Lecture 24

CIS 341: COMPILERS

Announcements

- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Available Soon
  - Due: Wednesday, April 27th

- Final Exam:
  - According to registrar: Monday, May 2nd noon - 2:00pm

LOOPS AND DOMINATORS

Loops in Control-flow Graphs

- Taking into account loops is important for optimizations.
  - The 90/10 rule applies, so optimizing loop bodies is important

- Should we apply loop optimizations at the AST level or at a lower representation?
  - Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is good to interleave them.

- Loops may be hard to recognize at the quadruple / LLVM IR level.
  - Many kinds of loops: while, do/while, for, continue, goto…

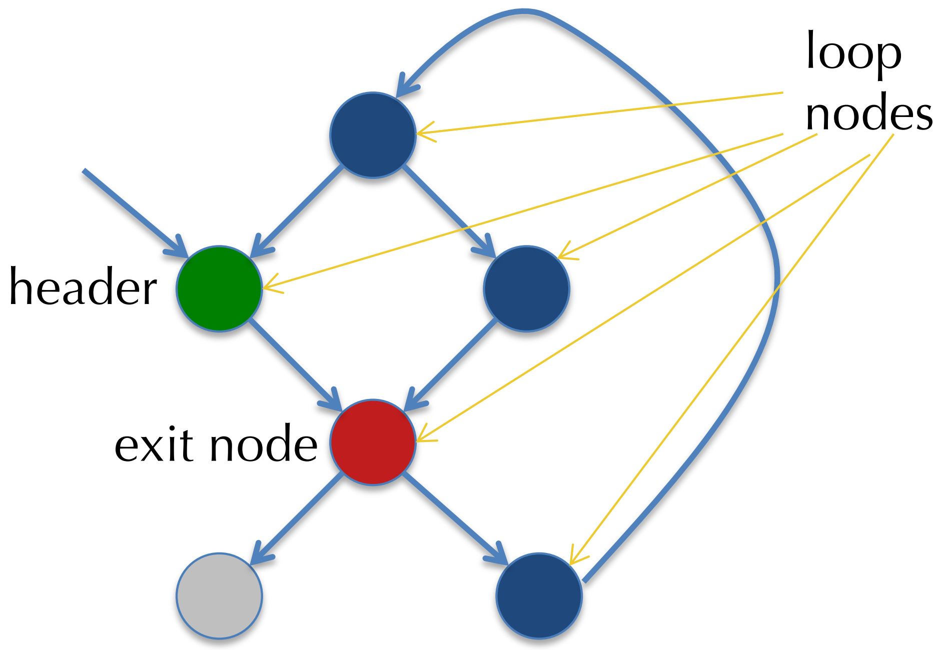
- Problem: *How do we identify loops in the control-flow graph?*

Definition of a Loop

- A *loop* is a set of nodes in the control flow graph.
  - One distinguished entry point called the *header*

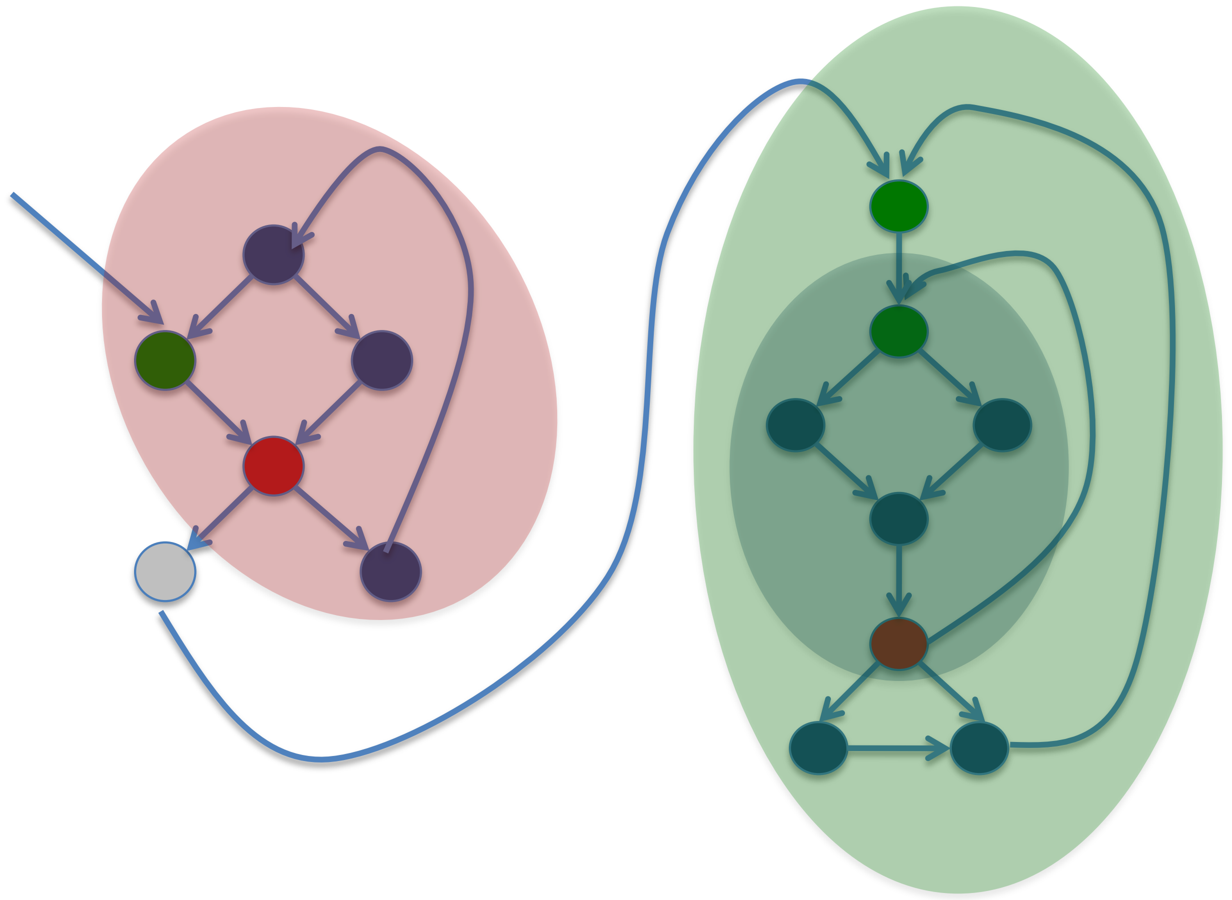
- Every node is reachable from the header & the header is reachable from every node.
  - A loop is a *strongly connected component*

- No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes

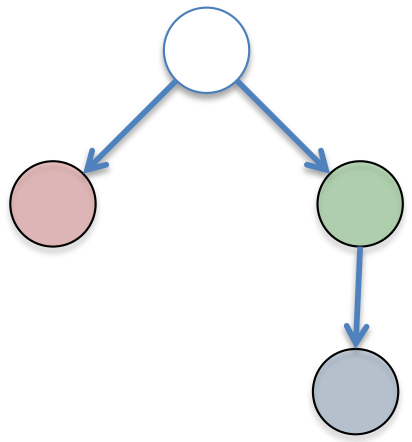


Nested Loops

- Control-flow graphs may contain many loops
- Loops may contain other loops:



Control Tree:



The control tree depicts the nesting structure of the program.

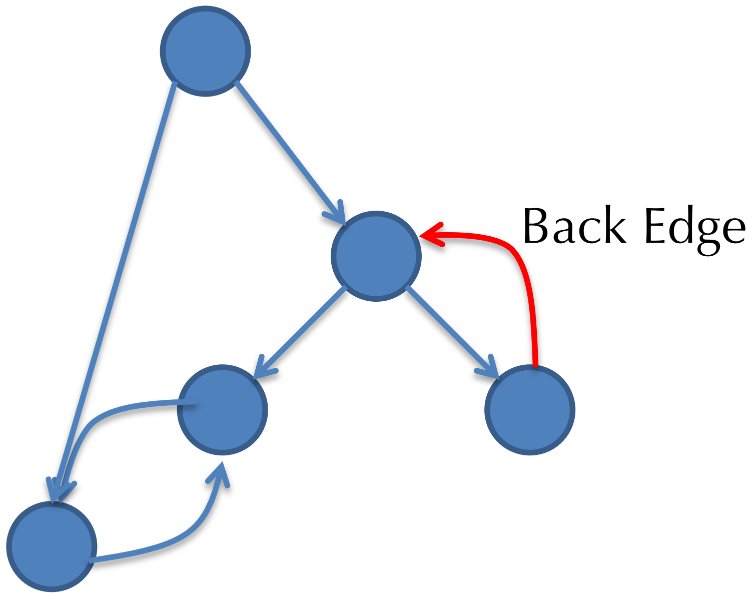
Control-flow Analysis

- Goal: Identify the loops and nesting structure of the CFG.

- Control flow analysis is based on the idea of *dominators*:
- Node A *dominates* node B if the only way to reach B from the start node is through node A.

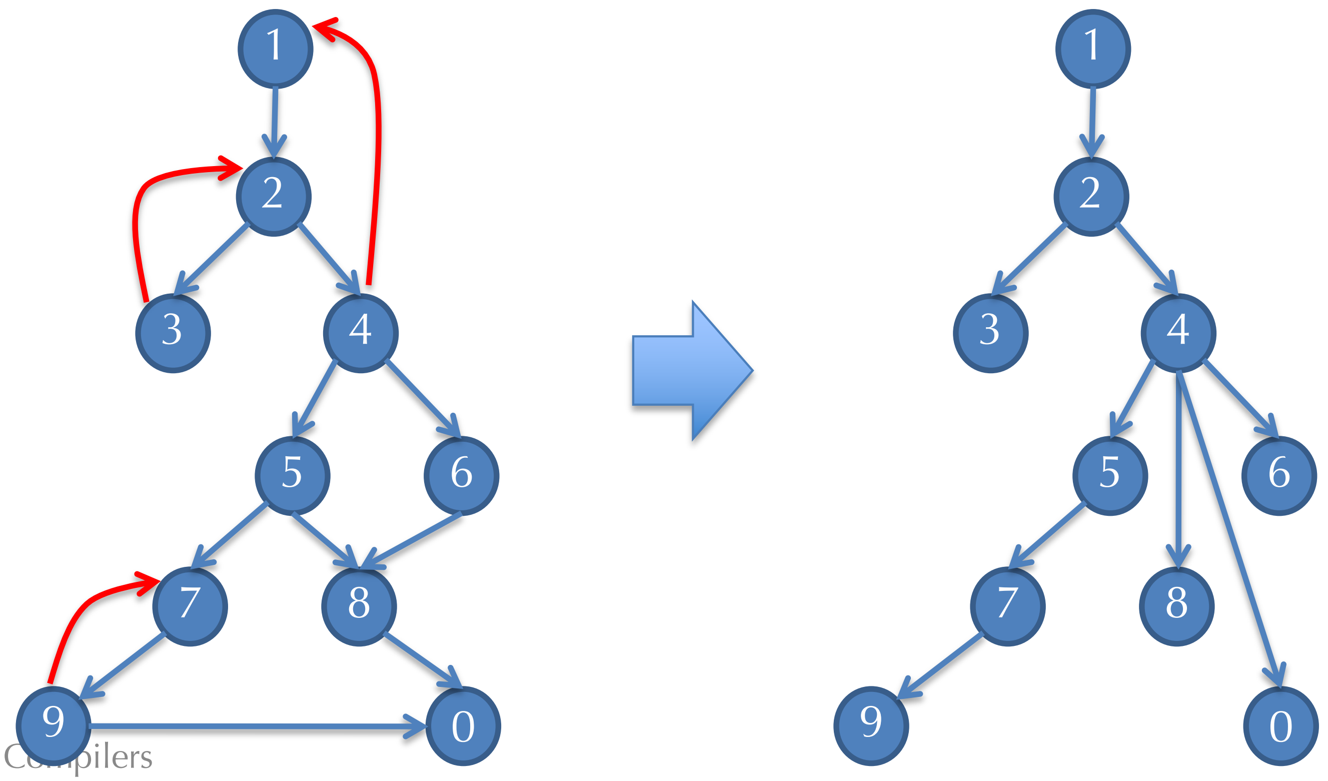
- An edge in the graph is a *back edge* if the target node dominates the source node.

- A loop contains at least one back edge.



