COS320: Compiling Techniques

Zak Kincaid

April 9, 2024

Register allocation

Motivation

- Your LLVMlite compiler places each uid in its own stack slot
- Every binary operation is compiled to 2 loads, the operation, and a store
  - Loads and stores are expensive
- *Register allocation* is the problem of determining a mapping from IR-level "virtual registers" to machine registers

Live variables

- A variable x is live at a point n if there is some path starting from n that *reads* the value of x before *writing* it.
  - Intuition: a variable is live if its value might be needed later in some computation.
- If a variable x is *not* live, we can free/re-use the memory associated with x
- If two variables are not live at the same time, we can store them in the same memory (ideally, a register)

Live variables

- Live variables is a *backwards* dataflow analysis problem
  - Information flows from control flow *successors* to their *predecessors*

Forwards: Compute *least* **IN**, **OUT** s.t.

1. $\textbf{IN}[s] = \top$
2. For all $n \in N$, *post*$_{\mathcal{L}}(n, \textbf{IN}[n]) \sqsubseteq \textbf{OUT}[n]$
3. For all $p \to n \in E$, $\textbf{OUT}[p] \sqsubseteq \textbf{IN}[n]$

Backwards: Compute *least* **IN**, **OUT** s.t.

1. $\textbf{OUT}[n] = \top$ for each *return* block n
2. For all $n \in N$, *pre*$_{\mathcal{L}}(n, \textbf{OUT}[n]) \sqsubseteq \textbf{IN}[n]$
3. For all $n \to s \in E$, $\textbf{IN}[s] \sqsubseteq \textbf{OUT}[n]$

  - Backwards analyses work in essentially the same was as forwards analyses

Live variables

- Live variables is a *backwards* dataflow analysis problem
  - Information flows from control flow *successors* to their *predecessors*

Forwards: Compute *least* **IN**, **OUT** s.t.

1. $\mathbf{IN}[s] = \top$
2. For all $n \in N$, ***post***$_{\mathcal{L}}(n, \mathbf{IN}[n]) \sqsubseteq \mathbf{OUT}[n]$
3. For all $p \to n \in E$, $\mathbf{OUT}[p] \sqsubseteq \mathbf{IN}[n]$

Backwards: Compute *least* **IN**, **OUT** s.t.

1. $\mathbf{OUT}[n] = \top$ for each *return* block n
2. For all $n \in N$, ***pre***$_{\mathcal{L}}(n, \mathbf{OUT}[n]) \sqsubseteq \mathbf{IN}[n]$
3. For all $n \to s \in E$, $\mathbf{IN}[s] \sqsubseteq \mathbf{OUT}[n]$

  - Backwards analyses work in essentially the same was as forwards analyses
- Live variables as a (gen/kill) data flow analysis:

Live variables

- Live variables is a *backwards* dataflow analysis problem
  - Information flows from control flow *successors* to their *predecessors*

Forwards: Compute *least* **IN**, **OUT** s.t.

1. **IN**$[s] = \top$
2. For all $n \in N$, **post**$_{\mathcal{L}}(n,$ **IN**$[n]) \sqsubseteq$ **OUT**$[n]$
3. For all $p \to n \in E$, **OUT**$[p] \sqsubseteq$ **IN**$[n]$

Backwards: Compute *least* **IN**, **OUT** s.t.

1. **OUT**$[n] = \top$ for each *return* block n
2. For all $n \in N$, **pre**$_{\mathcal{L}}(n,$ **OUT**$[n]) \sqsubseteq$ **IN**$[n]$
3. For all $n \to s \in E$, **IN**$[s] \sqsubseteq$ **OUT**$[n]$

  - Backwards analyses work in essentially the same was as forwards analyses
- Live variables as a (gen/kill) data flow analysis:
  - Abstract domain: 2^{Var}
    - *Existential* $\Rightarrow$ order is $\subseteq$, join is $\cup$, $\top$ is *Var*, $\bot$ is $\emptyset$
  - *pre*$(elt, L) = (L \setminus$ *kill*$(elt)) \cup$ *gen*(elt)

Live variables

- Live variables is a *backwards* dataflow analysis problem
  - Information flows from control flow *successors* to their *predecessors*

Forwards: Compute *least* **IN**, **OUT** s.t.

1. **IN**$[s] = \top$
2. For all $n \in N$, *post*$_{\mathcal{L}}(n, \textbf{IN}[n]) \sqsubseteq \textbf{OUT}[n]$
3. For all $p \to n \in E$, **OUT**$[p] \sqsubseteq$ **IN**$[n]$

Backwards: Compute *least* **IN**, **OUT** s.t.

1. **OUT**$[n] = \top$ for each *return* block n
2. For all $n \in N$, *pre*$_{\mathcal{L}}(n, \textbf{OUT}[n]) \sqsubseteq \textbf{IN}[n]$
3. For all $n \to s \in E$, **IN**$[s] \sqsubseteq$ **OUT**$[n]$

  - Backwards analyses work in essentially the same was as forwards analyses
- Live variables as a (gen/kill) data flow analysis:
  - Abstract domain: 2^{Var}
    - *Existential* $\Rightarrow$ order is $\subseteq$, join is $\cup$, $\top$ is *Var*, $\bot$ is $\emptyset$
  - *pre*$(elt, L) = (L \setminus \textit{kill}(elt)) \cup \textit{gen}(elt)$
  - *kill*$(x := e) = \{x\}$, *kill*(`cbr x, l1, l2`) $= \emptyset$

Live variables

- Live variables is a *backwards* dataflow analysis problem
  - Information flows from control flow *successors* to their *predecessors*

Forwards: Compute *least* **IN**, **OUT** s.t.

