$ AND [$PavIn_B$ OR ...]

$AntIn_B$ = true, $PavIn_B$ = true $\Rightarrow$ $Const_B$ = true

$PPout_P$ = $PPIn_B$, $AntLoc_B$= true, $AvOut_C$ = true $\Rightarrow$ $PPIn_B$ = true.

$Insert_P$ = $PPOut_P$ AND ($\neg$ $AvOut_P$) AND ($\neg$ $PPIn_P$ OR $\neg$ $Transp_P$) =

true AND (true) AND ($\neg$ $PPIn_P$ OR true) = true,

so we may insert a+b at the end of the preheader.

$Remove_B$ = $AntLoc_B$ and $PPIn_B$ = true AND true, so we may remove a+b from the loop body.

What About While & For Loops?

The problem here is that the loop may iterate zero times, so the loop invariant isn't really very busy (anticipated) in the preheader.

We can, however, change a while (or for) into a do while:

```
while (expr){    if (expr)
   body       ≡    do {body}     ≈
}                  while (expr)


goto L:
  do {body}
 L:
  while (expr)
```

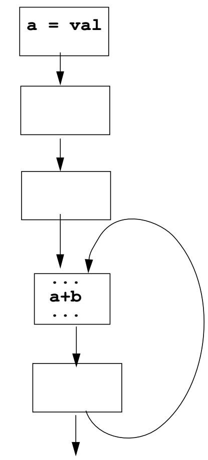
After we know the loop will iterate once, we can evaluate the loop invariant.

Code Placement in Partial Redundancy Elimination

While partial redundancy elimination correctly places code to avoid unnecessary reevaluation of expressions along execution paths, its choice of code placement can sometimes be disappointing.

It always moves an expression back as far as possible, as long as computations aren't added to unwanted execution paths. This may unnecessarily lengthen live ranges, making register allocation more difficult.

For example, in



where will we insert a+b?

$\text{Insert}_P = \text{PPOut}_P \text{ AND } (\neg \text{ AvOut}_P)$
$\quad \text{AND } (\neg \text{ PPIn}_P \text{ OR } \neg \text{ Transp}_P)$

The last term will be true at the top block, but not elsewhere.

In “Lazy Code Motion” (PLDI 1992), Knoop, Ruething and Steffan show how to eliminate partial redundancies while minimizing register pressure.

Their technique seeks to evaluate an expression as “late as possible” while still maintaining computational optimality (no redundant or unnecessary evaluations on *any* execution paths).

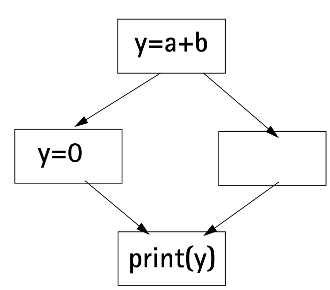
Their technique places loop invariants in the loop preheader rather than in an earlier predecessor block as Morel & Renvoise do.

Partial Dead Code Elimination

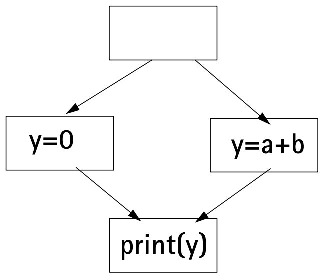
Partial Redundancy Elimination aims to never reevaluate an expression on any path, and never to add an expression on any path where it isn't needed.

These ideas suggest an interesting related optimization—eliminating expressions that are partially dead.

Consider



On the left execution path, a+b is dead, and hence useless. We'd prefer to compute a+b only on paths where it is used, obtaining



This optimization is investigated in "Partial Dead Code Elimination" (PLDI 1994), Knoop, Ruething and Steffan.

This optimization "sinks" computations onto paths where they are needed.

CS 701 Final Exam (Reminder)

Thursday, December 11, 11:00 a.m.— 1:00 p.m., in class.

Procedure & Code Placement

We have seen many optimizations that aim to reduce the number of instructions executed by a program.

Another important class of optimizations derives from the fact that programs often must be paged in virtual memory and almost always are far bigger then the I-cache.

Hence how procedures and basic blocks are placed in memory is important. Page faults and I-cache misses can be *very* costly.

In "Profile Guided Code Positioning," Pettis and Hansen explore three kinds of code placement optimizations:

1. Procedure Positioning.

   Try to keep procedures that often call each other close together.

2. Basic Block Positioning.

   Try to place the most frequently executed series of basic blocks "in sequence."

3. Procedure Splitting.

   Place infrequently executed "fluff" in a different memory area than heavily executed code.

Procedure Placement

Procedures (and classes in Java) are normally separately compiled. They are then placed in memory by a linker or loader in an arbitrary order.

This arbitrary ordering can be problematic:

If A calls B frequently, and A and B happen to be placed far apart in memory, the calls will cross page boundaries and perhaps cause I-cache conflicts (if code in A and B happen to map to common cache locations).

However,
if A and B are placed close together in memory, they may both fit on the same page *and* fit into the I-cache without conflicts.

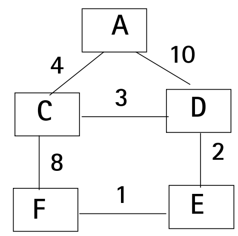
Pettis & Hansen suggest a "closest is best" procedure placement policy.

