

CMSC 430
Introduction to Compilers
Fall 2016

Register Allocation

Introduction

- Change code that uses an unbounded set of *virtual* registers to code that uses a finite set of *actual* regs
 - For bytecode targets, can let the JIT handle this
 - Even with finite set of bytecode regs—that finite set is probably large
 - But critical for compiling to real hardware
- Critical properties
 - Produce correct code
 - Minimize added *spill code*
 - The code needed to move values between registers and memory, that wasn't needed when assuming unbounded set of registers
 - Memory operations are slow on modern processors
 - Minimize space for spilled registers
 - Operate efficiently
 - E.g., not exponential in size of code

Register allocation approaches

- Local allocation (within basic blocks)
 - In single forward pass through block, spill and load regs as necessary
 - (Could also try to look at block as a whole to determine some better allocation)
- Global allocation (across basic blocks)
 - Use graph coloring
- Local allocation is simple to implement
 - But can introduce inefficiencies at block boundaries
 - Most compilers use graph-coloring based global allocation

Spill code

- Where should spilled registers be stored?
 - Each instance of a function needs its own storage
 - ⇒ store on stack
 - Can allocate space for spilled regs in function prolog code
 - Refer to reg storage using frame pointer
 - Need to reserve *feasible* set of physical regs only for spilling
- Inserted spill code
 - Definition of a spilled register *rs*
 - `add rs, r2, r3` — insert “`store n(%ebp), rs`” afterward
 - Use of spilled register *rs*
 - `add rs, r2, r3` — insert “`load rs, n(%ebp)`” before

Instruction set

- For illustration purposes, we'll use the instruction set from codegen-*.ml
 - Will write rn for register n

```
type instr =
  | ILoad of reg * src          (* dst, src *)
  | IStore of dst * reg         (* dst, src *)
  | IMov of reg * reg          (* dst, src *)
  | IAdd of reg * reg * reg     (* dst, src1, src2 *)
  | IMul of reg * reg * reg     (* dst, src1, src2 *)
  | IJump of int               (* pc offset *)
  | IIfZero of reg * int       (* src, pc offset *)
  | IReturn
```

Live ranges

- A register is *live*
 - Starting at its definition ($x \leftarrow \dots$), inclusive
 - Ending at the point it becomes dead ($y \leftarrow \dots x \dots$), inclusive
 - Can represent as an interval $[i,j]$ or *live range* within a block
 - Also need to know which regs live on exit

<u>Source code</u>	<u>Live regs (end of instr)</u>
<code>ILoad r1, 42</code>	<code>r1</code>
<code>IMov r2, r1</code>	<code>r1 r2</code>
<code>IMul r3, r1, r2</code>	<code>r1 r2 r3</code>
<code>ILoad r4, 5</code>	<code>r1 r2 r3 r4</code>
<code>IAdd r5, r4, r2</code>	<code>r1 r3 r5</code>
<code>ILoad r6, 8</code>	<code>r1 r3 r5 r6</code>
<code>IMul r7, r5, r6</code>	<code>r1 r3 r7</code>
<code>IAdd r8, r7, r3</code>	<code>r1 r8</code>
<code>IAdd r1, r8, r1</code>	<code>r1</code>
<code>IStore &1234, r1</code>	<code>(none)</code>

Local register allocation

- Algorithm
 - Start with empty reg set
 - Load from memory into reg on demand
 - When no reg available, spill to free one
 - Need policy on which reg to spill
 - Common approach: one whose next use is farthest in the future
 - Keep values “used soon” in registers
 - Similar to cache line / page replacement

Example

- One possible bottom-up allocation to 3 regs (ra-rc)
 - Notice r1 spilled to memory after first IMul