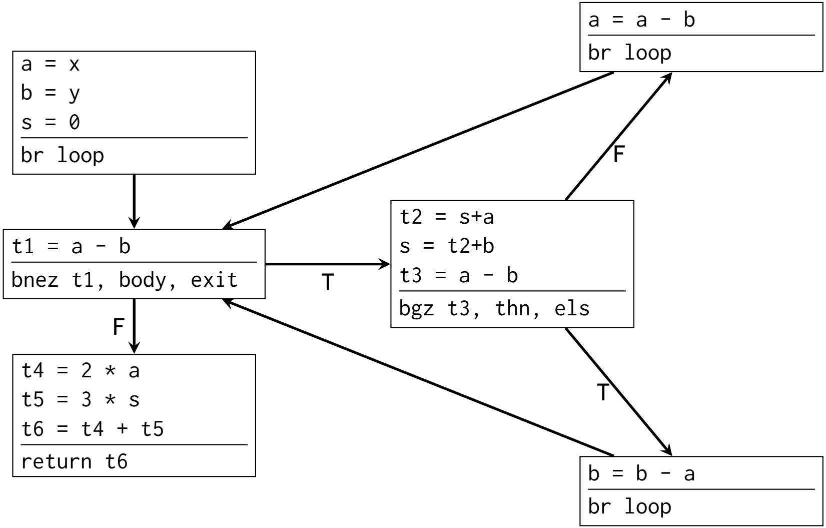
1. **IN**$[s] = \top$
2. For all $n \in N$, *post*$_{\mathcal{L}}(n,$ **IN**$[n]) \sqsubseteq$ **OUT**$[n]$
3. For all $p \to n \in E$, **OUT**$[p] \sqsubseteq$ **IN**$[n]$

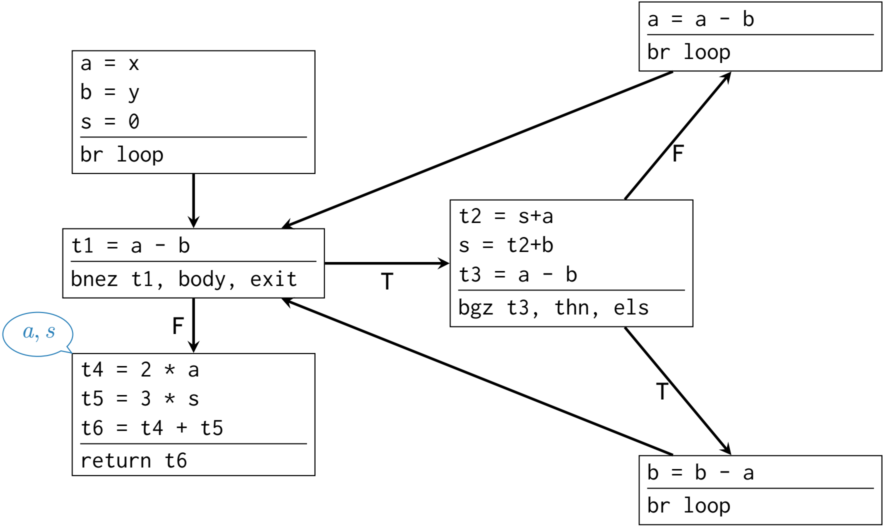
Backwards: Compute *least* **IN**, **OUT** s.t.

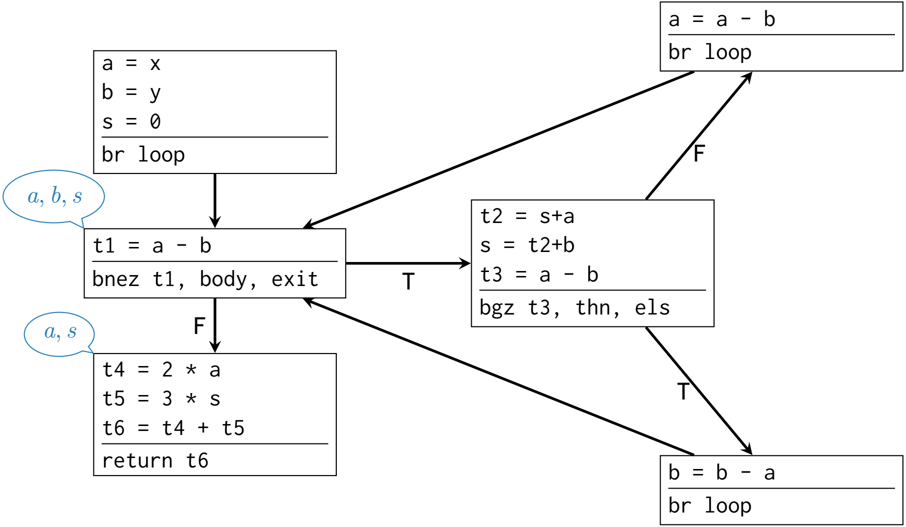
1. **OUT**$[n] = \top$ for each *return* block n
2. For all $n \in N$, *pre*$_{\mathcal{L}}(n,$ **OUT**$[n]) \sqsubseteq$ **IN**$[n]$
3. For all $n \to s \in E$, **IN**$[s] \sqsubseteq$ **OUT**$[n]$