Dominator Trees

- Domination is transitive:
  - if A dominates B and B dominates C then A dominates C
- Domination is anti-symmetric:
  - if A dominates B and B dominates A then A = B
- Every flow graph has a dominator tree
  - The Hasse diagram of the dominates relation

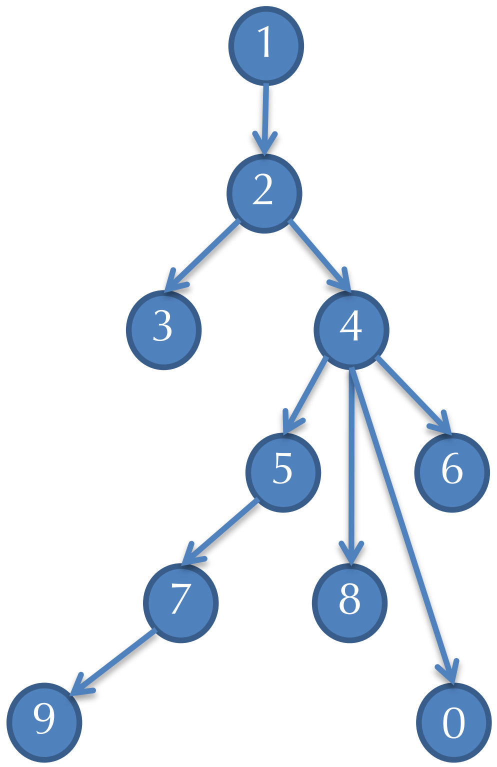


Dominator Dataflow Analysis

- We can define Dom[n] as a forward dataflow analysis.
  - Using the framework we saw earlier: Dom[n] = out[n] where:
- "A node B is dominated by another node A if A dominates *all* of the predecessors of B."
  - $\mathrm{in}[n] := \bigcap_{n' \in \mathrm{pred}[n]} \mathrm{out}[n']$
- "Every node dominates itself."
  - $\mathrm{out}[n] := \mathrm{in}[n] \cup \{n\}$
- Formally: $\mathcal{L}$ = set of nodes ordered by $\subseteq$
  - $\top = \{\text{all nodes}\}$
  - $F_n(x) = x \cup \{n\}$
  - $\sqcap$ is $\cap$
- Easy to show monotonicity and that F_n distributes over meet.
  - So algorithm terminates and is MOP

Improving the Algorithm

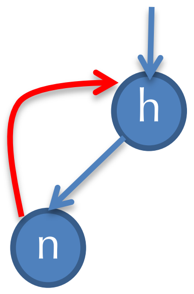
- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
  - *e.g.*, Dom[8] = {1,2,4,8}, Dom[7] = {1,2,4,5,7}
  - There is a lot of sharing among the nodes
- More efficient way to represent Dom sets is to store the dominator *tree*.
  - doms[b] = immediate dominator of b
  - doms[8] = 4, doms[7] = 5
- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
  - Traverse up tree, looking for least common ancestor:
  - Dom[8] ∩Dom[7] = Dom[4]



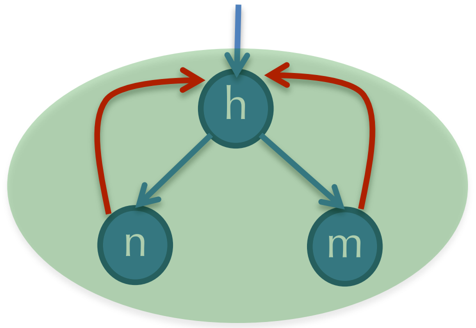
- See: "A Simple, Fast Dominance Algorithm" Cooper, Harvey, and Kennedy

Completing Control-flow Analysis

- Dominator analysis identifies *back edges*:
  - Edge n → h where h dominates n
- Each back edge has a *natural loop*:
  - h is the header
  - All nodes reachable from h that also reach n without going through h
- For each back edge n → h, find the natural loop:
  - {n′ | n is reachable from n′ in G – {h}} ∪ {h}

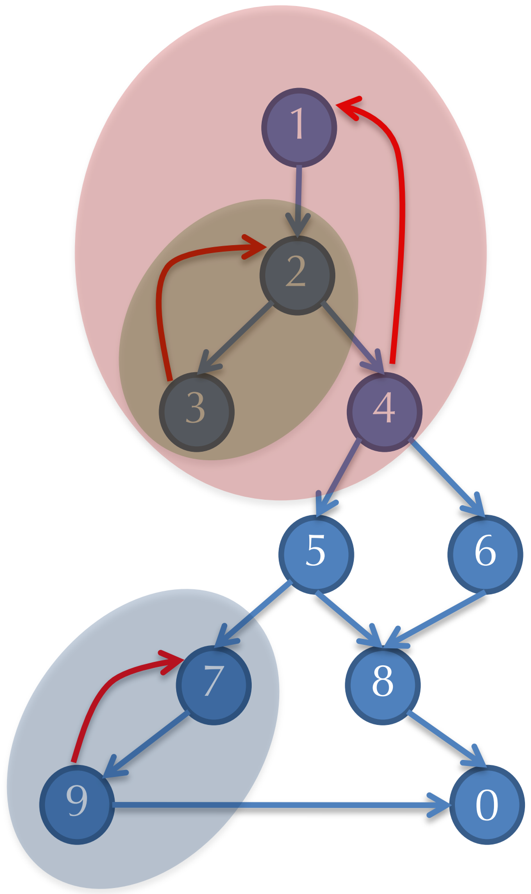


- Two loops may share the same header: merge them



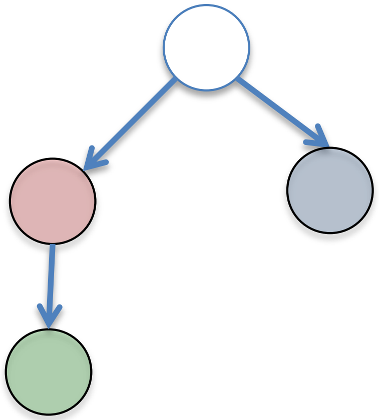
- Nesting structure of loops is determined by set inclusion
  - Can be used to build the control tree

Example Natural Loops



Natural Loops

Control Tree:



The control tree depicts the nesting structure of the program.

Uses of Control-flow Information

- Loop nesting depth plays an important role in optimization heuristics.
  - Deeply nested loops pay off the most for optimization.

- Need to know loop headers / back edges for doing
  - loop invariant code motion
  - loop unrolling

- Dominance information also plays a role in converting to SSA form
  - Used internally by LLVM to do register allocation.

See HW6: Dataflow Analysis

IMPLEMENTATION

REGISTER ALLOCATION

Register Allocation Problem

- Given: an IR program that uses an unbounded number of temporaries
  - e.g. the uids of our LLVM programs
- Find: a mapping from temporaries to machine registers such that
  - program semantics is preserved (i.e. the behavior is the same)
  - register usage is maximized
  - moves between registers are minimized
  - calling conventions / architecture requirements are obeyed
- Stack Spilling
  - If there are k registers available and m > k temporaries are live at the same time, then not all of them will fit into registers.
  - So: "spill" the excess temporaries to the stack.

Linear-Scan Register Allocation

Simple, greedy register-allocation strategy:

1. Compute liveness information: `live(x)`
   – recall: `live(x)` is the set of uids that are live on entry to `x`'s definition
2. Let `pal` be the set of usable registers
   – usually reserve a couple for spill code [our implementation uses rax,rcx]
3. Maintain "layout" `uid_loc` that maps uids to locations
   – locations include registers and stack slots n, starting at n=0
4. Scan through the program. For each instruction that defines a uid `x`
   – `used` = `{r | reg r = uid_loc(y)` s.t. `y` ∈ `live(x)}`
   – `available = pal - used`
   – If `available` is empty: *// no registers available, spill*
     `uid_loc(x) := slot n` ; n = n + 1
   – Otherwise, pick `r` in `available`: *// choose an available register*
     `uid_loc(x) := reg r`

For HW6

- HW 6 implements two naive register allocation strategies:
  - `none`: spill all registers
  - `greedy`: uses linear scan
- Also offers choice of liveness
  - `trivial`: assume all variables are live everywhere
  - `dataflow`: use the dataflow algorithms

- Your job: do "better" than these.
- Quality Metric:
  - registers other than rbp count positively
  - rbp counts negatively (it is used for spilling)
  - shorter code is better

- Linear scan is OK
  - but… how can we do better?

GRAPH COLORING

Register Allocation

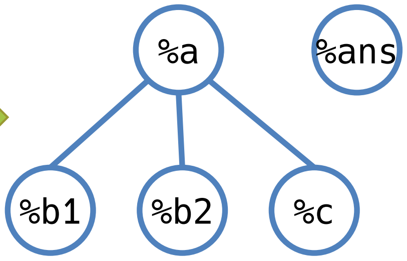
Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is live at the same time as m
3. Try to color the graph
   - Each color corresponds to a register
4. In case step 3. fails, "spill" a register to the stack and repeat the whole process.
5. Rewrite the program to use registers

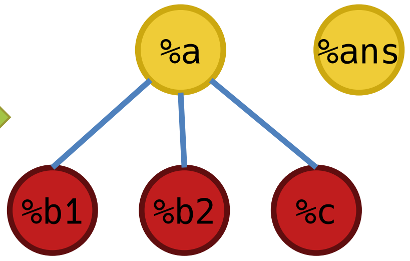
Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



Interference Graph



2-Coloring of the graph
red = r8
yellow = r9

Register Allocation Questions

- Can we efficiently find a k-coloring of the graph whenever possible?
  - Answer: in general the problem is NP-complete (it requires search)
  - But, we can do an efficient approximation using heuristics.
- How do we assign registers to colors?
  - If we do this in a smart way, we can eliminate redundant MOV instructions.
- What do we do when there aren't enough colors/registers?
  - We have to use stack space, but how do we do this effectively?

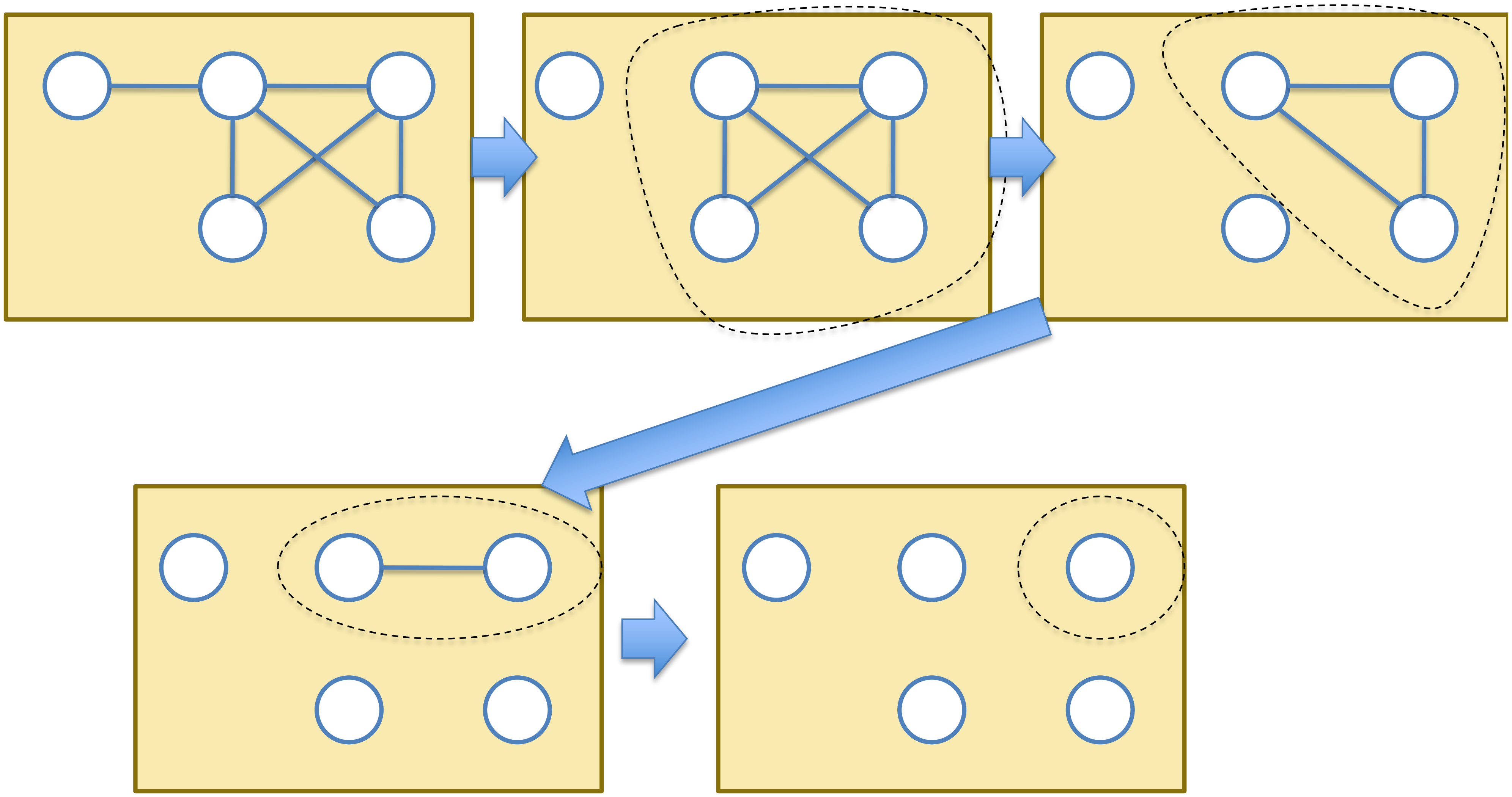
Coloring a Graph: Kempe's Algorithm

Kempe [1879] provides this algorithm for K-coloring a graph.

