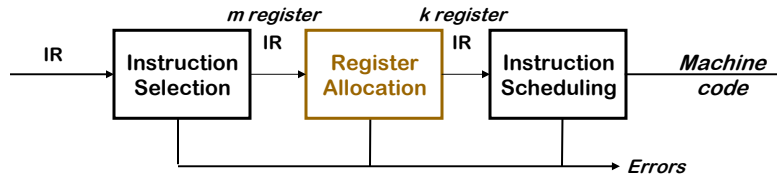


Register Allocation

Part of the compiler's back end



Critical properties

- Produce correct code that uses k (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold *spilled values*
- Operate efficiently
 $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

CMSC430 Spring 2007

1

Global Register Allocation

The big picture



At each point in the code

- 1 Determine which values will reside in registers
- 2 Select a register for each such value

The goal is an allocation that "minimizes" running time

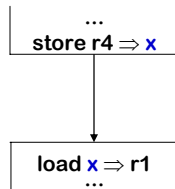
Most modern, global allocators use a graph-coloring paradigm

- Build a "**conflict graph**" or "**interference graph**"
- Find a k -coloring for the graph, or change the code to a nearby problem that it can k -color

CMSC430 Spring 2007

2

Global Register Allocation



This is an assignment problem,
not an allocation problem !

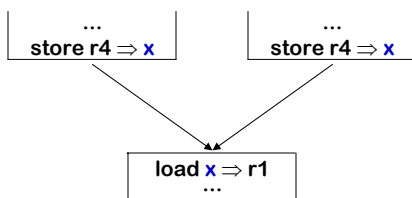
What's harder across multiple blocks?

- Could replace a load with a move
- Good assignment would obviate the move
- Must build a control-flow graph to understand inter-block flow
- Can spend an inordinate amount of time adjusting the allocation

CMSC430 Spring 2007

3

Global Register Allocation



What if one block has x in a
register, but the other does not?

A more complex scenario

- Block with multiple predecessors in the control-flow graph
- Must get the “right” values in the “right” registers in each predecessor
- In a loop, a block can be its own predecessors

This adds tremendous complications

CMSC430 Spring 2007

4

Global Register Allocation

Taking a global approach

- Abandon the distinction between local & global
- Make systematic use of registers or memory
- Adopt a general scheme to approximate a good allocation

Graph coloring paradigm (Lavrov & (later) Chaitin)

- 1 Build an interference graph G , for the procedure
 - > Computing LIVE is harder than in the local case
 - > G is not an interval graph
- 2 (try to) construct a k -coloring
 - > Minimal coloring is NP-Complete
 - > Spill placement becomes a critical issue
- 3 Map colors onto physical registers

CMSC430 Spring 2007

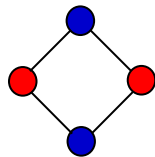
5

Graph Coloring (A Background Digression)

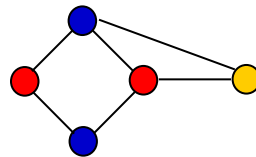
The problem

A graph G is said to be k -colorable iff the nodes can be labeled with integers $1 \dots k$ so that no edge in G connects two nodes with the same label

Examples



2-colorable



3-colorable

Each color can be mapped to a distinct physical register

CMSC430 Spring 2007

6

Building the Interference Graph

What is an “interference” ? (or conflict)

- Two values interfere if there exists an operation where both are simultaneously live
- If x and y interfere, they cannot occupy the same register

To compute interferences, we must know where values are “live”

The interference graph, G_I ,

- Nodes in G_I represent values, or live ranges
- Edges in G_I represent individual interferences
 - > For $x, y \in G_I$, $\langle x, y \rangle \in$ iff x and y interfere
- A k -coloring of G_I can be mapped into an allocation to k registers

Building the Interference Graph

To build the interference graph

- 1 Discover live ranges
 - > Build SSA form
 - > At each ϕ -function, take the union of the arguments
- 2 Compute LIVE sets for each block
 - > Use an iterative data-flow solver
 - > Solve equations for LIVE over domain of live range names
- 3 Iterate over each block
 - > Track the current LIVE set
 - > At each operation, add appropriate edges & update LIVE
 - Edge from result to each value in LIVE
 - Remove result from LIVE
 - Edge from each operand to each value in LIVE

What is a Live Range?