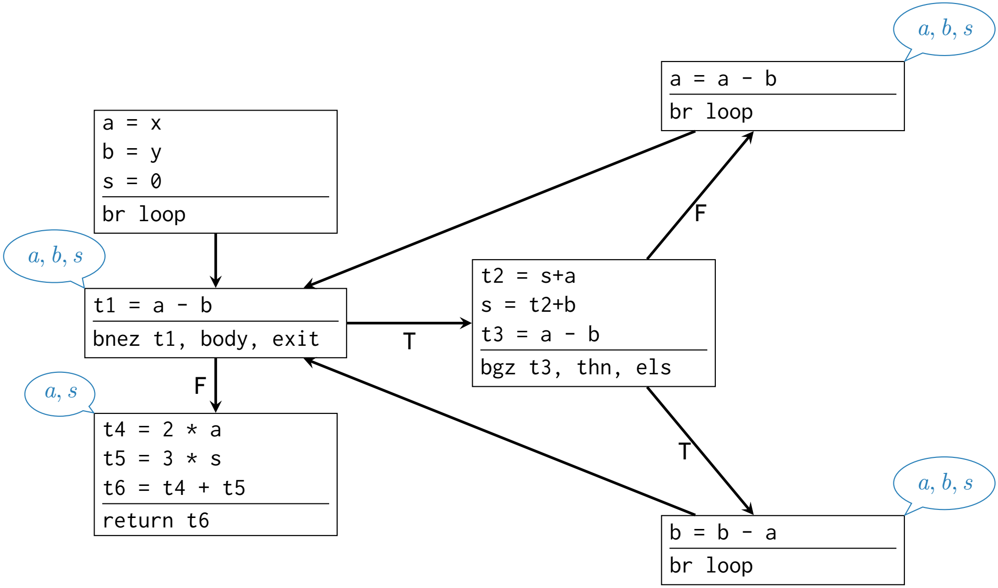
  - Backwards analyses work in essentially the same was as forwards analyses
- Live variables as a (gen/kill) data flow analysis:
  - Abstract domain: 2^{Var}
    - *Existential* $\Rightarrow$ order is $\subseteq$, join is $\cup$, $\top$ is *Var*, $\bot$ is $\emptyset$
  - *pre*$(elt, L) = (L \setminus$ *kill*$(elt)) \cup$ *gen*(elt)
  - *kill*$(x := e) = \{x\}$, *kill*(`cbr x, l1, l2`) $= \emptyset$
  - *gen*$(x := e) = \{y : y$ *in* $e\}$, *gen*(`cbr x, l1, l2`) $= \{x\}$

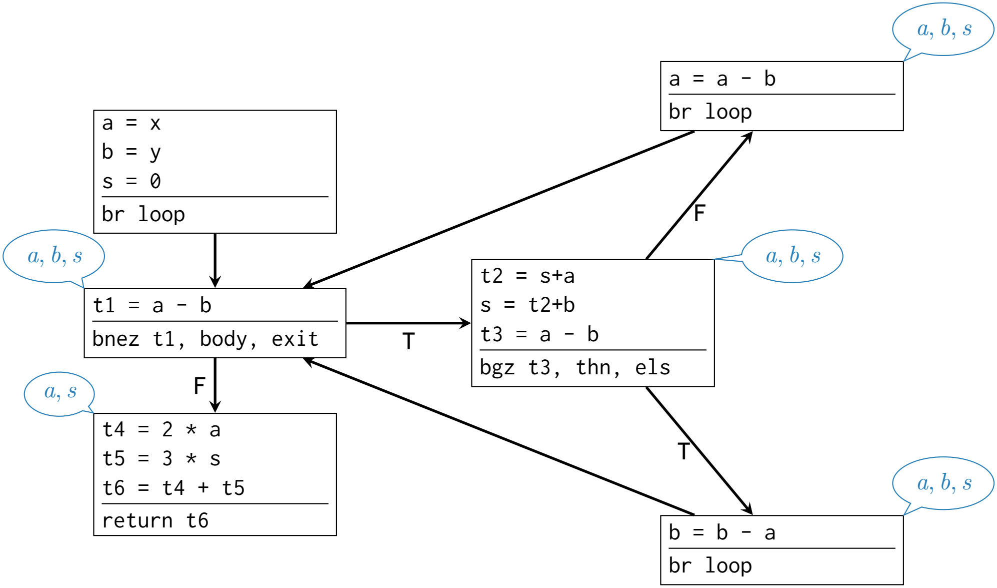
```
foo(int x, int y) {
  a := x;
  b := y;
  s := 0;
  while (a != b) {
    s := s + a + b;
    if (a > b) {
      a := a - b;
    } else {
      b := b - a;
    }
  }
  return 2 * a + 3 * s;
}
```