It's a recursive algorithm that works in three steps:

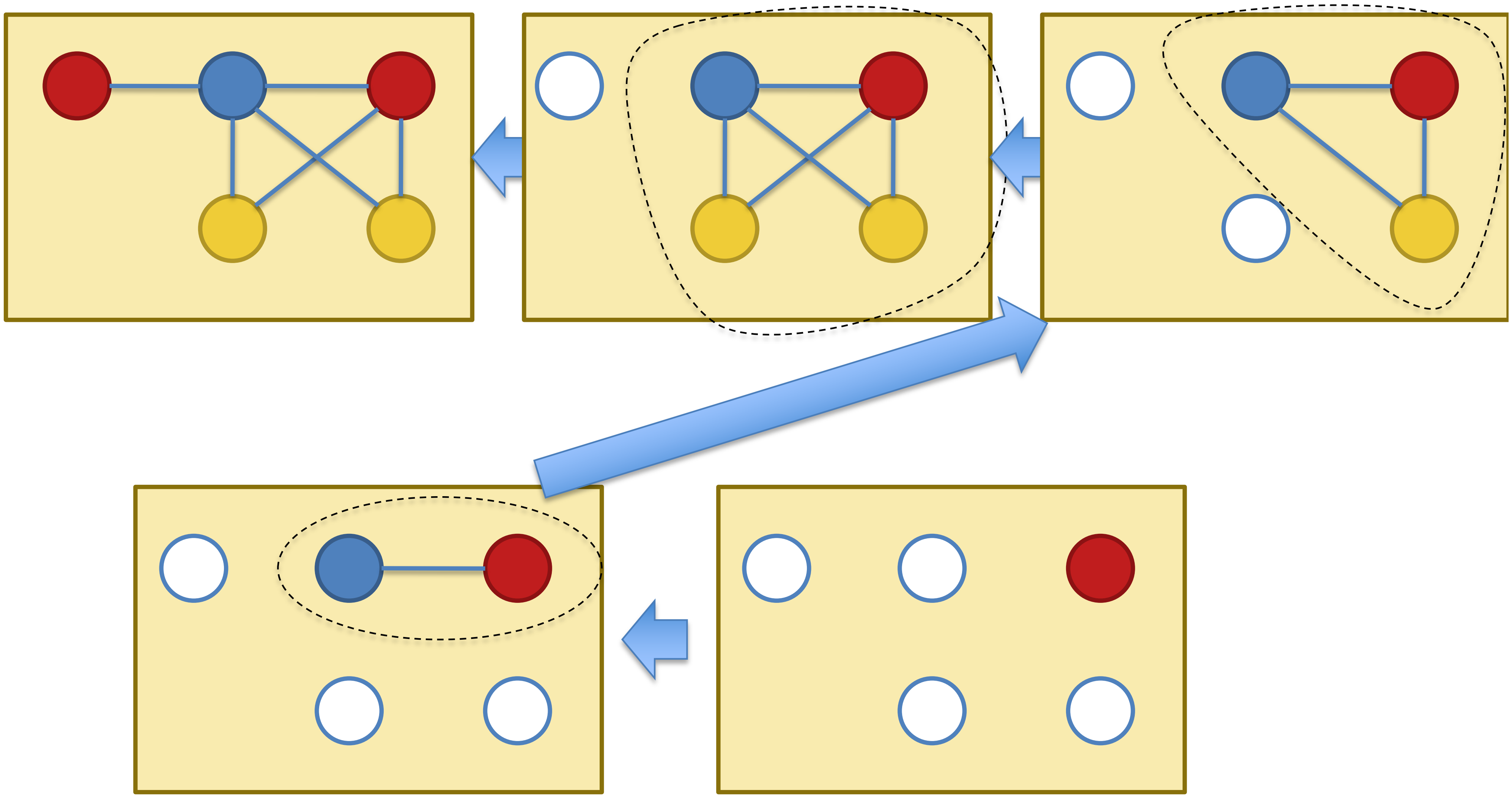
1. Find a node with degree < K and cut it out of the graph.
   - Remove the nodes and edges.
   - This is called *simplifying* the graph
2. Recursively K-color the remaining subgraph
3. When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was < K). Pick such a color.

Example: 3-color this Graph



Recursing Down the Simplified Graphs

Example: 3-color this Graph

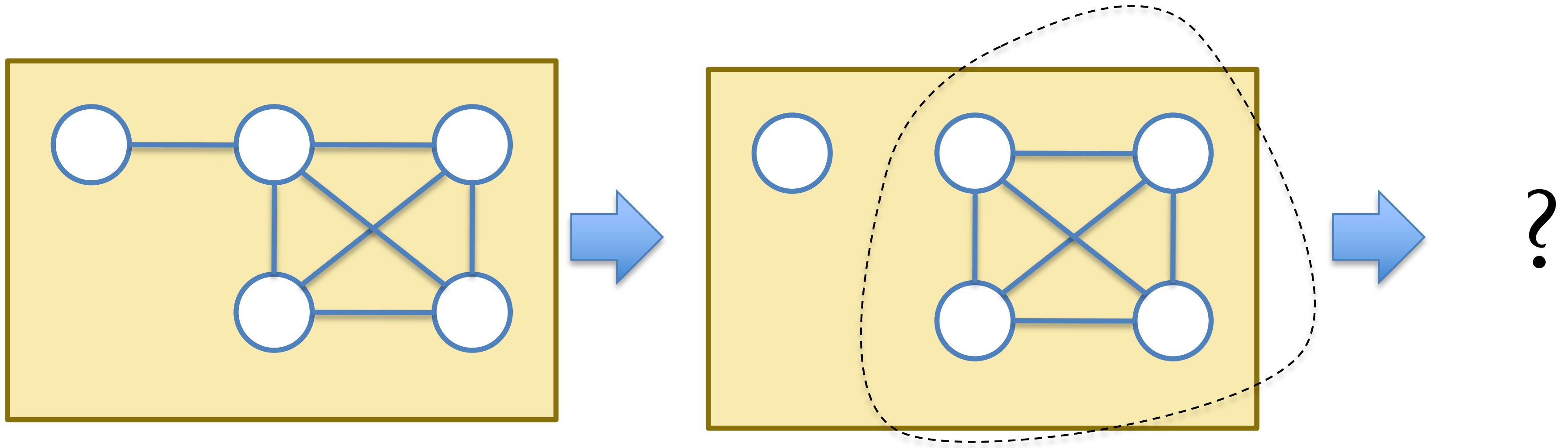


Assigning Colors on the way back up.

Failure of the Algorithm

- If the graph cannot be colored, it will simplify to a graph where every node has at least K neighbors.
  - This can happen even when the graph is K-colorable!
  - This is a symptom of NP-hardness (it requires search)

- Example: When trying to 3-color this graph:



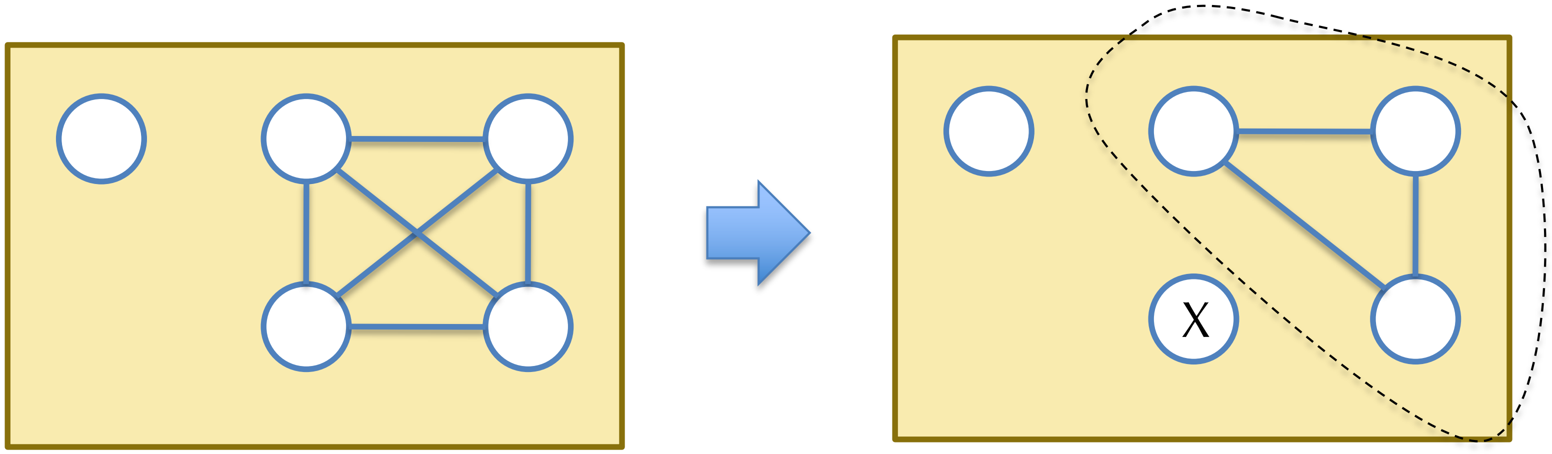
Spilling

- Idea: If we can't K-color the graph, we need to store one temporary variable on the stack.
- Which variable to spill?
  - Pick one that isn't used very frequently
  - Pick one that isn't used in a (deeply nested) loop
  - Pick one that has high interference
    (since removing it will make the graph easier to color)
- In practice: some weighted combination of these criteria

- When coloring:
  - Mark the node as spilled
  - Remove it from the graph
  - Keep recursively coloring

Spilling, Pictorially

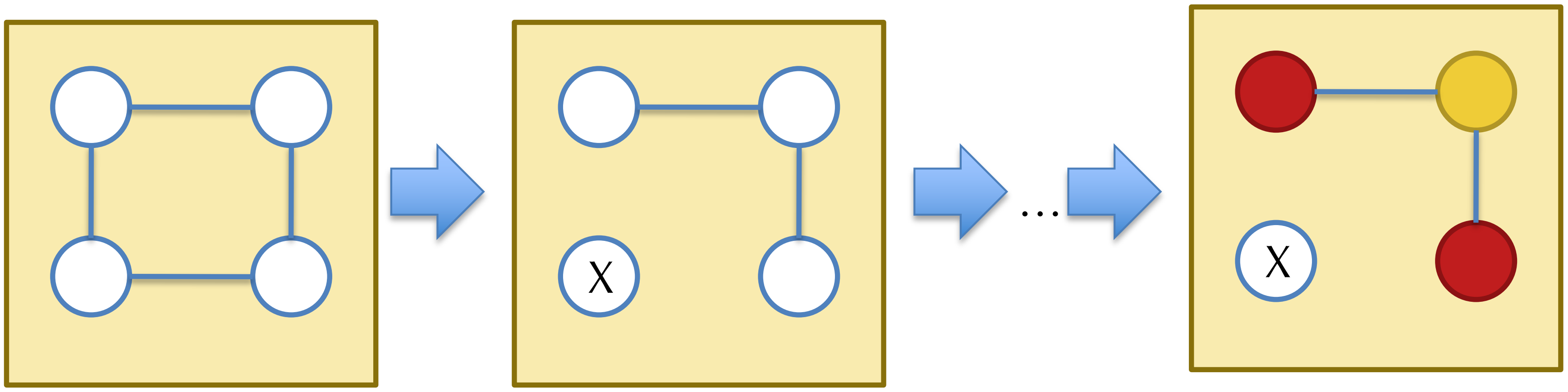
- Select a node to spill
- Mark it and remove it from the graph
- Continue coloring



Optimistic Coloring

- Sometimes it is possible to color a node marked for spilling.
  - If we get "lucky" with the choices of colors made earlier.

- Example: When 2-coloring this graph:



- Even though the node was marked for spilling, we can color it.
- So: on the way down, mark for spilling, but don't actually spill…

Accessing Spilled Registers

- If optimistic coloring fails, we need to generate code to move the spilled temporary to & from memory.
- Option 1: Reserve registers specifically for moving to/from memory.
  - Con: Need at least two registers (one for each source operand of an instruction), so decreases total # of available registers by 2.
  - Pro: Only need to color the graph once.
  - Not good on 32bit x86 because there are too few registers & too many constraints on how they can be used.
  - OK on 64bit x86 and other processors. (We use this for HW6)

- Option 2: Rewrite the program to use a new temporary variable, with explicit moves to/from memory.
  - Pro: Need to reserve fewer registers.
  - Con: Introducing temporaries changes live ranges, so must recompute liveness & recolor graph

Example Spill Code

- Suppose temporary `t` is marked for spilling to stack slot located at `[rbp+offs]`
- Rewrite the program like this:

```
t = a op b;           t = a op b            // defn. of t
…                     Mov [rbp+offs], t
                      …
x = t op c            Mov t37, [rbp+offs]   // use 1 of t
…                     x = t37 op c
                      …
y = d op t            Mov t38, [rbp+offs]   // use  2 of t
                      y = d op t38
```

- Here, `t37` and `t38` are freshly generated temporaries that replace `t` for different uses of `t`.
- Rewriting the code in this way breaks `t`'s live range up:

  `t`, `t37`, `t38` are only live across one edge

Precolored Nodes

- Some variables must be pre-assigned to registers.
  - E.g. on X86 the multiplication instruction: IMul must define %rax
  - The "Call" instruction should kill the caller-save registers %rax, %rcx, %rdx.
  - Any temporary variable live across a call interferes with the caller-save registers.

- To properly allocate temporaries, we treat registers as nodes in the interference graph with pre-assigned colors.
  - Pre-colored nodes can't be removed during simplification.
  - Trick: Treat pre-colored nodes as having "infinite" degree in the interference graph – this guarantees they won't be simplified.
  - When the graph is empty except the pre-colored nodes, we have reached the point where we start coloring the rest of the nodes.