- A set LR of definitions $\{d_1, d_2, \dots, d_n\}$ such that for any two definitions d_i and d_j in LR, there exists some use u that is reached by both d_i and d_j .
- How can we compute live ranges?
 - > For each basic block b in the program, compute **REACHESOUT(b)**—the set of definitions that reach the exit of basic block b
 - $d \in \text{REACHESOUT}(b)$ if there is no other definition on some path from d to the end of block b
 - > For each basic block b , compute **LIVEIN(b)**—the set of variables that are live on entry to b
 - $v \in \text{LIVEIN}(b)$ if there is a path from the entry of b to a use of v that contains no definition of v
 - > At any block where control flow joins, for each live variable v , merge the live ranges associated with definitions in **REACHESOUT(p)**, for all predecessors of b , that assign a value to v .

CMSC430 Spring 2007

9

Computing LIVE Sets

A value v is live at p if \exists a path from p to some use of v along which v is not re-defined

Data-flow problems are expressed as simultaneous equations

$$\text{LIVEOUT}(b) = \bigcup_{s \in \text{succ}(b)} \text{LIVEIN}(s)$$

$$\text{LIVEIN}(b) = (\text{LIVEOUT}(b) \cap \text{NOTDEF}(b)) \cup \text{IN}(b)$$

where

IN(x) is the set of names used before redefinition in block x

NOTDEF(x) is the set of names not redefined in x

As output,

LIVEOUT(x) is the set of names live on exit from block x

LIVEIN(x) is the set of names live on entry to block x

CMSC430 Spring 2007

10

Computing LIVE Sets

The compiler solve the equations with an iterative algorithm

```
WorkList ← { all blocks }
while ( WorkList ≠ ∅ )
  remove a block b from WorkList
  Compute LIVEOUT(b)
  Compute LIVEIN(b)
  if LIVEIN(b) changed
    then add pred (b) to WorkList
```

The Worklist Iterative
Algorithm

Why does this work?

- LIVEOUT, LIVEIN $\subseteq 2^{\text{Names}}$
- IN, NOTDEF are constant for b
- Equations are monotone
- Finite chains in the lattice
⇒ will reach a fixed point !

Speed of convergence
depends on the order in which
blocks are “removed” & their
sets recomputed

This is the world’s quickest introduction to data-flow analysis !

Observation on Coloring for Register Allocation

- Suppose you have k registers—try to color the graph with k colors
- Any vertex n that has fewer than k neighbors in the interference graph ($n^\circ < k$) can **always** be colored!
 - > Pick any color not used by its neighbors — there must be one
- Idea for Chaitin’s algorithm:
 - > Pick any vertex n such that $n^\circ < k$ and put it on the stack
 - > Remove that vertex and all edges incident from the interference graph
 - This may make some new nodes have fewer than k neighbors
 - > At the end, if some vertex n still has k or more neighbors, then spill the live range associated with n
 - > Otherwise successively pop vertices off the stack and color them in the lowest color not used by some neighbor

Chaitin's Algorithm

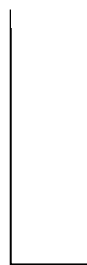
1. While there are vertices with fewer than k neighbors remaining in the interference graph G_i
 - > Pick any vertex n such that $n^d < k$ and put it on the stack
 - > Remove that vertex and all edges incident to it from G_i
 - This may cause additional vertices to have fewer than k neighbors
2. If any vertices remain in the interference graph G_i (all such vertices have k or more neighbors) then:
 - > Pick a vertex n (using some heuristic condition) and spill the live range associated with n
 - > Remove vertex n from G_i , along with all edges incident to it and put it on the stack
 - > If this causes some vertex in G_i to have fewer than k neighbors, then go to step 1; otherwise, repeat step 2
3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor

CMSC430 Spring 2007

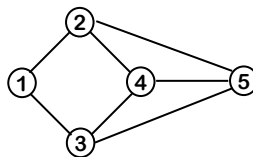
13

Chaitin's Algorithm in Practice

3 Registers



Stack

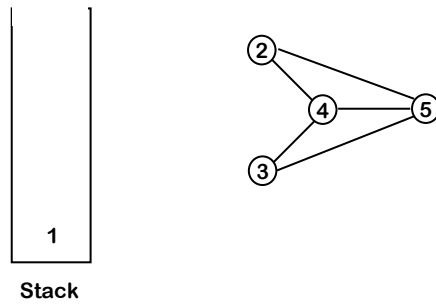


CMSC430 Spring 2007

14

Chaitin's Algorithm in Practice

3 Registers

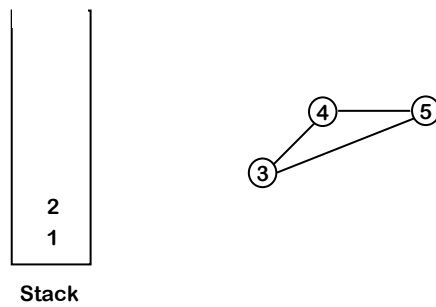


CMSC430 Spring 2007

15

Chaitin's Algorithm in Practice

3 Registers

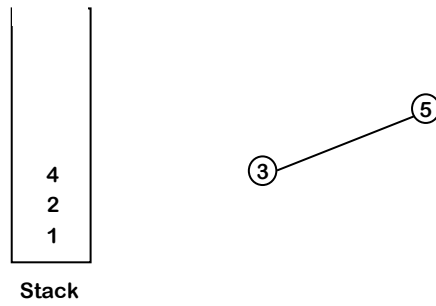


CMSC430 Spring 2007

16

Chaitin's Algorithm in Practice

3 Registers



CMSC430 Spring 2007

17

Chaitin's Algorithm in Practice

3 Registers

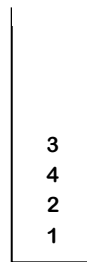


CMSC430 Spring 2007

18

Chaitin's Algorithm in Practice

3 Registers



Stack

5

Colors:

1: ●

2: ●

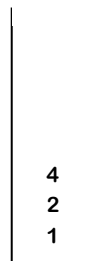
3: ●

CMSC430 Spring 2007

19

Chaitin's Algorithm in Practice

3 Registers



Stack

3

5

Colors:

1: ●

2: ●

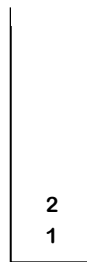
3: ●

CMSC430 Spring 2007

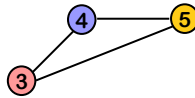
20

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

1: ●

2: ●

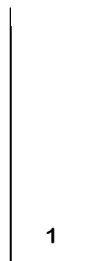
3: ●

CMSC430 Spring 2007

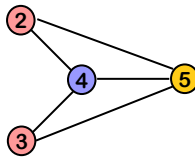
21

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

1: ●

2: ●

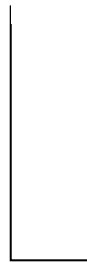
3: ●

CMSC430 Spring 2007

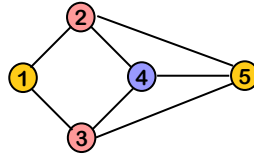
22

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

1: 

2: 

3: 

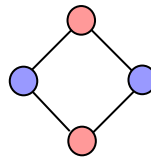
CMSC430 Spring 2007

23

Improvement in Coloring Scheme

- Due to Briggs, Cooper, Kennedy, and Torczon
- Instead of stopping at the end when all vertices have at least k neighbors, put each on the stack according to some priority
 - > When you pop them off they may still color!