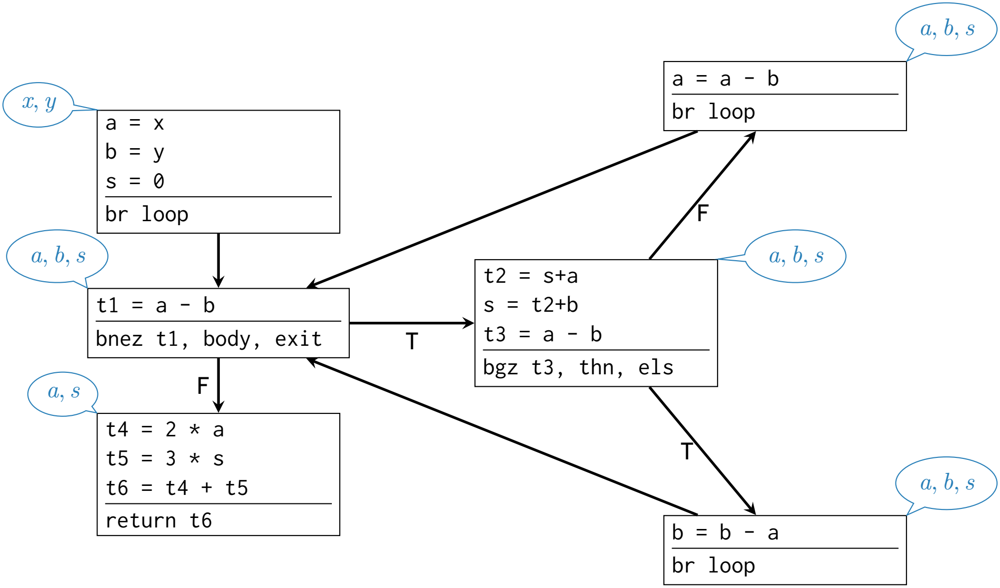

a = x
b = y
s = 0
br loop
t1 = a - b
bnez t1, body, exit
T
F
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
T
a = a - b
br loop
b = b - a
br loop


a = x
b = y
s = 0
br loop
t1 = a - b
bnez t1, body, exit
T
F
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
T
a = a - b
br loop
b = b - a
br loop


a = x
b = y
s = 0
br loop
a, b, s
t1 = a - b
bnez t1, body, exit
T
F
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
T
a = a - b
br loop
b = b - a
br loop


a, b, s
a = a - b
br loop
a = x
b = y
s = 0
br loop
F
a, b, s
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
t1 = a - b
bnez t1, body, exit
T
F
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
T
a, b, s
b = b - a
br loop

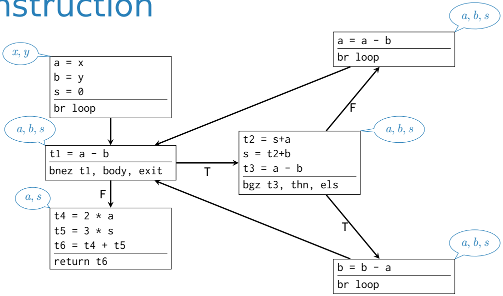

a, b, s
a = a - b
br loop
a = x
b = y
s = 0
br loop
F
a, b, s
a, b, s
t1 = a - b
bnez t1, body, exit
T
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
a, s
T
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
a, b, s
b = b - a
br loop

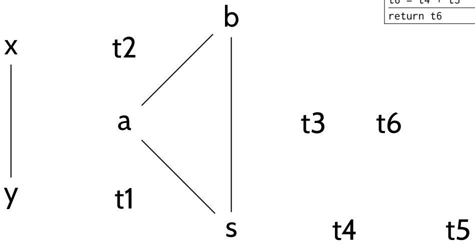

x, y
a = x
b = y
s = 0
br loop
a, b, s
t1 = a - b
bnez t1, body, exit
T
F
a, s
t4 = 2 * a
t5 = 3 * s
t6 = t4 + t5
return t6
a, b, s
t2 = s+a
s = t2+b
t3 = a - b
bgz t3, thn, els
F
T
a, b, s
a = a - b
br loop
a, b, s
b = b - a
br loop

Interference graph

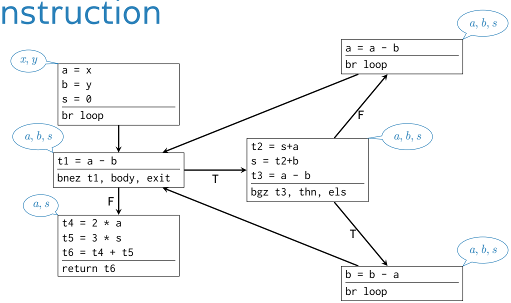
- An interference graph for a CFG is an undirected graph (V, I) where
  - Vertices V = program variables
  - Edges I connect variables x and y iff there is some program point where x and y are simultaneously live
    - **"Program point" includes intermediate points within basic blocks**
- Vertices that are adjacent in the interference graph cannot be stored in the same memory location