Picking Good Colors

- When choosing colors during the coloring phase, *any* choice is semantically correct, but some choices are better for performance.
- Example:

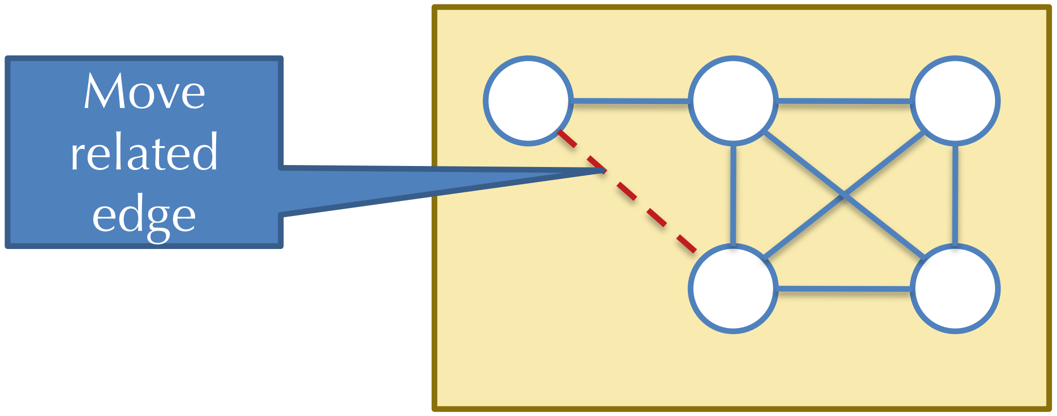
```
movq t1, t2
```

  - If t1 and t2 can be assigned the same register (color) then this move is redundant and can be eliminated.

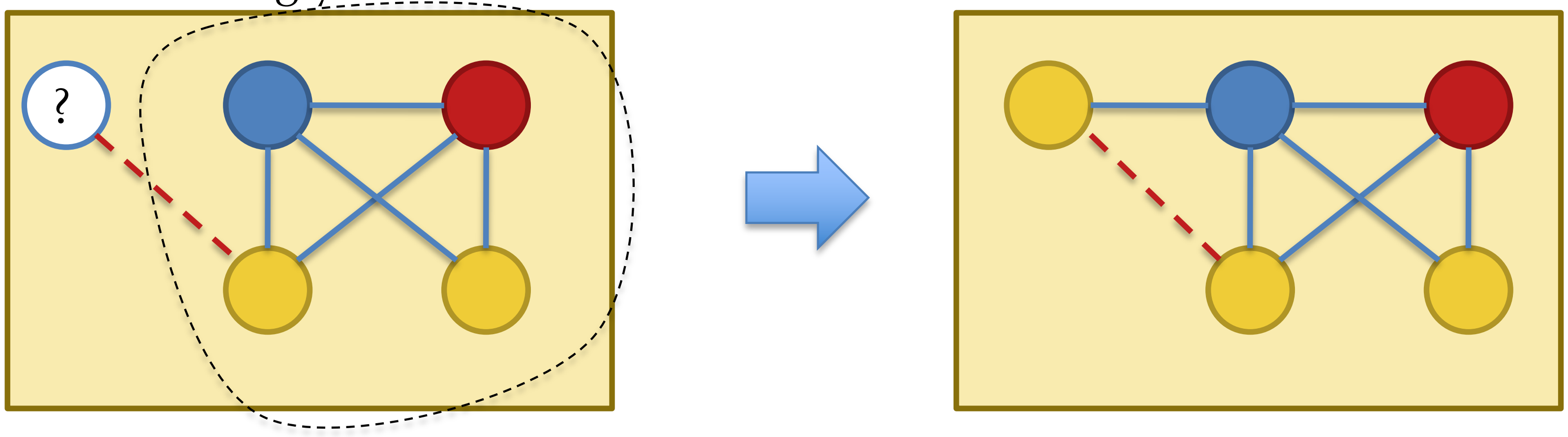
- A simple color choosing strategy that helps eliminate such moves:
  - Add a new kind of "move related" edge between the nodes for t1 and t2 in the interference graph.
  - When choosing a color for t1 (or t2), if possible, pick a color of an already colored node reachable by a move-related edge.

Example Color Choice

- Consider 3-coloring this graph, where the dashed edge indicates that there is a Mov from one temporary to another.

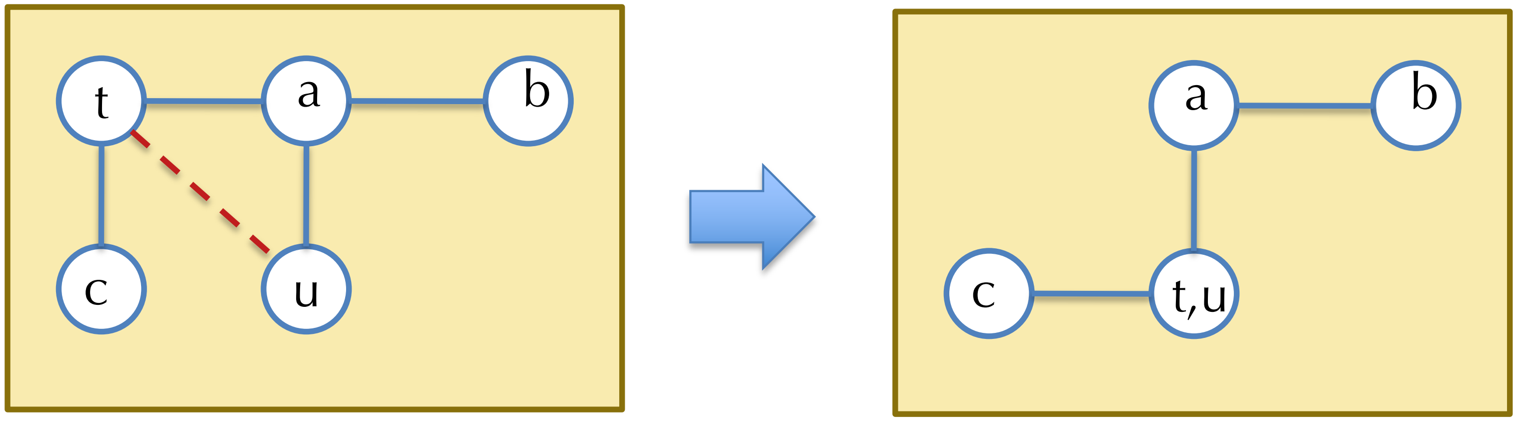


- After coloring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.

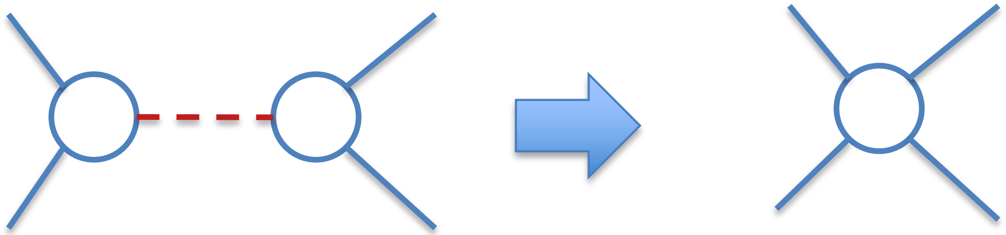


Coalescing Interference Graphs

- A more aggressive strategy is to *coalesce* nodes of the interference graph if they are connected by move-related edges.
  - Coalescing the nodes *forces* the two temporaries to be assigned the same register.



- Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated.
- Problem: coalescing can sometimes increase the degree of a node.



Conservative Coalescing

- Two strategies are guaranteed to preserve the k-colorability of the interference graph.

1. *Brigg's strategy*: It's safe to coalesce x & y if the resulting node will have fewer than k neighbors (with degree ≥ k).

2. *George's strategy:* We can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree < k.

Complete Register Allocation Algorithm

1. Build interference graph (precolor nodes as necessary).
   - Add move related edges
2. Reduce the graph (building a stack of nodes to color).
   1. Simplify the graph as much as possible without removing nodes that are move related (i.e. have a move-related neighbor). Remaining nodes are high degree or move-related.
   2. Coalesce move-related nodes using Brigg's or George's strategy.
   3. Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced.
   4. If no nodes can be coalesced *freeze* (remove) a move-related edge and keep trying to simplify/coalesce.
3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2.
4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack).
   1. If a node must be spilled, insert spill code as on slide 14 and rerun the whole register allocation algorithm starting at step 1.

Last details

- After register allocation, the compiler should do a peephole optimization pass to remove redundant moves.
- Some architectures specify calling conventions that use registers to pass function arguments.
  - It's helpful to move such arguments into temporaries in the function prelude so that the compiler has as much freedom as possible during register allocation. (Not an issue on X86, though.)

Phi nodes

Alloc "promotion"

Register allocation

REVISITING SSA

Single Static Assignment (SSA)

- LLVM IR names (via `%uids`) *all* intermediate values computed by the program.
- It makes the order of evaluation explicit.
- Each `%uid` is assigned to only once
  - Contrast with the mutable quadruple form
  - Note that dataflow analyses had these kill[n] sets because of updates to variables…
- Naïve implementation of backend: map `%uids` to stack slots
- Better implementation: map `%uids` to registers (as much as possible)

- Question: How do we convert a source program to make maximal use of `%uids`, rather than alloca-created storage?
  - two problems: control flow & location in memory

- Then: How do we convert SSA code to x86, mapping `%uids` to registers?
  - Register allocation.

Alloca vs. %UID

- Current compilation strategy:

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

$\Rightarrow$

```
%x = alloca i64
%y = alloca i64
store i64* %x, 3
store i64* %y, 0
%x1 = load %i64* %x
%tmp1 = add i64 %x1, 1
store i64* %x, %tmp1
%x2 = load %i64* %x
%tmp2 = add i64 %x2, 2
store i64* %y, %tmp2
```

- Directly map source variables into `%uids`?

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

$\Rightarrow$

```
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

$\Rightarrow$

```
%x1 = add i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %x2, 2
```

- Does this always work?

What about If-then-else?

- How do we translate this into SSA?

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
   %y1 = …
   %x1 = …
   %z1 = …
   %p = icmp …
   br i1 %p, label %then, label %else
then:
   %x2 = add i64 %y1, 1
   br label %merge
else:
   %x3 = mult i64 %y1, 2
merge:
   %z2 = %add i64 ???, 3
```

- What do we put for `???`

Phi Functions

- Solution: ϕ functions
  - Fictitious operator, used only for analysis
    - implemented by Mov at x86 level
  - Chooses among different versions of a variable based on the path by which control enters the phi node.
    `%uid = phi <ty> v`$_1$`, <label`$_1$`>, … , v`$_n$`, <label`$_n$`>`

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %x4 = phi i64 %x2, %then, %x3, %else
  %z2 = %add i64 %x4, 3
```

Phi Nodes and Loops

- Importantly, the `%uids` on the right-hand side of a phi node can be defined "later" in the control-flow graph.
  - Means that `%uids` can hold values "around a loop"
  - Scope of `%uids` is defined by *dominance*

```
entry:
  %y1 = …
  %x1 = …
  br label %body

body:
  %x2 = phi i64 %x1, %entry, %x3, %body
  %x3 = add i64 %x2, %y1
  %p = icmp slt i64, %x3, 10
  br i1 %p, label %body, label %after

after:
  …
```

Alloca Promotion

- Not all source variables can be allocated to registers
  - If the address of the variable is taken (as permitted in C, for example)
  - If the address of the variable "escapes" (by being passed to a function)
- An alloca instruction is called promotable if neither of the two conditions above holds

```
entry:
  %x = alloca i64           // %x cannot be promoted
  %y = call malloc(i64 8)
  %ptr = bitcast i8* %y to i64**
  store i65** %ptr, %x      // store the pointer into the heap
```