2 Registers:



2-colorable

CMSC430 Spring 2007

24

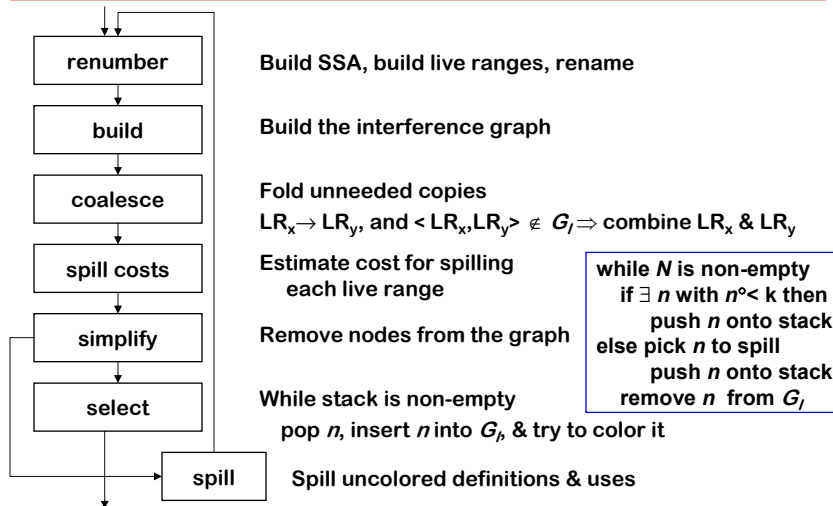
Chaitin-Briggs Algorithm

1. While there are vertices with fewer than k neighbors remaining in the interference graph G_i
 - > Pick any vertex n such that $n^{\circ} < k$ and put it on the stack
 - > Remove that vertex and all edges incident to it from G_i
 - This may create vertices with fewer than k neighbors
2. If any vertices remain in the interference graph G_i (all such vertices have k or more neighbors) then:
 - > Pick a vertex n (using some heuristic condition), put it on the stack and remove vertex n from G_i , along with all edges incident to it
 - > If this causes some vertex in G_i to have fewer than k neighbors, then go to step 1; otherwise, repeat step 2
3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor
 - > If some vertex cannot be colored, then pick a live range to spill, spill it, and restart at step 1

CMSC430 Spring 2007

25

Chaitin Allocator (Bottom-up Coloring)

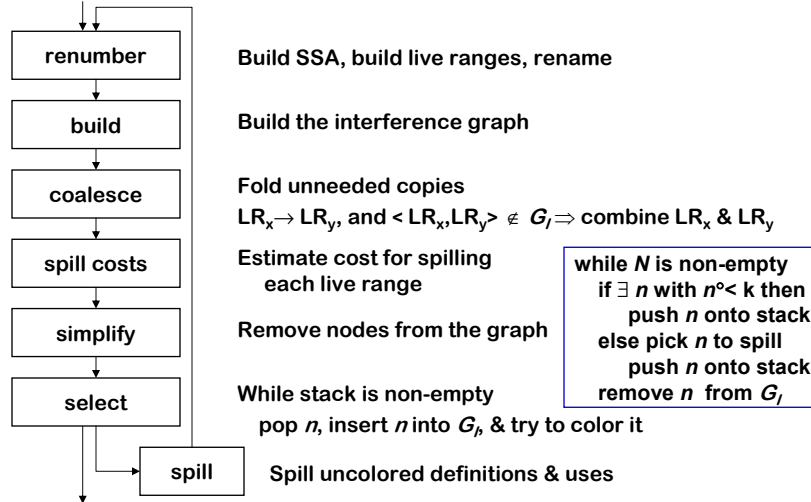


Chaitin's algorithm

CMSC430 Spring 2007

26

Chaitin-Briggs Allocator (Bottom-up Coloring)



Briggs' algorithm (1989)

CMSC430 Spring 2007

27

Picking a Spill Candidate

When $\exists n \in G_b, n^o \geq k$, simplify must pick a spill candidate

Chaitin's heuristic

- Minimize spill cost \div current degree
- If LR_x has a negative spill cost, spill it pre-emptively
 - > Cheaper to spill it than to keep it in a register
- If LR_x has an infinite spill cost, it cannot be spilled
 - > No value dies between its definition & its use