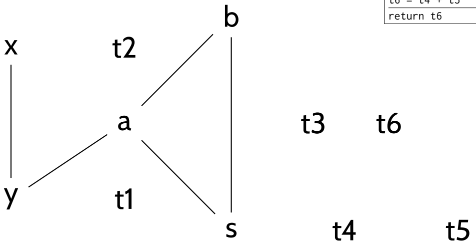
Interference graph construction



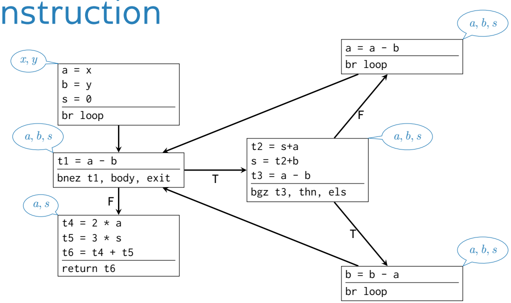


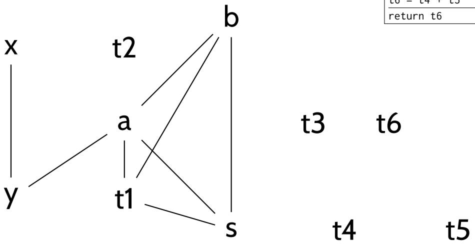
Interference graph construction



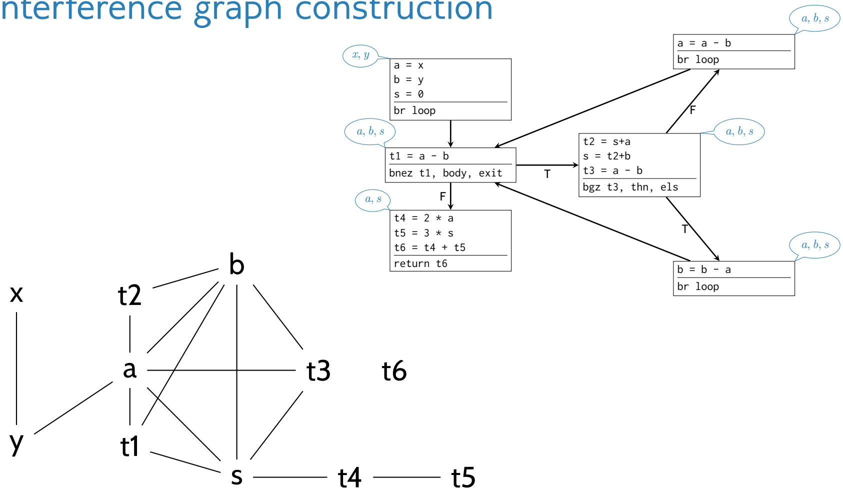


Interference graph construction





Interference graph construction



Interference graph coloring

- A K-coloring of the interference graph is a function $c : V \rightarrow \{1, ..., K\}$ such that if x and y are adjacent in I, then $c(x) \neq c(y)$.
- Basic idea (due to Chaitin): if a processor has K registers, then a K-coloring of its interference graph corresponds to a valid memory layout.

Interference graph coloring

- A K-coloring of the interference graph is a function $c : V \to \{1, \ldots, K\}$ such that if x and y are adjacent in I, then $c(x) \neq c(y)$.
- Basic idea (due to Chaitin): if a processor has K registers, then a K-coloring of its interference graph corresponds to a valid memory layout.
- Problem: Determining whether a graph is K-colorable is NP-complete
  - *But*: we don't need an optimal coloring – any coloring will do
  - If we use more colors than we have registers, can *spill*: place the variable in memory rather than a register
    - May need to reserve some registers for intermediate computations (e.g., accessing memory)

Greedy coloring

- Idea: assign colors to nodes in some order
  - For each node, assign a color that isn't already assigned to one of its neighbors
    - No color available $\Rightarrow$ spill
  - If a node has $< K$ neighbors, a color is always available

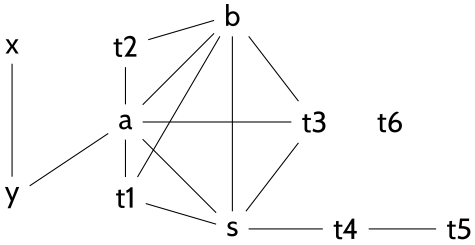
Greedy coloring

- Idea: assign colors to nodes in some order
  - For each node, assign a color that isn't already assigned to one of its neighbors
    - No color available $\Rightarrow$ spill
  - If a node has $< K$ neighbors, a color is always available
- Process:
  - Simplify: choose a node with $< K$ neighbors. Add it to a stack & remove it from the graph
  - Spill: if all nodes have $\geq K$ neighbors, choose one to *potentially* spill. Add it to a stack & remove it from the graph.
  - Color: traverse the stack, assigining colors to the *Simplified* vertices, and either color or spill *Spilled* vertices

Greedy coloring

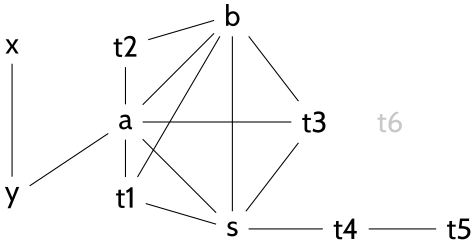
- Idea: assign colors to nodes in some order
  - For each node, assign a color that isn't already assigned to one of its neighbors
    - No color available $\Rightarrow$ spill
  - If a node has $< K$ neighbors, a color is always available
- Process:
  - Simplify: choose a node with $< K$ neighbors. Add it to a stack & remove it from the graph
  - Spill: if all nodes have $\geq K$ neighbors, choose one to *potentially* spill. Add it to a stack & remove it from the graph.
  - Color: traverse the stack, assigining colors to the *Simplified* vertices, and either color or spill *Spilled* vertices
- Not optimal: may use more colors than needed
  - fast & works well in practice.

