

Register Allocation III

Last time

- Register allocation across function calls

Today

- Register allocation options

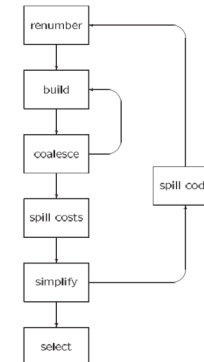
CS553 Lecture

Register Allocation III

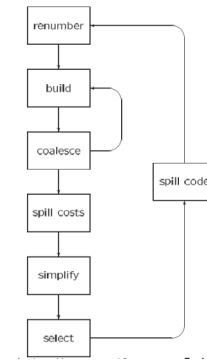
1

Interference Graph Allocators

Chaitin



Briggs



CS553 Lecture

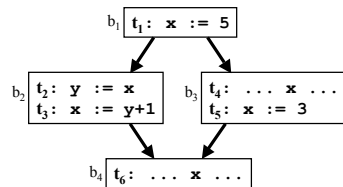
Register Allocation III

2

Granularity of Allocation (Renumber step in Briggs)

What is allocated to registers?

- Variables/Temporaries
- Live ranges/Webs (*i.e.*, du-chains with common uses)
- Values (*i.e.*, definitions; same as variables with SSA)



Variables: 2 (x & y)
 Live Ranges/Web: 3 ($t_1 \rightarrow t_2, t_4$;
 $t_2 \rightarrow t_3$;
 $t_3, t_5 \rightarrow t_6$)
 Values: 4 ($t_1, t_2, t_3, t_5, \phi(t_3, t_5)$)

What are the tradeoffs?

Each allocation unit is given a symbolic register name (*e.g.*, s_1, s_2 , etc.)

CS553 Lecture

Register Allocation III

3

Coalescing

Move instructions

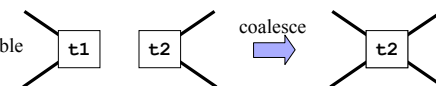
- Code generation can produce unnecessary move instructions
`mov t1, t2`
- If we can assign t_1 and t_2 to the same register, we can eliminate the move

Idea

- If t_1 and t_2 are not connected in the interference graph, **coalesce** them into a single variable

Problem

- Coalescing can increase the number of edges and make a graph uncolorable
- Limit coalescing to avoid uncolorable graphs



CS553 Lecture

Register Allocation III

4

Coalescing Logistics

Rule

- When building the interference graph, do NOT make virtual registers interfere due to copies.
- If the virtual registers s1 and s2 do not interfere and there is a copy statement $s1 = s2$ then s1 and s2 can be coalesced.
- Example

```
a = t + u
...
b = a
c = a
...
x = b + w
z = c + y
```

Before Coalescing

```
ab = t + u
...
c = ab
...
x = ab + w
z = c + y
```

After Coalescing

Computing the Interference Graph (in MiniJava compiler)

Use results of live variable analysis

Computing interference graph to enable coalescing

```
for each flow graph node n do
  for each def in def(n) do
    for each temp in liveout(n) do
      if (not stmt(n) isa MOVE or use != temp) then
        E ← E ∪ (def, temp)
```

Coalescing in MiniJava compiler

Currently the InterferenceGraph only has one Temp.Temp associated with each node

- represent each merged node with just one of the temps
- keep a separate map of representatives mapped to sets of temps
- also keep a map of temps mapped to their representative
- when rewriting the code use the representative instead of the original temp

Register Allocation: Spilling

If we can't find a k-coloring of the interference graph

- Spill variables (nodes) until the graph is colorable

Choosing variables to spill

- Choose least frequently accessed variables
- Break ties by choosing nodes with the most conflicts in the interference graph
- Yes, these are heuristics!

Weighted Interference Graph

Goal

- $\text{Weight}(s) = \sum_{\forall \text{ references } r \text{ of } s} f(r)$ is execution frequency of r

Static approximation

- Use some reasonable scheme to rank variables
- Some possibilities
 - $\text{Weight}(s)$ = num of times s is used in program
 - $\text{Weight}(s) = 10 \times (\# \text{ uses in loops}) + (\# \text{ uses in straightline code})$
 - $\text{Weight}(s) = 20 \times (\# \text{ uses in loops}) + 2 \times (\# \text{ uses in straightline code}) + (\# \text{ uses in a branch statement})$

Improvement #1: Simplification Phase [Chaitin 81]

Idea

- Nodes with $< k$ neighbors are guaranteed colorable
- Improvement over simple greedy coloring algorithm

Remove them from the graph first

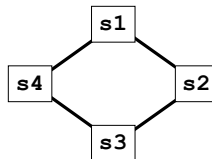
- Reduces the degree of the remaining nodes

Must spill only when all remaining nodes have degree $\geq k$

Referred to as pessimistic spilling

The Problem: Worst Case Assumptions

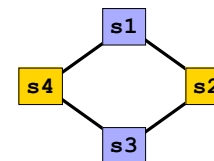
Is the following graph 2-colorable?