Spill cost is weighted cost of loads & stores needed to spill x

Bernstein *et al.* Suggest repeating simplify, select, & spill with several different spill choice heuristics & keeping the best

CMSC430 Spring 2007

28

Other Improvements to Chaitin-Briggs

Spilling partial live ranges

- Bergner introduced interference region spilling
- Limits spilling to regions of high demand for registers

Splitting live ranges

- Simple idea — break up one or more live ranges
- Lets allocator use different registers for distinct subranges
- Lets allocator spill subranges independently (use 1 spill location)

Conservative coalescing

- Combining $LR_x \rightarrow LR_y$ to form LR_{xy} may increase register pressure
- Limit coalescing to case where $LR_{xy}^\circ < k$
- Iterative form tries to coalesce before spilling

Chaitin-Briggs Allocator (Bottom-up Global)

Strengths & weaknesses

- ↑ Precise interference graph
- ↑ Strong coalescing mechanism
- ↑ Handles register assignment well
- ↑ Runs fairly quickly
- ↓ Known to overspill in tight cases
- ↓ Interference graph has no geography
- ↓ Spills a live range everywhere
- ↓ Long blocks devolve into spilling by use counts

Is improvement possible ?

- With rising spill costs, aggressive transformations, & long blocks

What about Top-down Coloring?

- The Big Picture
 - Use high-level priorities to rank live ranges
 - Allocate registers for them in priority order
 - Use coloring to assign specific registers to live ranges
- Use spill costs as priority function !
- The Details
 - Separate constrained from unconstrained live ranges
 - A live range is **constrained** if it has $\geq k$ neighbors in G_l
 - Color constrained live ranges first
 - Reserve pool of local registers for spilling (or spill & iterate)
 - Chow split live ranges before spilling them
 - > Split into block-sized pieces
 - > Recombine as long as $\rho < k$
- Unconstrained must receive a color !

CMSC430 Spring 2007

31

Tradeoffs in Global Allocator Design

Top-down versus bottom-up

- Top-down uses high-level information
- Bottom-up uses low-level structural information

Spilling

- Reserve registers versus iterative coloring

Precise versus imprecise graph

- Precision allows coalescing
- Imprecision speeds up graph construction

Big-iron \Rightarrow precise, iterated, bottom-up
JIT \Rightarrow imprecise, reserve, b-u or t-d

CMSC430 Spring 2007

32

Regional Approaches to Allocation

Hierarchical Register Allocation (Koblenz & Callahan)

- Analyze control-flow graph to find hierarchy of tiles
- Perform allocation on individual tiles, innermost to outermost
- Use summary of tile to allocate surrounding tile
- Insert compensation code at tile boundaries ($LR_x \rightarrow LR_y$)

Strengths

- Decisions are largely local
- Use specialized methods on individual tiles
- Allocator runs in parallel

Weaknesses

- Decisions are made on local information
- May insert too many copies
- Still, a promising idea

- Anecdotes suggest it is fairly effective
- Target machine is multi-threaded multiprocessor (Tera MTA)

Regional Approaches to Allocation

Probabilistic Register Allocation (Proebsting & Fischer)

- Attempt to generalize from Best's algorithm (bottom-up, local)
- Generalizes "furthest next use" to a probability
- Perform an initial local allocation using estimated probabilities
- Follow this with a global phase
 - > Compute a merit score for each LR as
(benefit from x in a register = probability it stays in a register)
 - > Allocate registers to LRs in priority order, by merit score, working from inner loops to outer loops
 - > Use coloring to perform assignment among allocated LRs
- Little direct experience (either anecdotal or experimental)
- Combines top-down global with bottom-up local

Regional Approaches to Allocation

Register Allocation via Fusion (Lueh, Adl-Tabataba, Gross)

- Use regional information to drive global allocation
- Partition CFGs into regions & build interference graphs
- Ensure that each region is k -colorable
- Merge regions by fusing them along CFG edges
 - > Maintain k -colorability by splitting along fused edge
 - > Fuse in priority order computed during the graph partition
- Assign registers using int. graphs *i.e.*, execution frequency