That is, they recommend that we place procedures that often call each other as close together as possible.

How?

First, we must obtain dynamic call frequencies using a profiling tool like gprof or qpt.

Given call frequencies, we create a call graph, with edges annotated with call frequencies:

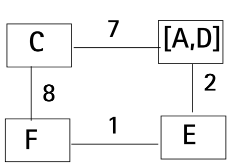


Group Procedures by Call Frequency

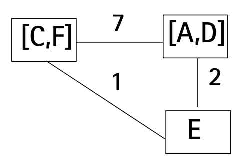
We find the pair of procedures that call each other most often, and group them for contiguous positioning.

The notation [A,D] means A and D will be adjacent (either in order A-D or D-A).

The two procedures chosen are combined in the call graph, which is simplified (much like move-related nodes in an interference graph):



Now C and F are grouped, without their relative order set (as yet):



Next [A,D] and [C,F] are to be joined, but in what exact order?

Four orderings are possible:

A-D-C-F ≡ F-C-D-A
A-D-F-C ≡ C-F-D-A
D-A-C-F ≡ F-C-A-D
D-A-F-C ≡ C-F-A-D

Are these four orderings equivalent?

No—Look at the original call graph. At the boundary between [A,D] and [C,F], which of the following is best:

D-C (3 calls),

D-F (0 calls)

A-C (4 calls)

A-F (0 calls)

A-C has the highest call frequency, so we choose D-A-C-F.

Finally, we have:

$\boxed{\text{D-A-C-F}} \xrightarrow{\;3\;} \boxed{\text{E}}$

We place E near D (call frequency 2) rather than near F (call frequency 1).

Our final ordering is
E-D-A-C-F.

Basic Block Placement

We often see conditionals of the form

if (error-test)
{Handle error case}
{Rest of Program}

Since error tests rarely succeed (we hope!), the error handling code "pollutes" the I-cache.

In general, we'd like to order basic blocks not in their order of appearance in the source program, but rather in order of their execution along frequently executed paths.

Placing frequently executed basic blocks together in memory fills the I-cache nicely, leads to a smaller working set *and* makes branch prediction easier.

Pettis & Hansen suggest that we profile execution to determine the frequency of inter-block transitions. We then will group blocks together that execute in sequence most often.

At the start, all basic blocks are grouped into singleton chains of one block each.

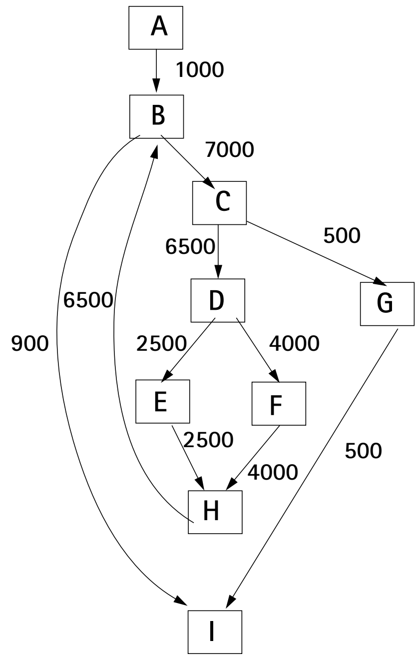
Then, in decreasing order of transition frequency, we visit arcs in the CFG.

If the blocks in the source and target can be linked into a longer chain then do so, else skip to the next transition.

When we are done, we have linked together blocks in paths in the CFG that are most frequently executed.

Linked basic blocks are allocated together in memory, in the sequence listed in the chain.

Example



Initially, each bock is in its own chain.

Frequency	Action
7000	Form B-C
6500	Form B-C-D
6500	Form H-B-C-D
4000	Form H-B-C-D-F
4000	H is already placed
2500	E can't be placed after D, leave it alone
2500	H is already placed
1000	A can't be placed before B, leave it alone
900	I can't be placed after B, leave it alone
500	G can't be placed after C, leave it alone
500	Form G-I

We will place in memory the following chains of basic blocks:

H-B-C-D-F, E, A, G-I

On some computers, the direction of a conditional branch predicts whether the branch is expected to be taken or not (e.g., the HP PA-RISC). On such machines, a backwards branch (forming a loop) is assumed taken; a forward branch is assumed not taken.

If the target architecture makes such assumptions regarding conditional branches, we place chains to (where possible) correctly predict the branch outcome.

Thus E and G-I are placed after H-B-C-D-F since D→E and C→G normally aren't taken.

On the SPARC (V 9) you can set a bit in each conditional branch indicating expected taken/not taken status.

On many machines internal branch prediction hardware can over-rule poorly made (or absent) static predictions.

Procedure Splitting

When we profile the basic blocks within a procedure, we'll see some that are frequently executed, and others that are executed rarely or never.

If we allocate all the blocks of a procedure contiguously, we'll intermix frequently executed blocks with infrequently executed ones.

An alternative is "fluff removal." We can split a procedure's body into two sets of basic blocks: these executed frequently and those executed infrequently (the dividing line is, of course, somewhat arbitrary).

Now when procedure bodies are placed in memory, frequently executed basic blocks will be placed near each other, and infrequently executed blocks will be placed elsewhere (though infrequently executed blocks are still placed near each other). In this way be expect to make better use of page frames and I-cache space, filling them with mostly active basic blocks.