Clearly 2-colorable

- But Chaitin's algorithm leads to an immediate block and spill
- The algorithm assumes the worst case, namely, that all neighbors will be assigned a different color

Improvement #2: Optimistic Spilling [Briggs 89]



Idea

- Some neighbors might get the same color
 - Nodes with k neighbors **might** be colorable
 - Blocking does not imply that spilling is necessary
 - Push blocked nodes on stack (rather than place in spill set)
 - Check colorability upon popping the stack, when more information is available
- } Defer decision

Algorithm [Briggs et al. 89]

```

while interference graph not empty do
  while  $\exists$  a node  $n$  with  $< k$  neighbors do
    Remove  $n$  from the graph
    Push  $n$  on a stack
  } simplify
  if any nodes remain in the graph then { blocked with  $\geq k$  edges }
    Pick a node  $n$  to spill { lowest spill-cost/highest degree }
    Push  $n$  on stack
    Remove  $n$  from the graph
  } defer decision
  while stack not empty do
    Pop node  $n$  from stack
  }
  if  $n$  is colorable then { make decision }
    Allocate  $n$  to a register
  else
    Insert spill code for  $n$  { Store after def; load before use }
    Reconstruct interference graph & start over
  
```

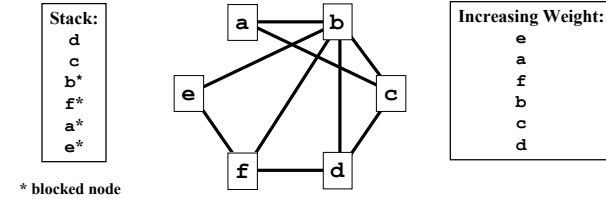
CS553 Lecture

Register Allocation III

13

Example

Attempt to 2-color this graph (,)



CS553 Lecture

Register Allocation III

14

Possible Register Allocation Design

Overall algorithm: graph coloring with simplification

Interference graph: two temps interfere if

- one is defined in a stmt and the other is live out of the same stmt
- exception is a MOVE statement where the temps are the source and dest

Coalesce: Briggs strategy

- coalesce if new node will have fewer than K neighbors of significant degree ($\geq K$) and both nodes are involved in a move

Spill heuristic:

- spill the node with the lowest weight and break ties by spilling the node with the most total adjacent edges

Simplification:

- optimistic, push by increasing weight and feasibility, select blocked nodes using spill heuristic

Select:

- attempt selection on everything in the stack before generating spill code

CS553 Lecture

Register Allocation III

15

Example

```

// missing stack setup
// callee-save assigns to temps
t2 = $s0
  
```

```

// procedure body
t3 = 1
Loop:
  if t3 > 20 goto End
  t3 = t3 + 1
  jal bar // defs: $v0
  goto Loop
End:
t4 = t3 + $v0
t5 = t4 + t3
$v0 = t5
  
```

```

// callee-save reads from temps
$s0 = t2

// sink stmt, uses callee-saves
// and caller-saves
// missing stack cleanup and ret
  
```

After Spilling t2

```

// missing stack setup
// callee-save assigns to temps
t2 = $s0
M[$fp-12] = t2
  
```

```

// procedure body
t3 = 1
Loop:
  if t3 > 20 goto End
  t3 = t3 + 1
  jal bar // defs: $v0
  goto Loop
End:
t4 = t3 + $v0
t5 = t4 + t3
$v0 = t5
  
```

```

// callee-save reads from temps
t2 = M[$fp-12]
$s0 = t2

// sink stmt, uses callee-saves
// and caller-saves
// missing stack cleanup and ret
  
```

CS553 Lecture

Register Allocation III

16

Example After coalesce

```
// missing stack setup
// callee-save assigns to temps

M[$fp-12] = $s0

// procedure body
t3 = 1
Loop:
if t3 > 20 goto End
t3 = t3 + 1
jal bar // defs: $v0
goto Loop
End:
t4 = t3 + $v0
t5 = t4 + t3
$v0 = t5

// callee-save reads from temps
$s0 = M[$fp-12]

// sink stmt, uses callee-saves
// and caller-saves
// missing stack cleanup and ret
```

CS553 Lecture

Register Allocation III

17

After allocation

```
// missing stack setup
// callee-save assigns to temps

M[$fp-12] = $s0

// procedure body
$s0 = 1
Loop:
if $s0 > 20 goto End
$s0 = $s0 + 1
jal bar // defs: $v0
goto Loop
End:
$v0 = $s0 + $v0
$s0 = $v0 + $s0
$v0 = $s0

// callee-save reads from temps
$s0 = M[$fp-12]

// sink stmt, uses callee-saves
// and caller-saves
// missing stack cleanup and ret
```

Improvement #3: Live Range Splitting [Chow & Hennessy 84]

Idea

- Start with variables as our allocation unit
- When a variable can't be allocated, split it into multiple subranges for separate allocation
- Selective spilling: put some subranges in registers, some in memory
- Insert memory operations at boundaries

Why is this a good idea?

CS553 Lecture

Register Allocation III

18

Improvement #4: Rematerialization [Chaitin 82]&[Briggs 84]

Idea

- Selectively re-compute values rather than loading from memory
- "Reverse CSE"

Easy case

- Value can be computed in single instruction, and
- All operands are available

Examples

- Constants
- Addresses of global variables
- Addresses of local variables (on stack)

CS553 Lecture

Register Allocation III

19

Next Time

Lecture

- Instruction scheduling

CS553 Lecture

Register Allocation III

20