Strengths

- Flexibility
- Fusion operator splits on low-frequency edges

Weaknesses

- Choice of regions is critical
- Breaks down if region connections have many live values

CMSC430 Spring 2007

35

List Scheduling

```
Cycle ← 1
Ready ← leaves of P
Active ← ∅

while (Ready ∪ Active ≠ ∅)
  if (Ready ≠ ∅) then
    remove an op from Ready
    S(op) ← Cycle
    Active ← Active ∪ op
  Cycle ← Cycle + 1

  for each op ∈ Active
    if (S(op) + delay(op) ≤ Cycle) then
      remove op from Active
      for each successor s of op in P
        if (s is ready) then
          Ready ← Ready ∪ s
```

Removal in priority order

Note: only one op per cycle

op has completed execution

If successor's operands are ready, put it on Ready

CMSC430 Spring 2007

36

Detailed Scheduling Algorithm I

Idea: Keep a collection of worklists $W[c]$, one per cycle

> We need $\text{MaxC} = \text{max delay} + 1$ such worklists

Code:

```
for each  $n \in N$  do begin count[n] := 0; earliest[n] = 0 end
for each  $(n1, n2) \in E$  do begin
  count[n2] := count[n2] + 1;
  successors[n1] := successors[n1]  $\cup$  {n2};
end
for  $i := 0$  to  $\text{MaxC} - 1$  do  $W[i] := \emptyset$ ;
Wcount := 0;
for each  $n \in N$  do
  if count[n] = 0 then begin
     $W[0] := W[0] \cup \{n\}$ ; Wcount := Wcount + 1;
  end
end
c := 0; // c is the cycle number
cW := 0; // cW is the number of the worklist for cycle c
instr[c] :=  $\emptyset$ ;
```

CMSC430 Spring 2007

37

Detailed Scheduling Algorithm II

```
while Wcount > 0 do begin
  while  $W[cW] = \emptyset$  do begin
    c := c + 1; instr[c] :=  $\emptyset$ ; cW := mod(cW+1, MaxC);
  end
  nextc := mod(c+1, MaxC);
  while  $W[cW] \neq \emptyset$  do begin
    Priority  $\rightarrow$  select and remove an arbitrary instruction x from  $W[cW]$ ;
    if  $\exists$  free issue units of type(x) on cycle c then begin
      instr[c] := instr[c]  $\cup$  {x}; Wcount := Wcount - 1;
      for each  $y \in \text{successors}[x]$  do begin
        count[y] := count[y] - 1;
        earliest[y] := max(earliest[y], c+delay(x));
        if count[y] = 0 then begin
          loc := mod(earliest[y], MaxC);
           $W[loc] := W[loc] \cup \{y\}$ ; Wcount := Wcount + 1;
        end
      end
    end
    else  $W[\text{nextc}] := W[\text{nextc}] \cup \{x\}$ ;
  end
end
end
```

CMSC430 Spring 2007

38

Instruction Scheduling ***(revisited)***

In an earlier lecture, we introduced list scheduling

- Efficient, greedy, local heuristic
- Technique of choice for more than 20 years

How can the compiler improve on local list scheduling?

- Different priority functions & tie breakers
- Use forward & backward list scheduling (Figures 12.4 & 12.5)
- Increase size of region fed to scheduler (classic answer)
- Try other algorithms

Little success with other algorithms on code written by humans

- Some compiler generated code defeats the list scheduler

What About OOO Execution?

Out-of-order microprocessors should simplify scheduling

- Processor looks at a window in the instruction stream
- Processor executes operations as they are ready
- Processor (typically) renames registers for correctness

Does this eliminate the need for instruction scheduling?

- For any finite window, \exists a worst case schedule
 - > 100 operation window \Rightarrow 110 for FU_1 , 110 for FU_2
- Schedule need not be perfect, but must be not bad
- OOO can compensate for mild variations in latency
 - > Cache miss, infrequent stall, ...

Depth-first
schedule !