<u>Source code</u>	<u>Live regs</u>	<u>Reg alloc (at exit)</u>		
		<u>ra</u>	<u>rb</u>	<u>rc</u>
ILoad r1, 42	r1	r1		
IMov r2, r1	r1 r2	r1	r2	
IMul r3, r1, r2	r1 r2 r3	r1	r2	r3
(spill r1 to memory)				
ILoad r4, 5	r1 r2 r3 r4	r4	r2	r3
IAdd r5, r4, r2	r1 r3 r5	r4	r2→r5	r3
ILoad r6, 8	r1 r3 r5 r6	r6	r5	r3
IMul r7, r5, r6	r1 r3 r7	r6	r5→r7	r3
IAdd r8, r7, r3	r1 r8	r6	r8	r3
(load r1 from memory)				
IAdd r1, r8, r1	r1	r1	r8	r3
IStore &1234, r1	(none)			

Example generated code

- One possible bottom-up allocation to 3 regs (ra-rc)
 - Notice r1 spilled to memory after first IMul

<u>Source code</u>	<u>Live regs</u>	<u>Reg alloc (at exit)</u>		
		<u>ra</u>	<u>rb</u>	<u>rc</u>
ILoad ra, 42	r1	r1		
IMov rb, ra	r1 r2	r1	r2	
IMul rc, ra, rb	r1 r2 r3	r1	r2	r3
(spill ra to memory for r1)				
ILoad ra, 5	r1 r2 r3 r4	r4	r2	r3
IAdd rb, ra, rb	r1 r3 r5	r4	r2→r5	r3
ILoad ra, 8	r1 r3 r5 r6	r6	r5	r3
IMul rb, rb, ra	r1 r3 r7	r6	r5→r7	r3
IAdd rb, rb, rc	r1 r8	r6	r8	r3
(load ra from memory for r1)				
IAdd ra, rb, ra	r1	r1	r8	r3
IStore &1234, ra	(none)			

Register reuse

- Note that in some cases, can reuse the same register as source and target in single instruction
 - Namely, when one live range ends and another begins

<u>Source code</u>	<u>Live regs</u>
<code>ILoad r1, 42</code>	<code>r1</code>
<code>ILoad r2, 43</code>	<code>r1 r2</code>
<code>IMul r3, r1, r2</code>	<code>(none)</code>

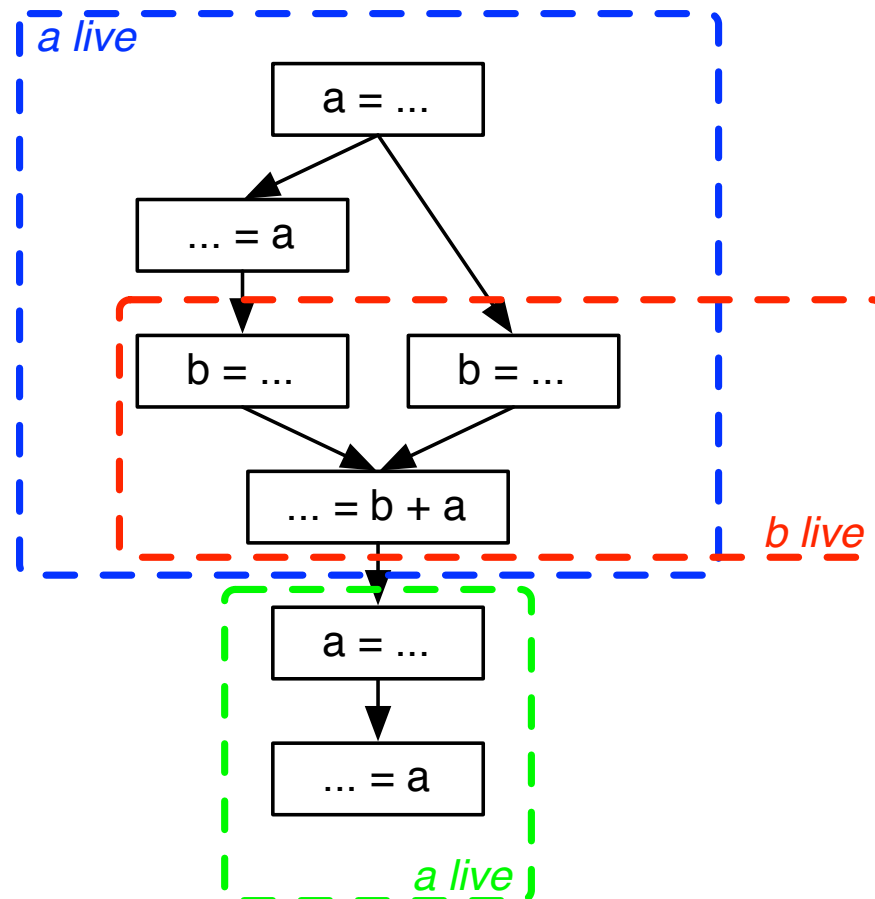
- Suppose `r1` \mapsto `ra` and `r2` \mapsto `rb`
- Then could assign `r3` to `ra`, `rb`, or some other register
- In previous slide, wrote register reuse as `r1` \rightarrow `r2`
 - `r1` is assigned at beginning of instruction, `r2` at end