3-coloring the interference graph



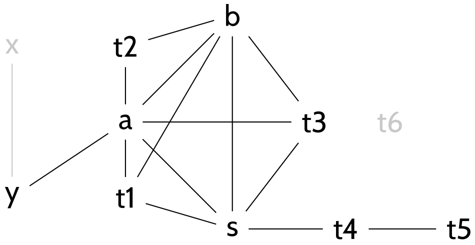
Stack:

3-coloring the interference graph



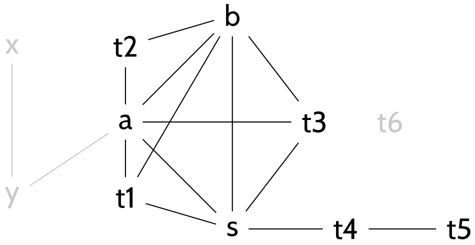
Stack: t6

3-coloring the interference graph



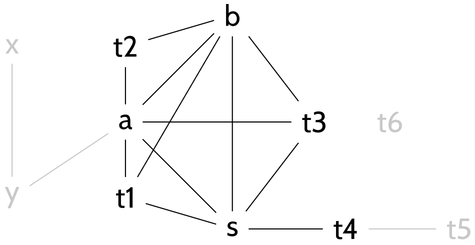
Stack: t6,x

3-coloring the interference graph



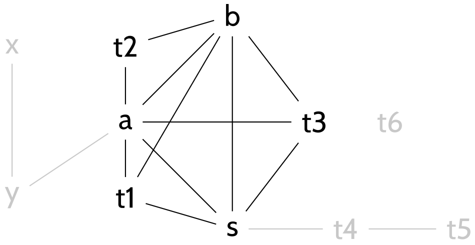
Stack: t6,x,y

3-coloring the interference graph



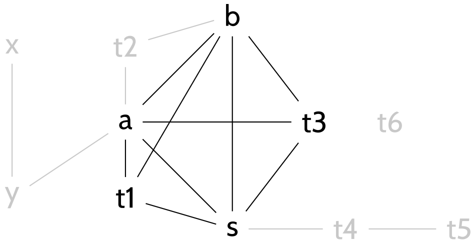
Stack: t6,x,y,t5

3-coloring the interference graph



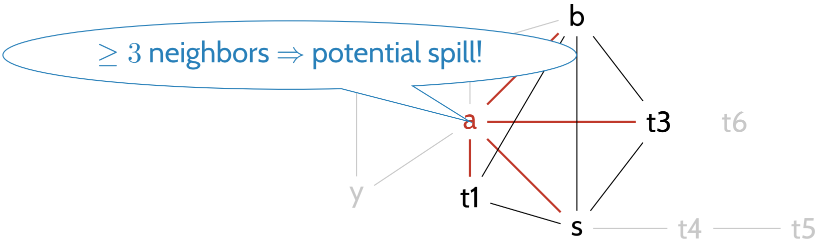
Stack: t6,x,y,t5,t4

3-coloring the interference graph



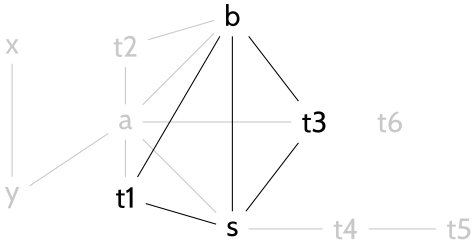
Stack: t6,x,y,t5,t4,t2

3-coloring the interference graph



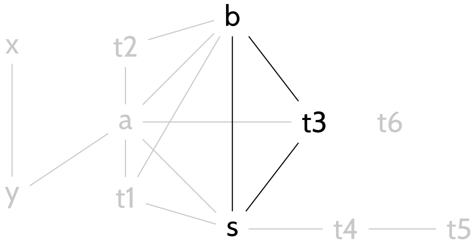
Stack: t6,x,y,t5,t4,t2

3-coloring the interference graph



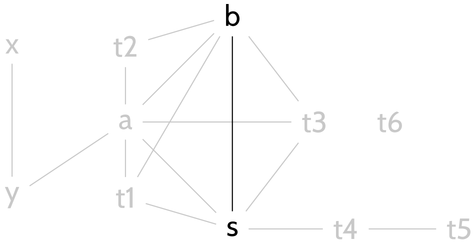
Stack: t6,x,y,t5,t4,t2,a

3-coloring the interference graph



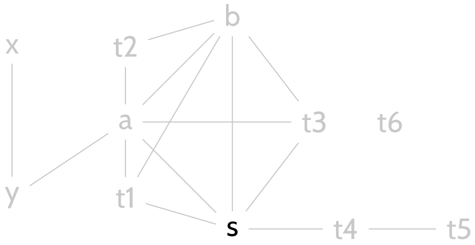
Stack: t6,x,y,t5,t4,t2,a,t1

3-coloring the interference graph



Stack: t6,x,y,t5,t4,t2,a,t1,t3

3-coloring the interference graph