Other Priority Functions

Computing Ranks

- Maximum path length containing it Decreases register use
 - > Favors critical path; tends towards depth-first
- Number of immediate successors in P Better with > 1 FU
 - > Favors longer ready queue; tends toward breadth-first
- Total number of descendants in P
 - > Favors heavily used values; tends toward breadth-first
- Add latency to node's rank
 - > Favors long operations to cover their latencies
- Increment rank of if node contains a last use
 - > Tends to shorten live ranges & decrease register pressure

These can be used as priorities or as tie-breakers

- > Use randomization & repetition

Forward versus Backward List Scheduling

Folk wisdom has long suggested doing both

- Some blocks amenable to forward scheduling
- Some blocks amenable to backward scheduling
- Conventional approach is to try both & keep best result

Does it matter?

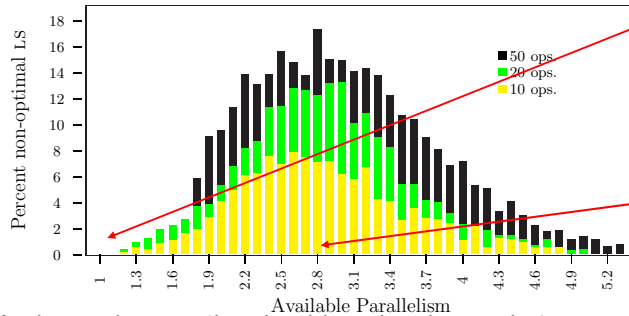
- Takes dependence graph that is both wide & deep
- Depends on detailed knowledge of the specific block
- See Figures 12.4 & 12.5 in EAC for a real example

Advice

- Use several forward & several backward passes
- Use different priorities & tie breakers, or use randomization

How Well Does List Scheduling Do?

Non-optimal list schedules (%) versus available parallelism
1 functional unit, randomly generated blocks of 10, 20, 50 ops



Most codes fall here, unless the compiler transforms them for ILP.

If the compiler transforms the code, it should avoid this area!

At the peak, compiler should apply other techniques

- Measure parallelism in list scheduler
- Invoke stronger techniques when high-probability of payoff

Scheduling over Larger Scopes

Basic idea is simple

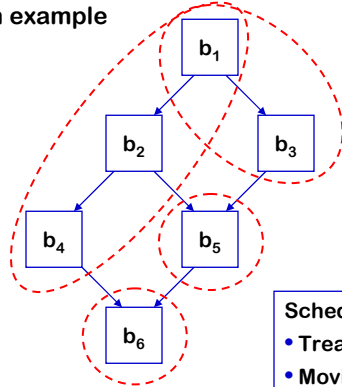
- Longer sequence of operations \Rightarrow more opportunities
- Pick a multi-block path & treat it as a single block
- Add compensation code for other exits (& entries)

Several distinct scopes

- Extended basic blocks
 - > A sequence b_1, b_2, \dots, b_n where b_i has 1 predecessor, $1 < i \leq n$
- Traces
 - > An arbitrary acyclic path, usually chosen from trace data
- Loops
 - > Think of source-language loop, can find arbitrary loops

Extended Basic Blocks

An example



Extended Basic Blocks

- b₁, b₂, b₄
- b₁, b₃
- b₅
- b₆

Both b₅ and b₆ have > 1 predecessors

Scheduling EBBs

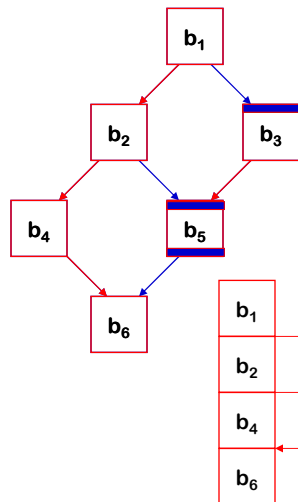
- Treat EBB as a block
- Moving an operation across boundary can necessitate compensation code
- Can restrict motion to zero growth case with 12 to 13% improvement [LCTES 98]

CMSC430 Spring 2007

45

Traces

Technique developed for the Multiflow Computer



- 1 Identify high-frequency path
- 2 Schedule it as if a single block
- 3 Insert compensation code
- 4 Schedule next important path