```
entry:
  %x = alloca i64         // %x cannot be promoted
  %y = call foo(i64* %x) // foo may store the pointer into the heap
```

- Happily, most local variables declared in source programs are promotable
  - That means they can be register allocated

Converting to SSA: Overview

- Start with the ordinary control flow graph that uses allocas
  - Identify "promotable" allocas
- Compute dominator tree information
- Calculate def/use information for each such allocated variable
- Insert ϕ functions for each variable at necessary "join points"

- Replace loads/stores to alloc'ed variables with freshly-generated `%uids`

- Eliminate the now unneeded load/store/alloca instructions.

Where to Place ϕ functions?

- Need to calculate the "Dominance Frontier"

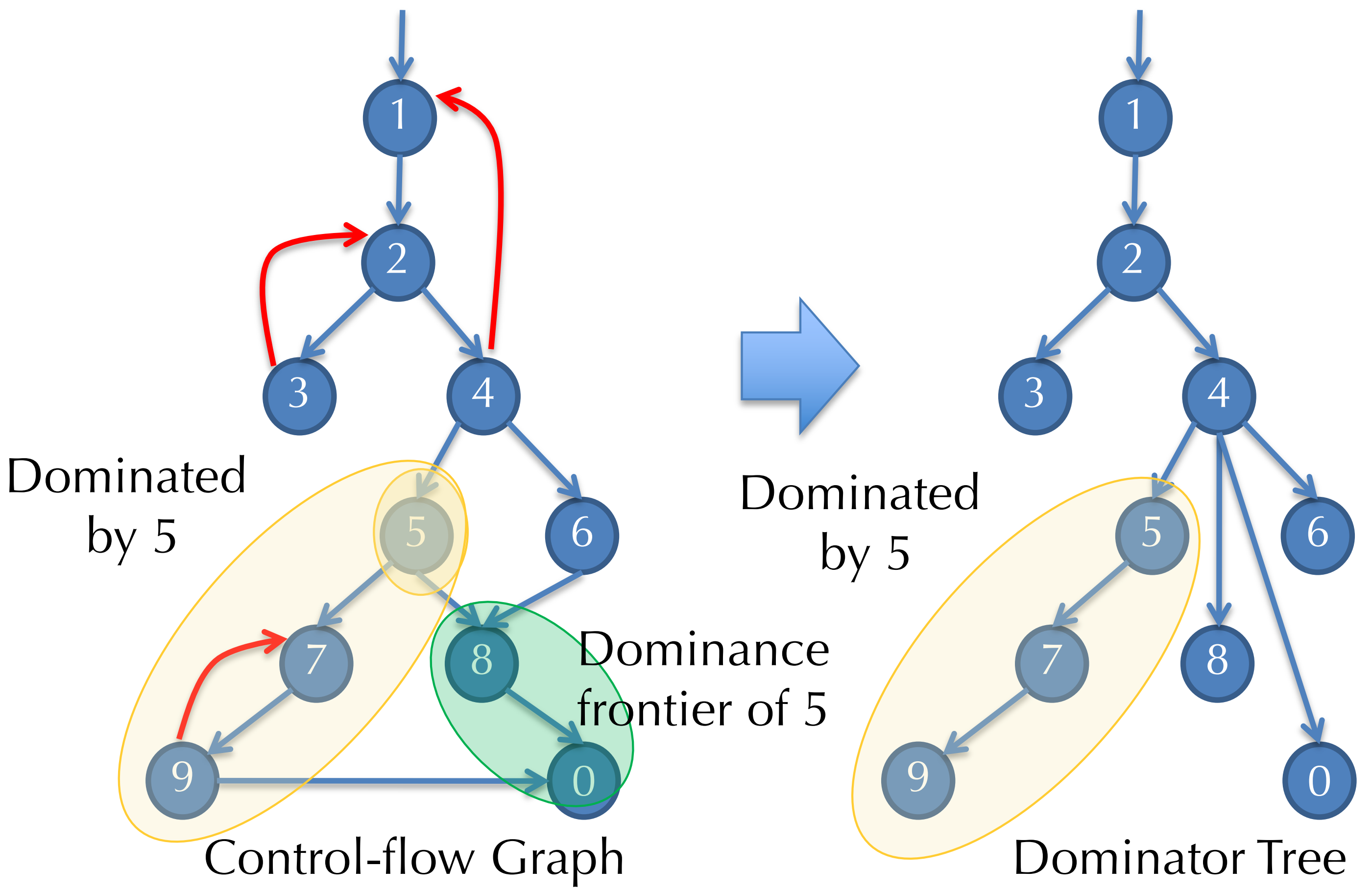
- Node A ***strictly dominates*** node B if A dominates B and A ≠ B.
  - Note: A does not strictly dominate B if A does not dominate B or A = B.

- The ***dominance frontier*** of a node B is the set of all CFG nodes y such that B dominates a predecessor of y but does not strictly dominate y
  - Intuitively: starting at B, there is a path to y, but there is another route to y that does not go through B

- Write DF[n] for the dominance frontier of node n.

Dominance Frontiers

- Example of a dominance frontier calculation results
- DF[1] = {1}, DF[2] = {1,2}, DF[3] = {2}, DF[4] = {1}, DF[5] = {8,0}, DF[6] = {8}, DF[7] = {7,0}, DF[8] = {0}, DF[9] = {7,0}, DF[0] = {}



Algorithm For Computing DF[n]

- Assume that doms[n] stores the dominator tree (so that doms[n] is the *immediate dominator* of n in the tree)
- Adds each B to the DF sets to which it belongs

```
for all nodes B
    if #(pred[B]) ≥ 2                        // (just an optimization)
     for each p ∈pred[B] {
          runner := p                        // start at the predecessor of B
          while (runner ≠ doms[B])           // walk up the tree adding B
               DF[runner] := DF[runner] ∪ {B}
               runner := doms[runner]
    }
```

Insert ϕ at Join Points

- Lift the DF[n] to a set of nodes N in the obvious way:

$$DF[N] = \bigcup_{n \in N} DF[n]$$

- Suppose that at variable x is defined at a set of nodes N.

  $DF_0[N] = DF[N]$
  $DF_{i+1}[N] = DF[DF_i[N] \cup N]$

  Let J[N] be the *least fixed point* of the sequence:

$$DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots$$

  That is, $J[N] = DF_k[N]$ for some k such that $DF_k[N] = DF_{k+1}[N]$
  - J[N] is called the “join points” for the set N
- We insert ϕ functions for the variable x at each node in J[N].
  - $x = \phi(x, x, \ldots, x)$; (one “x” argument for each predecessor of the node)
  - In practice, J[N] is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes, just as in the dataflow solver.

- Intuition:
  - If N is the set of places where x is modified, then DF[N] is the places where phi nodes need to be added, but those also “count” as modifications of x, so we need to insert the phi nodes to capture those modifications too…

Example Join-point Calculation

- Suppose the variable x is modified at nodes 3 and 6
  - Where would we need to add phi nodes?

- $DF_0[\{3,6\}] = DF[\{3,6\}] = DF[3] \cup DF[6] = \{2,8\}$
- $DF_1[\{3,6\}]$

  $= DF[DF_0\{3,6\} \cup \{3,6\}]$

  $= DF[\{2,3,6,8\}]$

  $= DF[2] \cup DF[3] \cup DF[6] \cup DF[8]$

  $= \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\}$
- $DF_2[\{3,6\}]$

  $= \ldots$

  $= \{1,2,8,0\}$

- So $J[\{3,6\}] = \{1,2,8,0\}$ and we need to add phi nodes at those four spots.

Phi Placement Alternative

- Less efficient, but easier to understand:

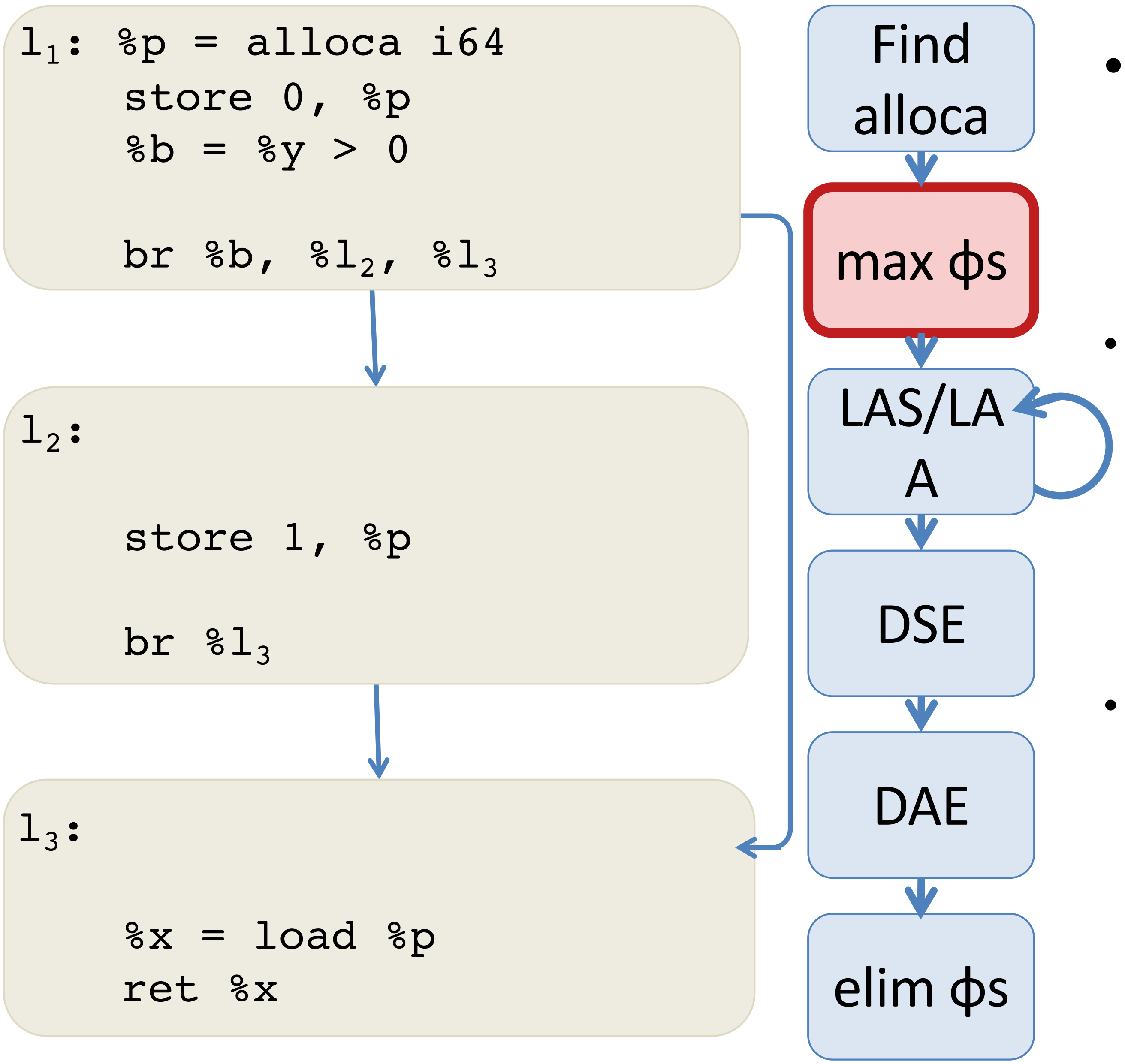
- Place phi nodes "maximally" (i.e. at every node with > 2 predecessors)

- If all values flowing into phi node are the same, then eliminate it:

```
%x = phi   t %y, %pred1   t %y  %pred2  … t %y %predK
// code that uses %x
⇒
// code with %x replaced by %y
```

- Interleave with other optimizations
  - copy propagation
  - constant propagation
  - etc.

Example SSA Optimizations



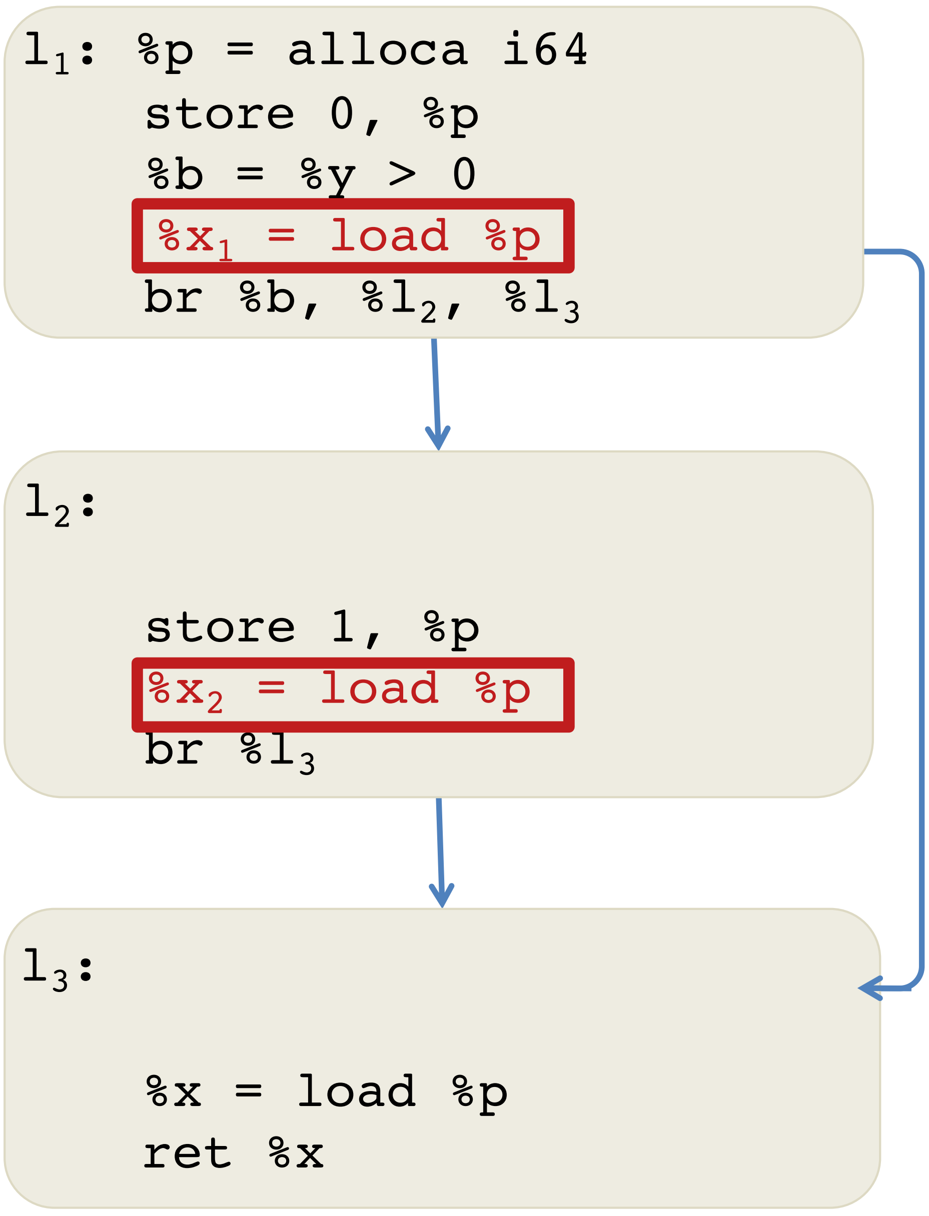
```
l₁: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l₂, %l₃
```