Global register allocation [Chaitin et al 1981]

- Definition: Graph coloring problem
 - Input: A graph G and an integer k
 - k is the number of “colors”
 - Output: an assignment of nodes of G to colors such that
 - No nodes that are connected by an edge have the same color
 - The assignment uses at most k colors
 - This problem is NP-hard for $k > 2$
- Reduce register allocation to *graph coloring*
 - Data flow analysis to find live ranges of virtual registers
 - Build a *color interference graph*, where
 - Nodes represent live ranges
 - Edge between two nodes indicates both ranges live at some point
 - Find k coloring of graph, for $k = \#$ of physical regs
 - If unable to find coloring, spill virtual regs and repeat

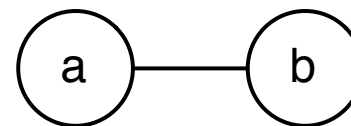
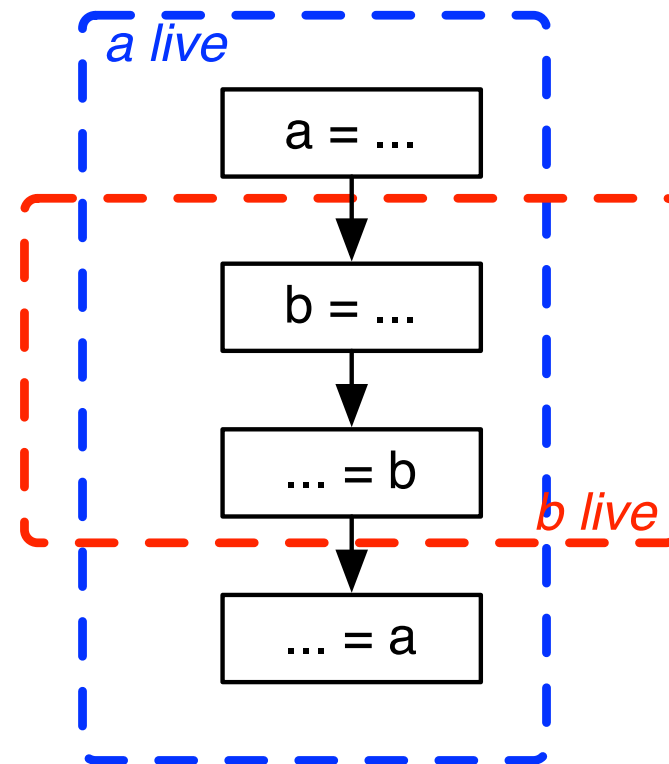
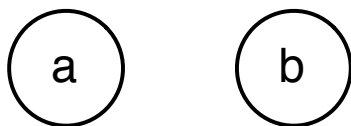
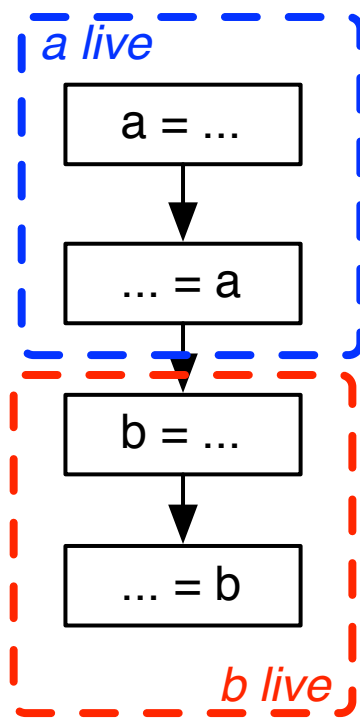
Live ranges