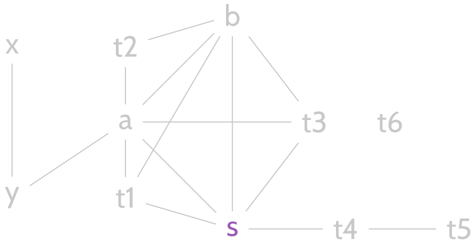


Stack: t6,x,y,t5,t4,t2,a,t1,t3,b

3-coloring the interference graph

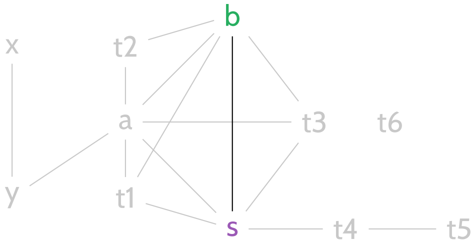
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



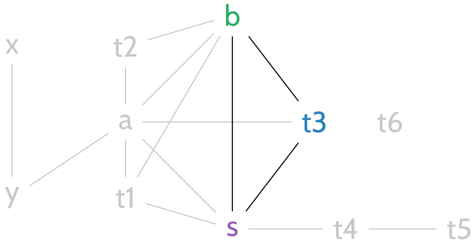
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



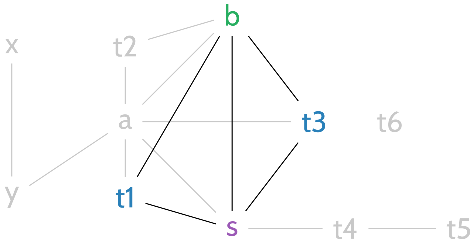
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



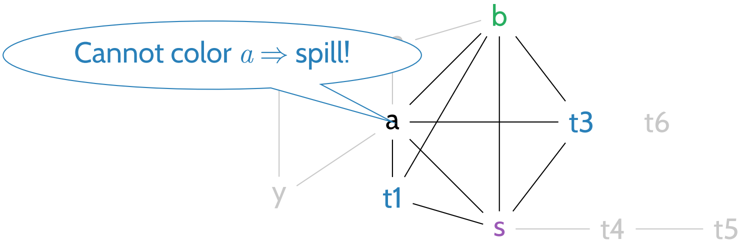
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



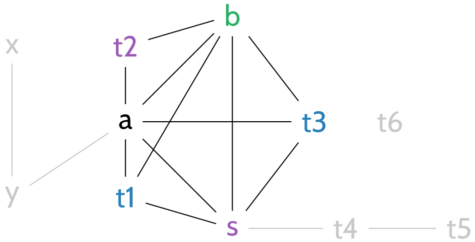
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



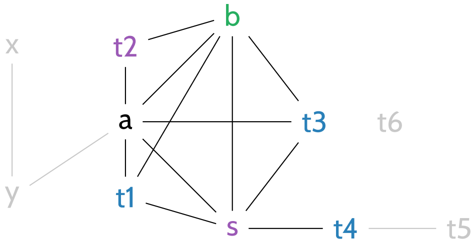
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



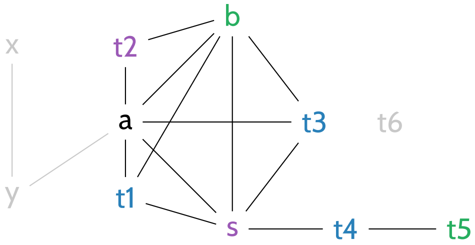
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



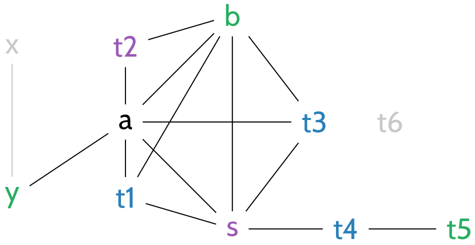
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



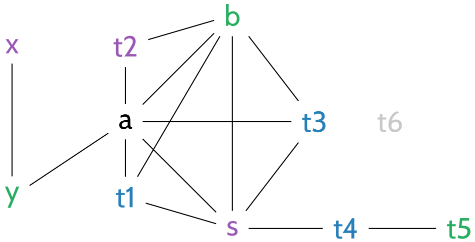
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



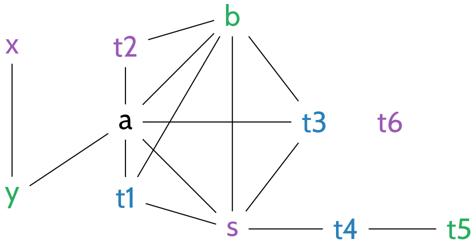
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