```
l₂:


    store 1, %p

    br %l₃
```

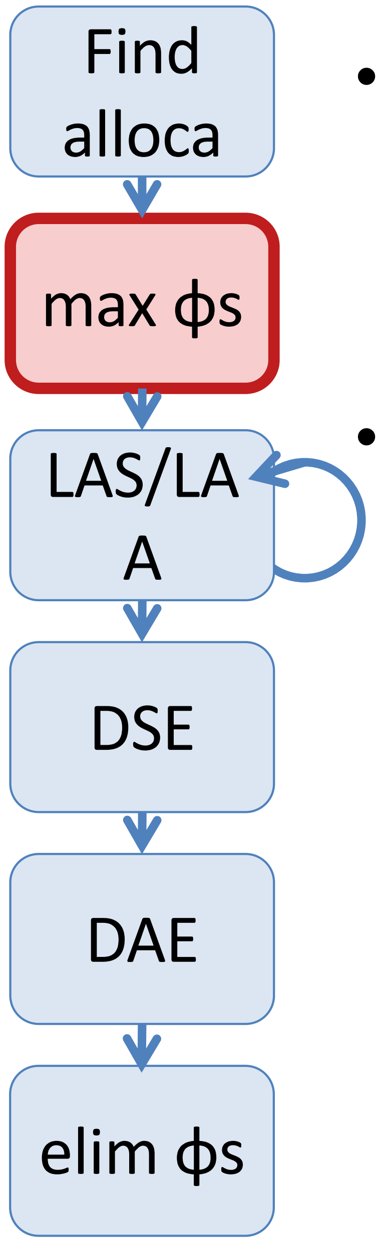
```
l₃:


    %x = load %p
    ret %x
```

- How to place phi nodes without breaking SSA?
- Note: the "real" implementation combines many of these steps into one pass.
  - Places phis directly at the dominance frontier
- This example also illustrates other common optimizations:
  - Load after store/alloca
  - Dead store/alloca elimination

Example SSA Optimizations

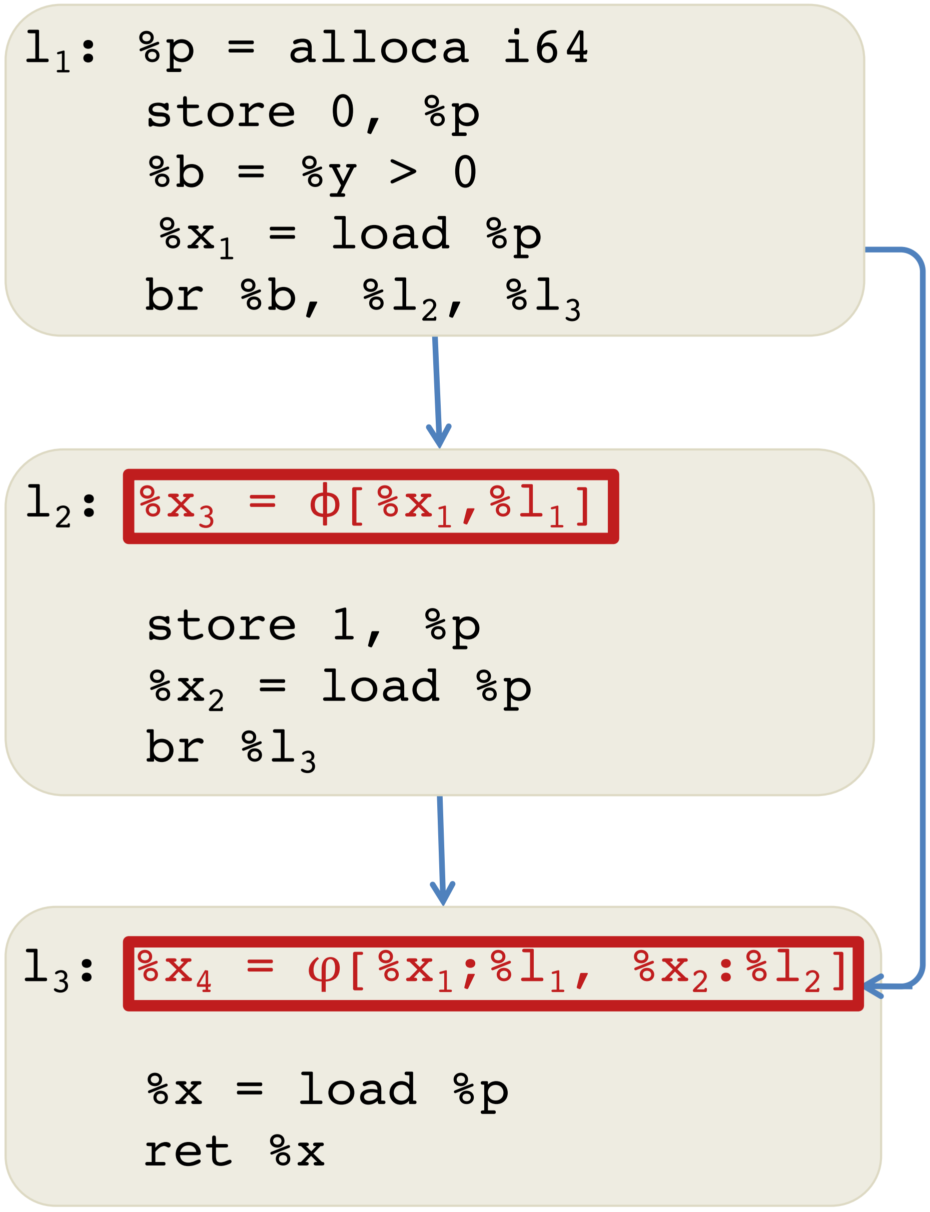


```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3

l2:

    store 1, %p
    %x2 = load %p
    br %l3

l3:

    %x = load %p
    ret %x
```

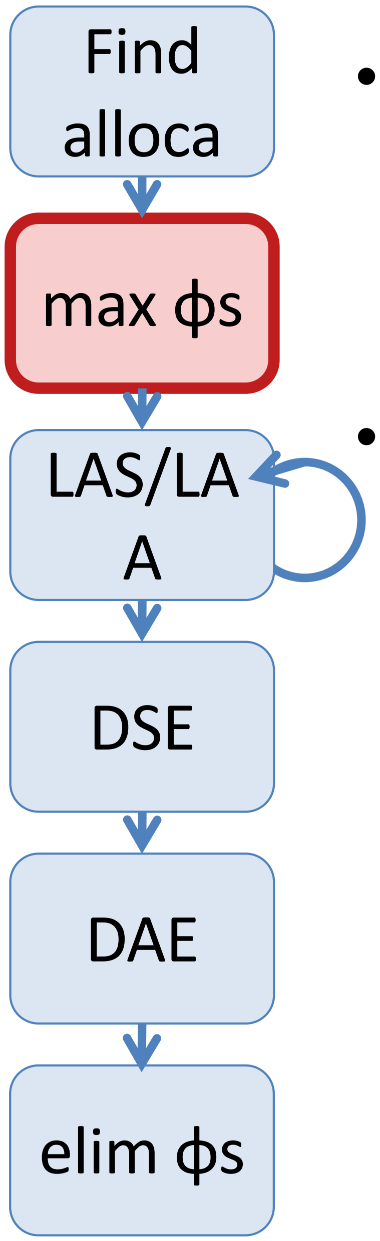


- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

Example SSA Optimizations

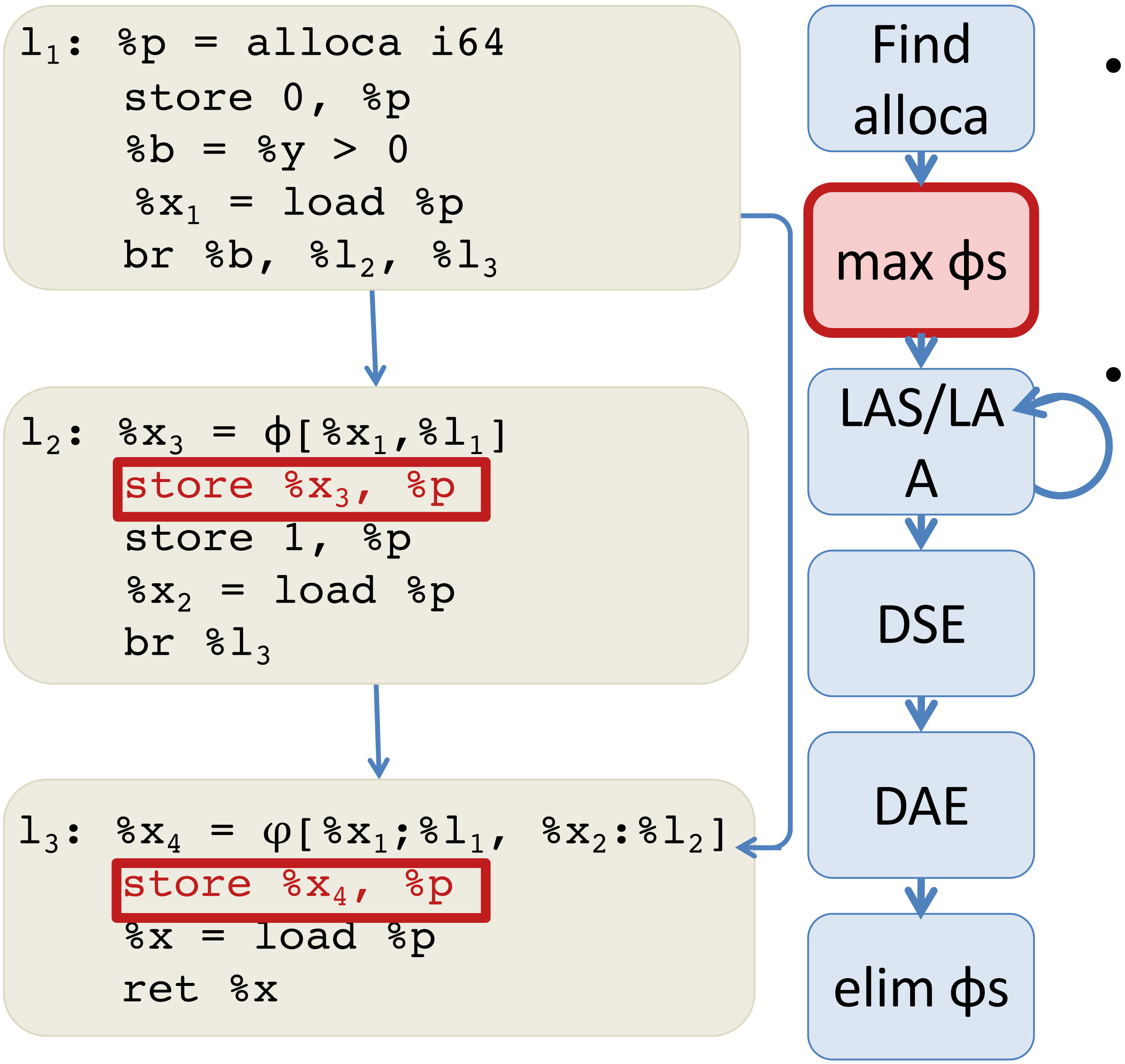


```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3

l2: %x3 = ϕ[%x1,%l1]

    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[%x1;%l1, %x2:%l2]

    %x = load %p
    ret %x
```



- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block

Example SSA Optimizations



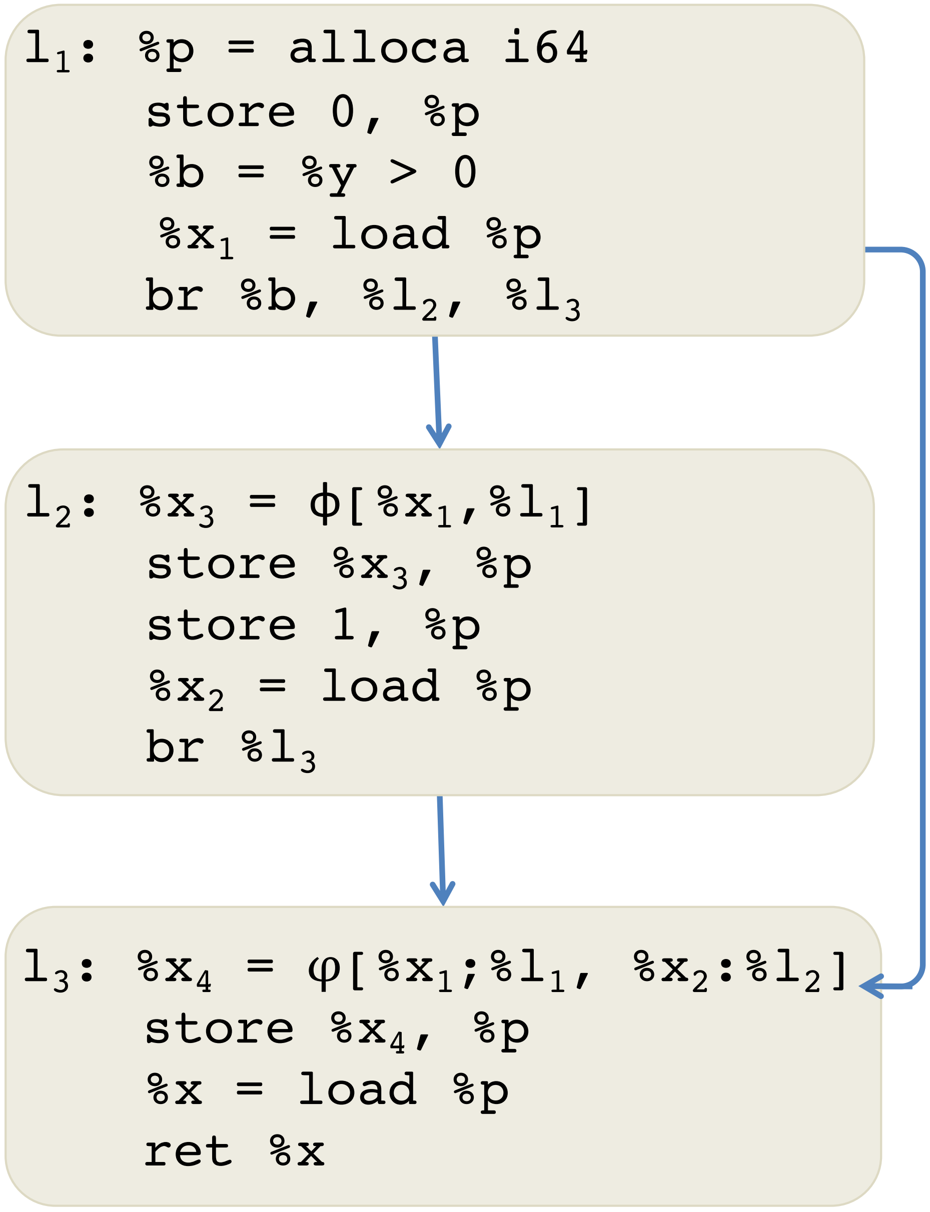
```
l₁: %p = alloca i64
    store 0, %p
    %b = %y > 0
     %x₁ = load %p
    br %b, %l₂, %l₃
```