- All nodes in CFG from definition to use, inclusive
 - Live ranges indicate when virtual registers should be in some physical reg to avoid spill code
 - (A single virtual register may comprise several live ranges)



Building the interference graph

- At each point p in the program
 - Add edge (x,y) for all pairs of live ranges x, y live at p

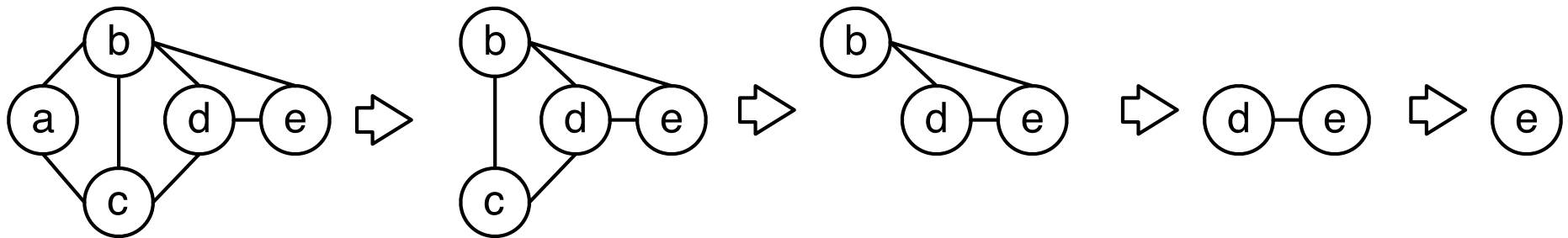


Graph coloring via simplification

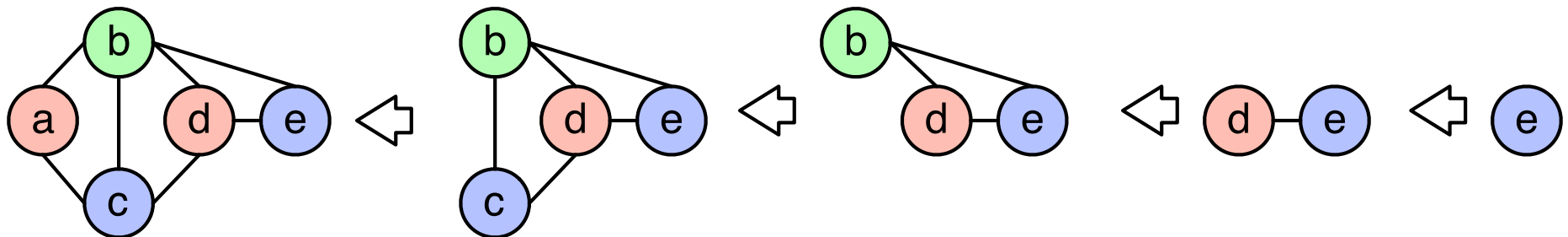
- Algorithm
 - Repeatedly remove nodes with degree $< k$ from graph
 - Push nodes onto stack, removing from graph
 - If every remaining node is degree $\geq k$
 - Spill node with lowest spill cost
 - Use some heuristic to guess which virtual reg best to spill
 - Remove node from graph
 - (Once spilled, no longer causes interference)
 - Reassemble graph with nodes popped from stack
 - Choose color differing from neighbors when added to graph
 - Always possible since node had degree $< k$

Graph coloring example

- Assume 3 physical registers
 - Simplify graph by removing nodes with < 3 neighbors