Results

- Fast trip through common path
- Off-path quality declines
- Compensation code ⇒ growth

CMSC430 Spring 2007

46

Loop Scheduling (Software Pipelining)

Loops deserve special attention

- Their bodies execute frequently
- They do most of the work in time-critical computations
- They often contain major holes & interlocks

The ideas

- Schedule multiple iterations together
- Run several iterations concurrently
- Shorten “**initiation interval**” for overall speed
 - > Cycles between initiation of different iterations
 - > = length of computation kernel

Example

```
loadI      r0,0 => r1
loadI      r0,400 => r2
floadAI    r0,c => fr1
10 floadAI  r1,a => fr2
11 fadd     fr2,fr1 => fr2
12 fstoreAI fr2 => r1,b
13 addI     r1,8 => r1
14 cmp_LE  r1,r2 => r3
15 cbr     r1 => 10,16
```

← 2 cycle delay

← 3 cycle delay

How fast (in cycles per iteration) can we execute this loop?

Minimum Number of Cycles in Kernel

- **Machine resource constraint:**
 - > N_u is the number of units of type u
 - > I_u is the number of instructions requiring a unit of type u
 - > $\lceil I_u / N_u \rceil$ is the minimum number of cycles required for one iteration of the loop based on unit u
 - > $\max_u \lceil I_u / N_u \rceil$ is the minimum number of cycles required for all units
- **Slope constraint**
 - > If the loop computes a recurrence over k_r cycles
 - > And the total delay along the recurrence cycle is d_r
 - > Then each iteration is going to require d_r / k_r cycles to execute
 - > $\max_r \lceil d_r / k_r \rceil$ is the minimum number of cycles per iteration to compute recurrences

Example

	loadI	r0,0 ⇒ r1	
	loadI	r0,400 ⇒ r2	
	floadAI	r0,c ⇒ fr1	
10	floadAI	r1,a ⇒ fr2	← 2 cycle delay
11	fadd	fr2,fr1 ⇒ fr2	
12	fstoreAI	fr2 ⇒ r1,b	
13	addI	r1,8 ⇒ r1	← 3 cycle delay
14	cmp_LE	r1,r2 ⇒ r3	
15	cbr	r3 ⇒ 10,16	

Floating Pt Unit:	1 instruction	
Load/Store Unit:	2 instructions	⇒ 3 cycles minimum
Integer Unit:	3 instructions	

Loop Scheduling

Mechanics

- Determine lower bound on initiation interval
 - > Number of issue slots
 - > Longest dependence chain Register constraint, too
- Lay out a schedule of appropriate length
- Use list scheduling with a modulo cycle count
 - > **Could fail** — just try with one more cycle per iteration
- Add a pre-loop & a post-loop to “fill” & “drain” the pipeline

It gets pretty intricate !

- Conditional control flow complicates matters even more

Example

```

loadI      r0,0 => r1
loadI      r0,400 => r2
floadAI    r0,c => fr1
10 floadAI  r1,a => fr2
11 fadd     fr2,fr1 => fr2
12 fstoreAI fr2 => r1,b
13 addI     r1,8 => r1
14 cmp_LE  r1,r2 => r3
15 cbr     r1 => 10,16
  
```

← 2 cycle delay

← 3 cycle delay

Load/Store Unit	Integer Unit	Floating Point Unit
floadAI r1,a => fr2	addI r1,8 => r1	
	cmp_LE r1,r2 => r3	
fstoreAI fr3-> r1,b-16	cbr r3 => 10,16	fadd fr2,fr1 => fr3

Final Code

```
ld r1,0
ld r2,400
fld fr1, c
p1 fld fr2,a(r1);   ai    r1,r1,8
p2                               comp r1,r2
p3                               beq  e1;      fadd  fr3,fr2,fr1
k1 fld fr2,a(r1);   ai    r1,r1,8
k2                               comp r1,r2
k3 fst fr3,b-16(r1); blek1;      fadd  fr3,fr2,fr1
e1 nop
e2 nop
e3 fst fr3,b-8(r1)
```