```
l₂: %x₃ = ϕ[%x₁,%l₁]
    store %x₃, %p
    store 1, %p
    %x₂ = load %p
    br %l₃
```

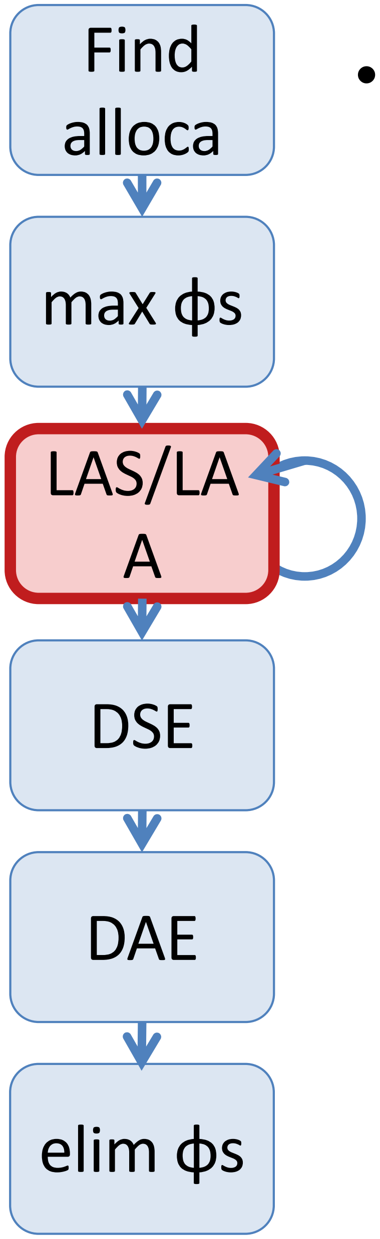
```
l₃: %x₄ = φ[%x₁;%l₁, %x₂:%l₂]
    store %x₄, %p
    %x = load %p
    ret %x
```

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block
  - Insert stores after φ-nodes

Example SSA Optimizations

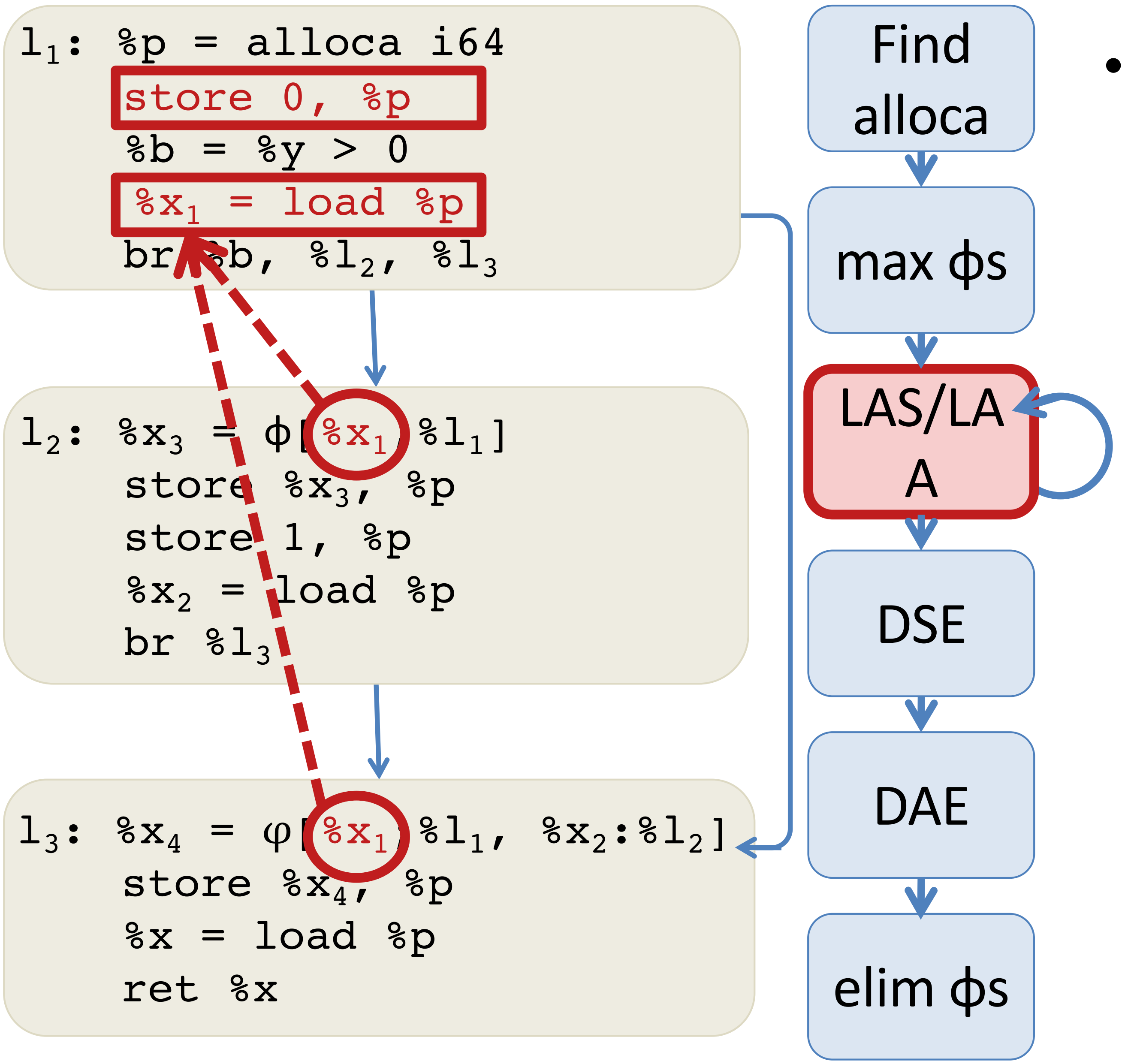


```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
     %x1 = load %p
    br %b, %l2, %l3

l2: %x3 = ϕ[%x1,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[%x1;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

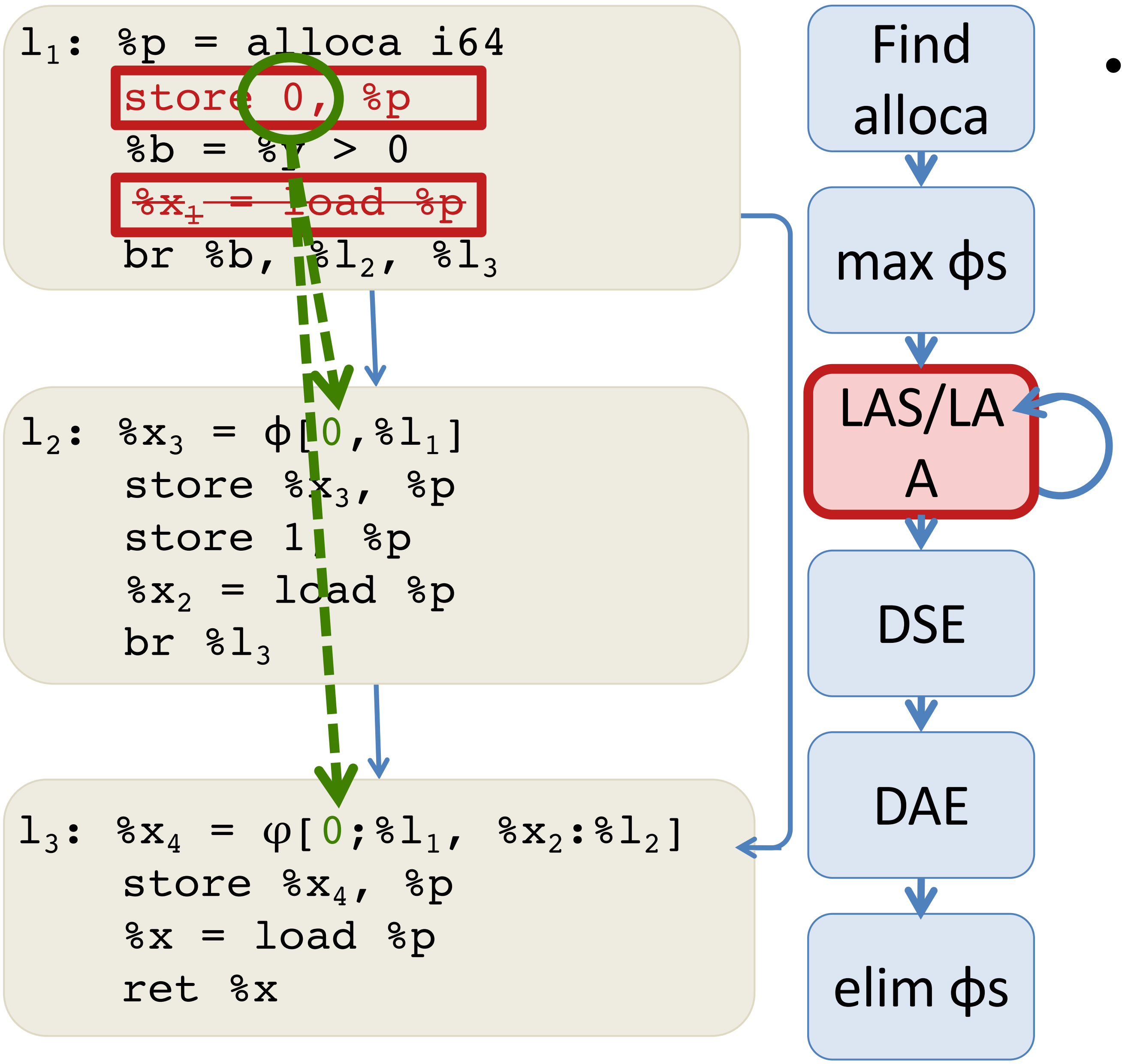
Example SSA Optimizations



```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3

l2: %x3 = ϕ[%x1,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[%x1,%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

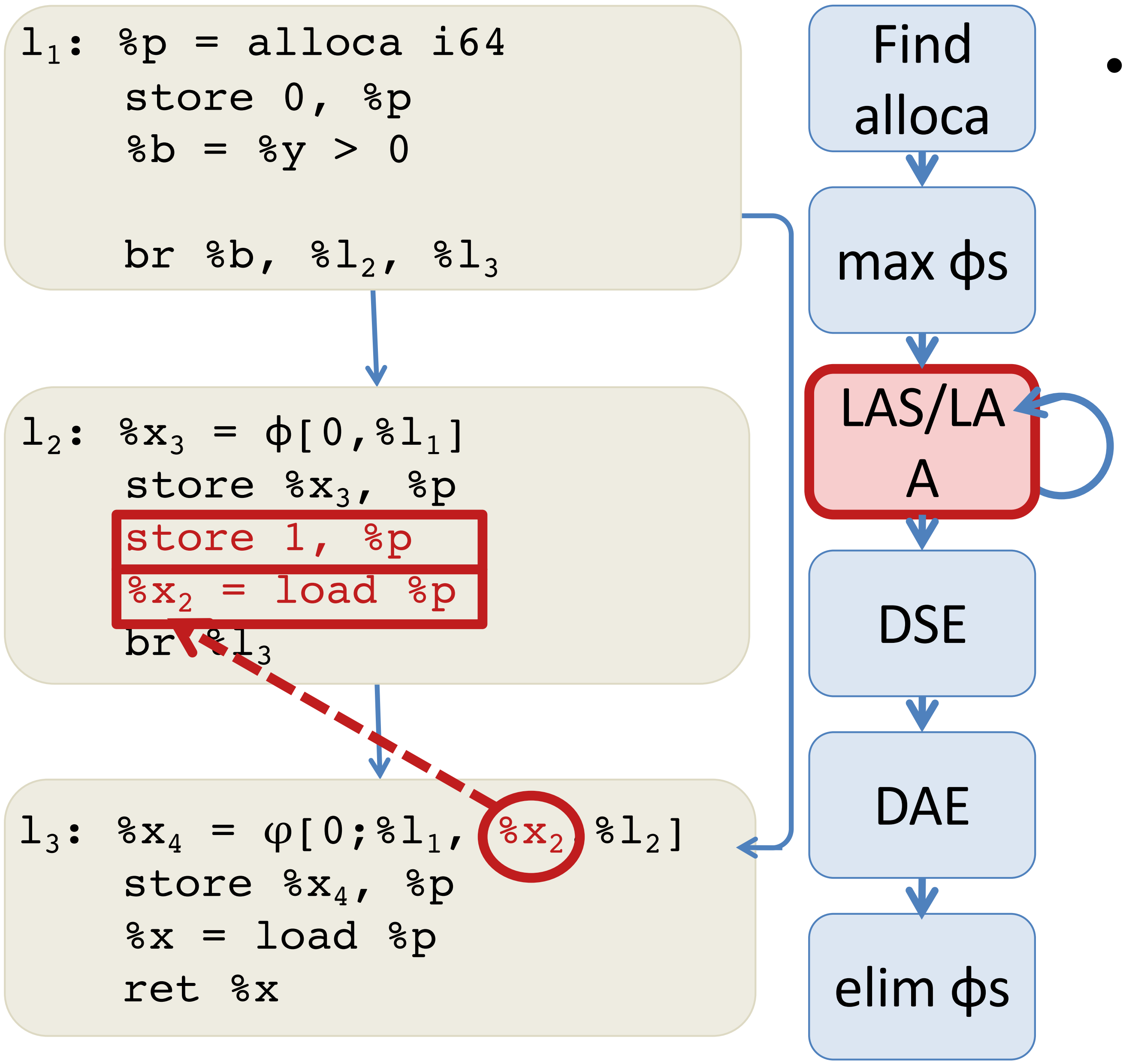
Example SSA Optimizations