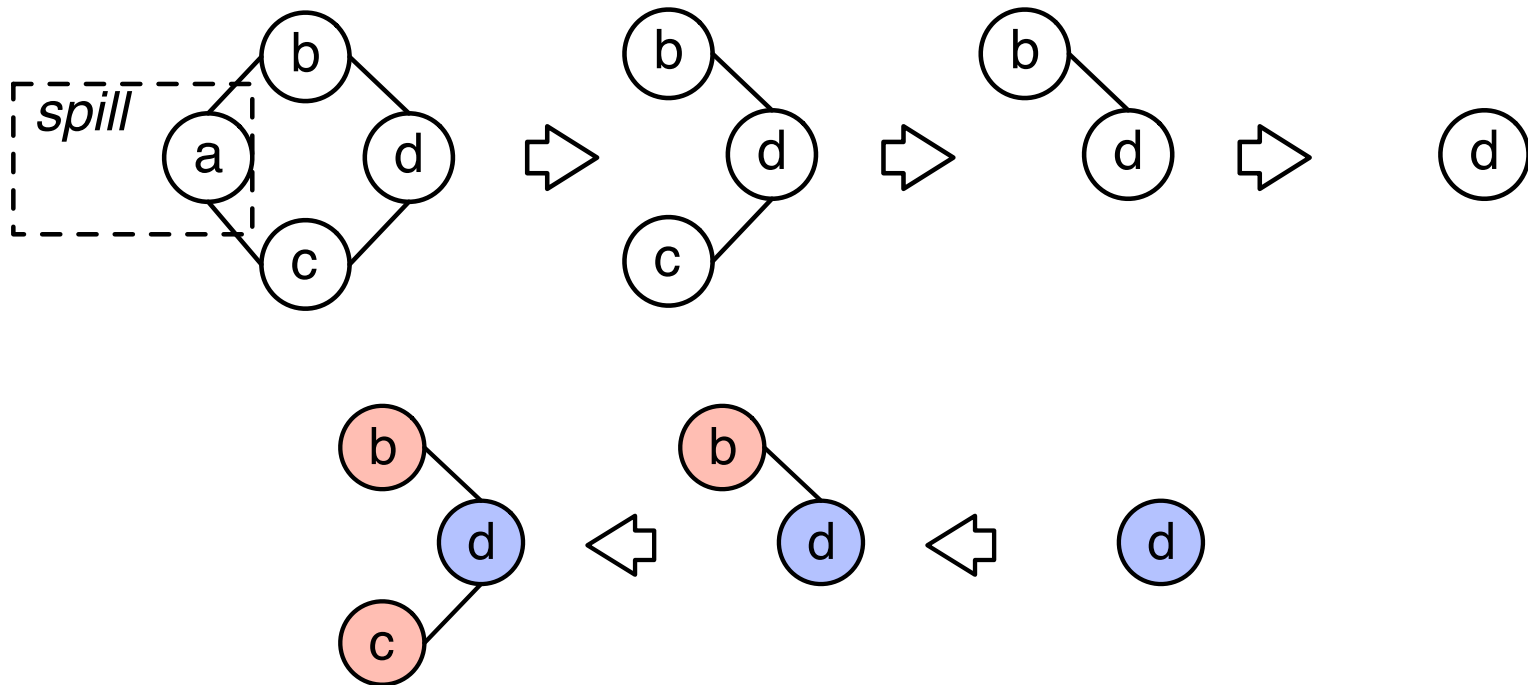


- Reassemble by popping nodes from stack
 - Assigning colors not used by neighbors



Graph coloring w/spill

- Assuming 2 physical registers
 - No node with < 2 neighbors
 - Must spill node with lowest spill cost
 - Remaining nodes can then be simplified and colored



Spill code

- Here, we've assumed that spilling a removes it completely from live range
 - But of course, the spill code will need to load and store to register
 - Thus, we either need to
 - Recompute live ranges after we insert spill code
 - Reserve a set of register that cannot be allocated to, but that we will use to load and store for spills

Discussion

- Global register allocation is an old idea
 - Material presented in these slides is just the beginning—there's been lots of work coming up with better variants
- *Register pressure* occurs when not enough physical registers available, requiring spills
- Register allocation and optimization interact
 - If we optimize before register alloc, might increase register pressure
 - E.g., by moving a computation earlier than it was before, thereby increasing live ranges
 - If we register alloc before optimizing, might create false dependencies
 - E.g., reg alloc maps what are conceptually separate variables to the same physical register; could confuse optimizer