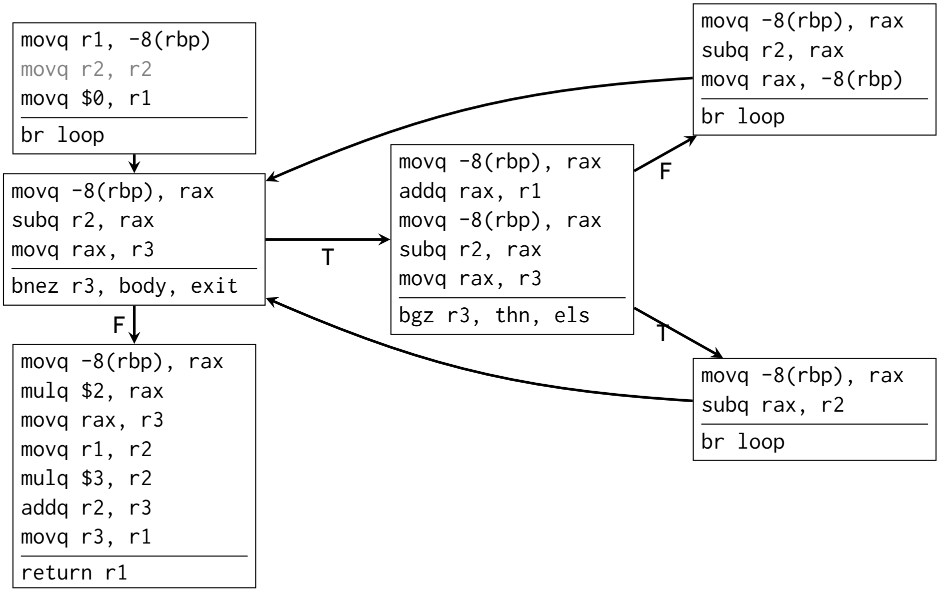
Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

3-coloring the interference graph



Stack: t6,x,y,t5,t4,t2,a,t1,t3,b,s

Suppose we have two reserved registers rax,rcx and three available registers r1,r2,r3



Accessing spilled registers

- Problem: we may need to use registers to access the stack slots that we use to store spilled virtual registers
- Easy option: reserve some registers for memory operations (`rax` and `rcx` in last slide)

Accessing spilled registers

- Problem: we may need to use registers to access the stack slots that we use to store spilled virtual registers
- Easy option: reserve some registers for memory operations (`rax` and `rcx` in last slide)
- Better option: genererate spill code, then re-run register allocator
  - Spill code may use new virtual registers
    - E.g., if `x` is spilled in `xloc`, `y` is spilled in `yloc`,
      `x = y` ⇝ `t = load xloc`; `store t yloc`
  - When we re-run the register allocator, we must allocate registers to these virtual registers
    - live range for new virtual register is very short
    - use some book-keeping to prevent infinite loop (don't spill virtual registers generated for spill code)

Pre-colored nodes

- Some instructions require the use of certain registers
  - E.g., the `call` must pass parameters in `rdi`, `rsi`, `rdx`, `rcx`, `r08`, `r09`
- Virtual registers that must be assigned a particular register should be considered "pre-colored"
  - Not a target for *Simplify* or *Spill*
  - Terminate register allocator when no *uncolored* nodes remain

Graph coalescing

- May be desirable to place two variables in the same register
  - E.g., if we have an assignment $x := y$ and x and y are in the same register, we can elide the `mov` instruction

Graph coalescing

- May be desirable to place two variables in the same register
  - E.g., if we have an assignment $x := y$ and x and y are in the same register, we can elide the `mov` instruction
- *Graph coalescing* collapses two (non-adjacent) vertices into one vertex with the neighborhood of both
- Coalescing creates more register pressure (high-degree vertices)

Graph coalescing

- May be desirable to place two variables in the same register
  - E.g., if we have an assignment $x := y$ and x and y are in the same register, we can elide the `mov` instruction
- *Graph coalescing* collapses two (non-adjacent) vertices into one vertex with the neighborhood of both
- Coalescing creates more register pressure (high-degree vertices)
- Strategies to preserve K-colorability:
  - Briggs': coalesce only when the resulting node has $< K$ neighbors with degree $\geq K$
    - What about coalescing with a pre-colored node?

Graph coalescing

- May be desirable to place two variables in the same register
  - E.g., if we have an assignment $x := y$ and x and y are in the same register, we can elide the `mov` instruction
- *Graph coalescing* collapses two (non-adjacent) vertices into one vertex with the neighborhood of both
- Coalescing creates more register pressure (high-degree vertices)
- Strategies to preserve K-colorability:
  - Briggs': coalesce only when the resulting node has $< K$ neighbors with degree $\geq K$
    - What about coalescing with a pre-colored node?
  - George's: coalesce x and y only when each neighbor of x is either a neighbor of y or has degree $< K$.

More register allocation

Graph coloring is not the end of the story…

- Spill selection: if an interference graph cannot be simplified, which register should be spilled?
  - Priority based on # of edges, # of uses of the variable, …
- Live range splitting
  - Might be desirable to allocate a single variable in different registers in different code sections
  - SSA already does some of this implicitly!
- See *Modern Compiler Implementation in ML* Ch 11 for (some) more details