```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    ~~%x1 = load %p~~
    br %b, %l2, %l3

l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[0;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



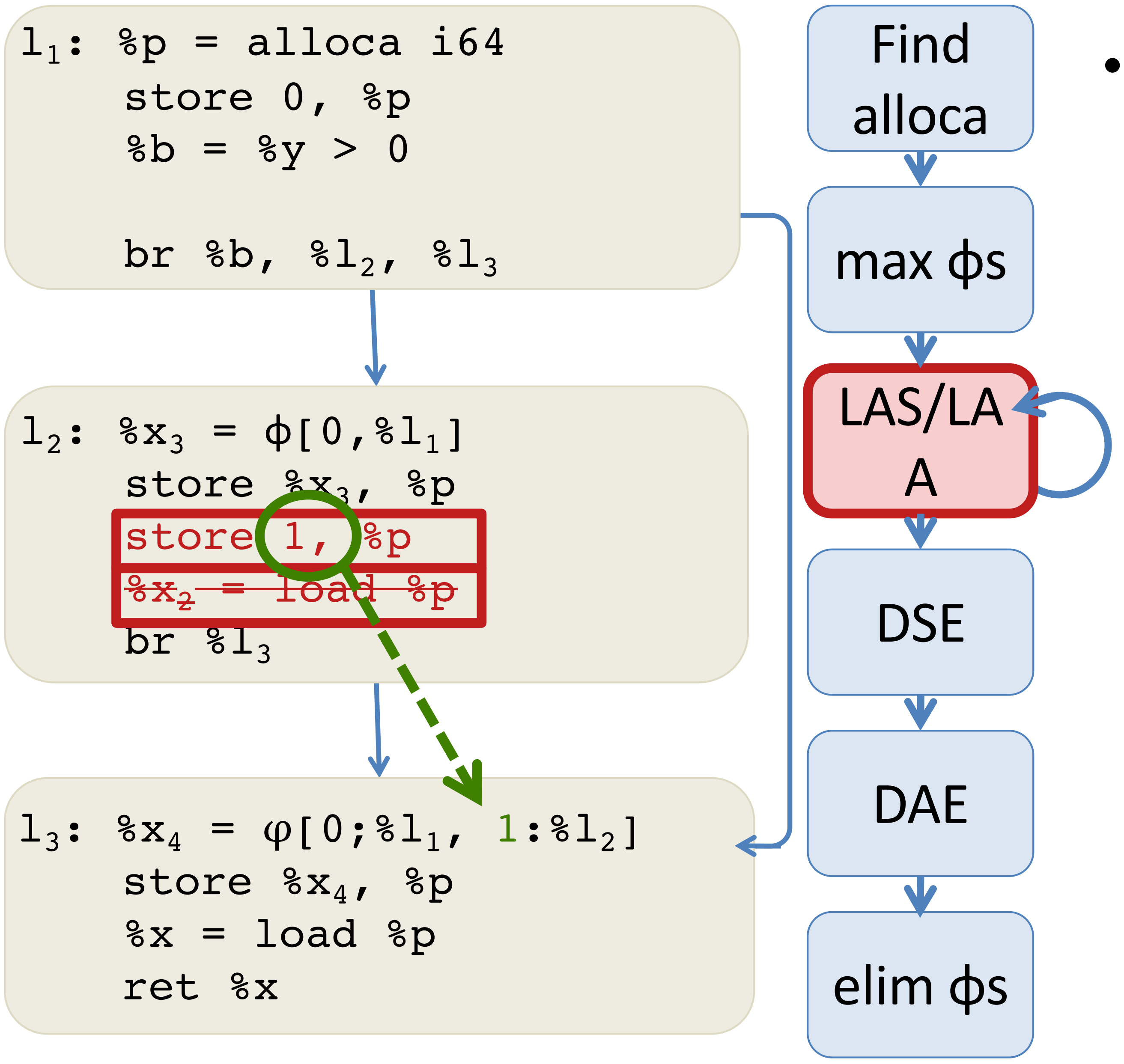
```
l₁: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l₂, %l₃
```

```
l₂: %x₃ = ϕ[0,%l₁]
    store %x₃, %p
    store 1, %p
    %x₂ = load %p
    br %l₃
```

```
l₃: %x₄ = φ[0;%l₁, %x₂, %l₂]
    store %x₄, %p
    %x = load %p
    ret %x
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



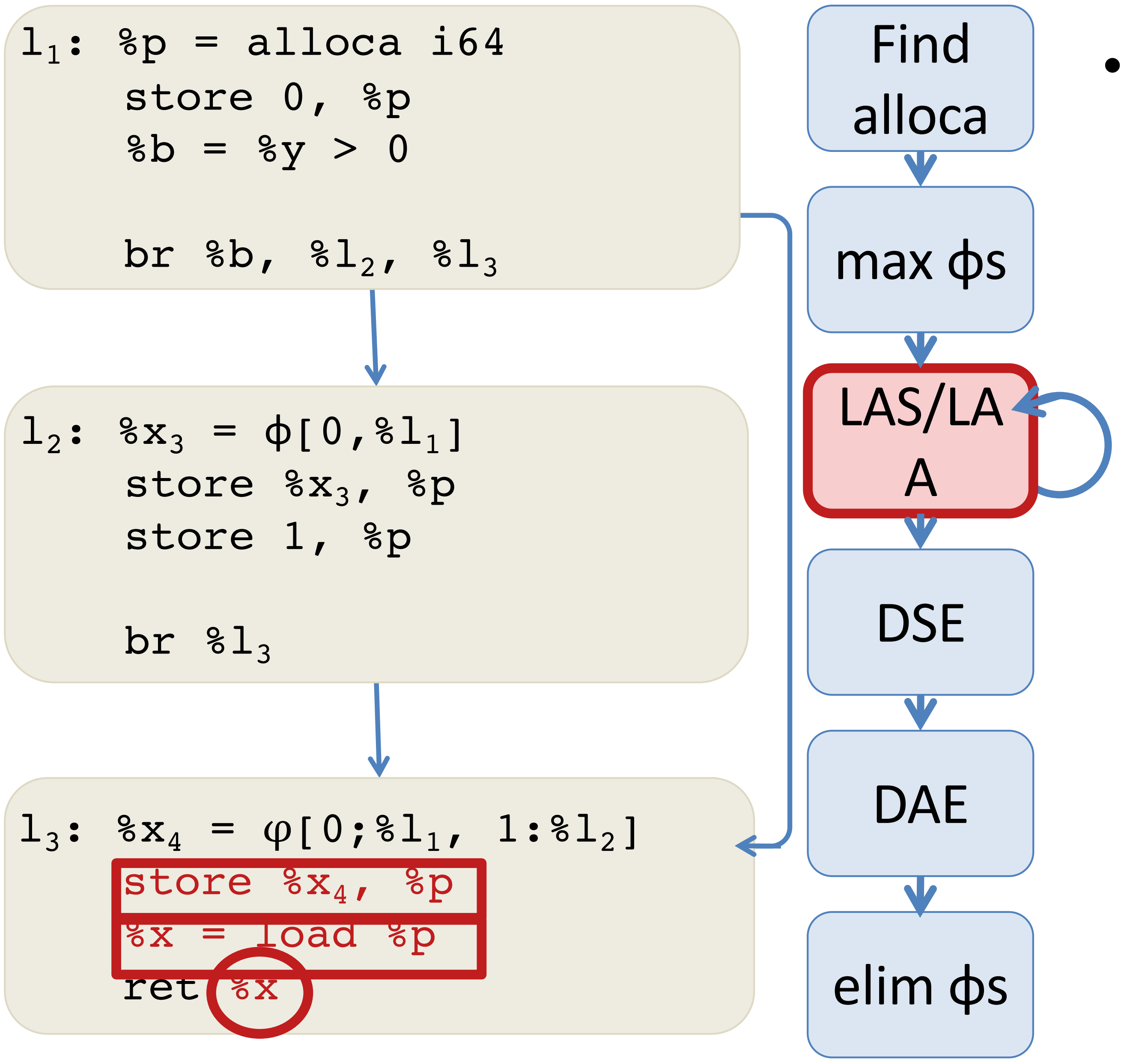
```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    ~~%x2 = load %p~~
    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



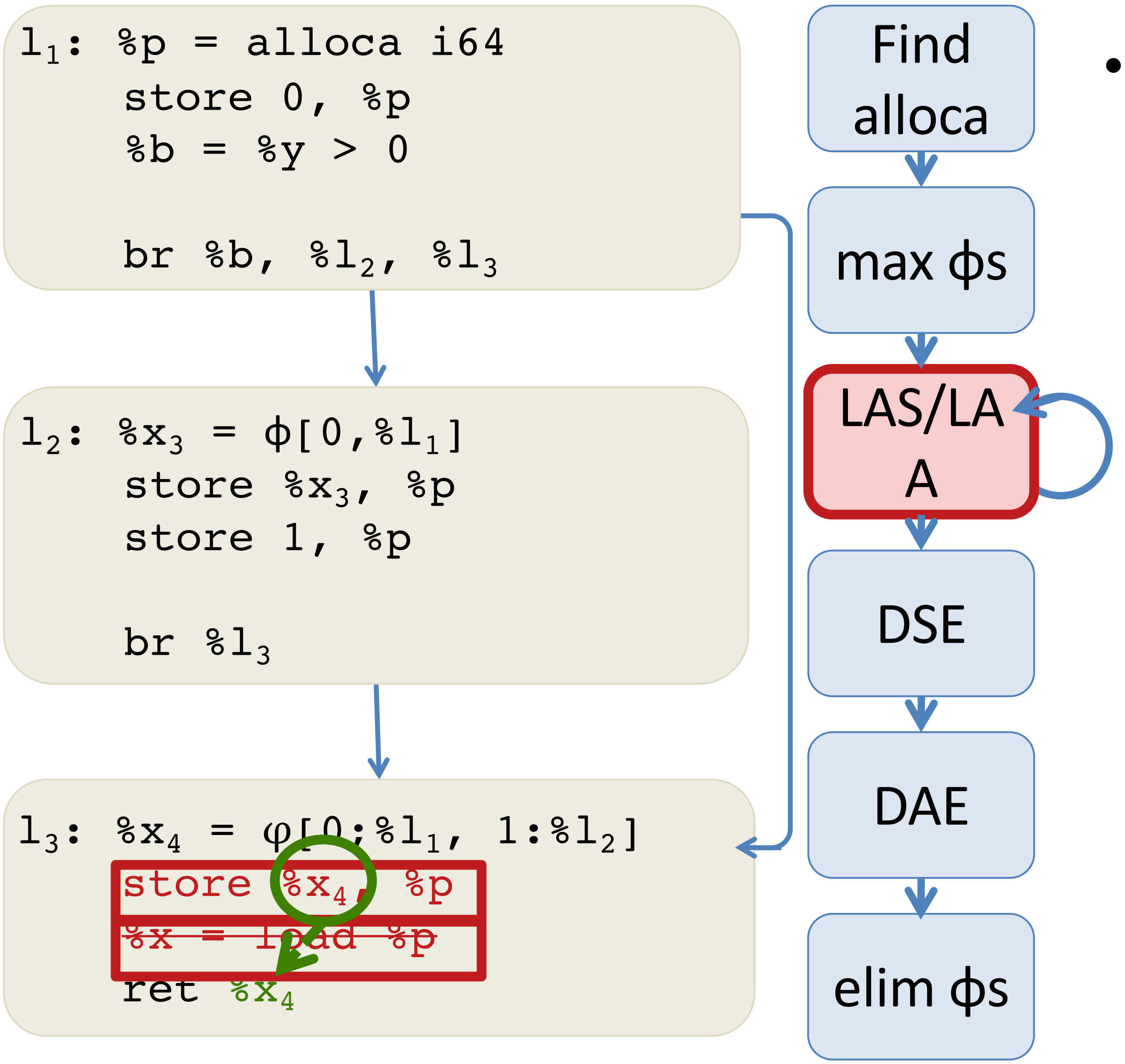
```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



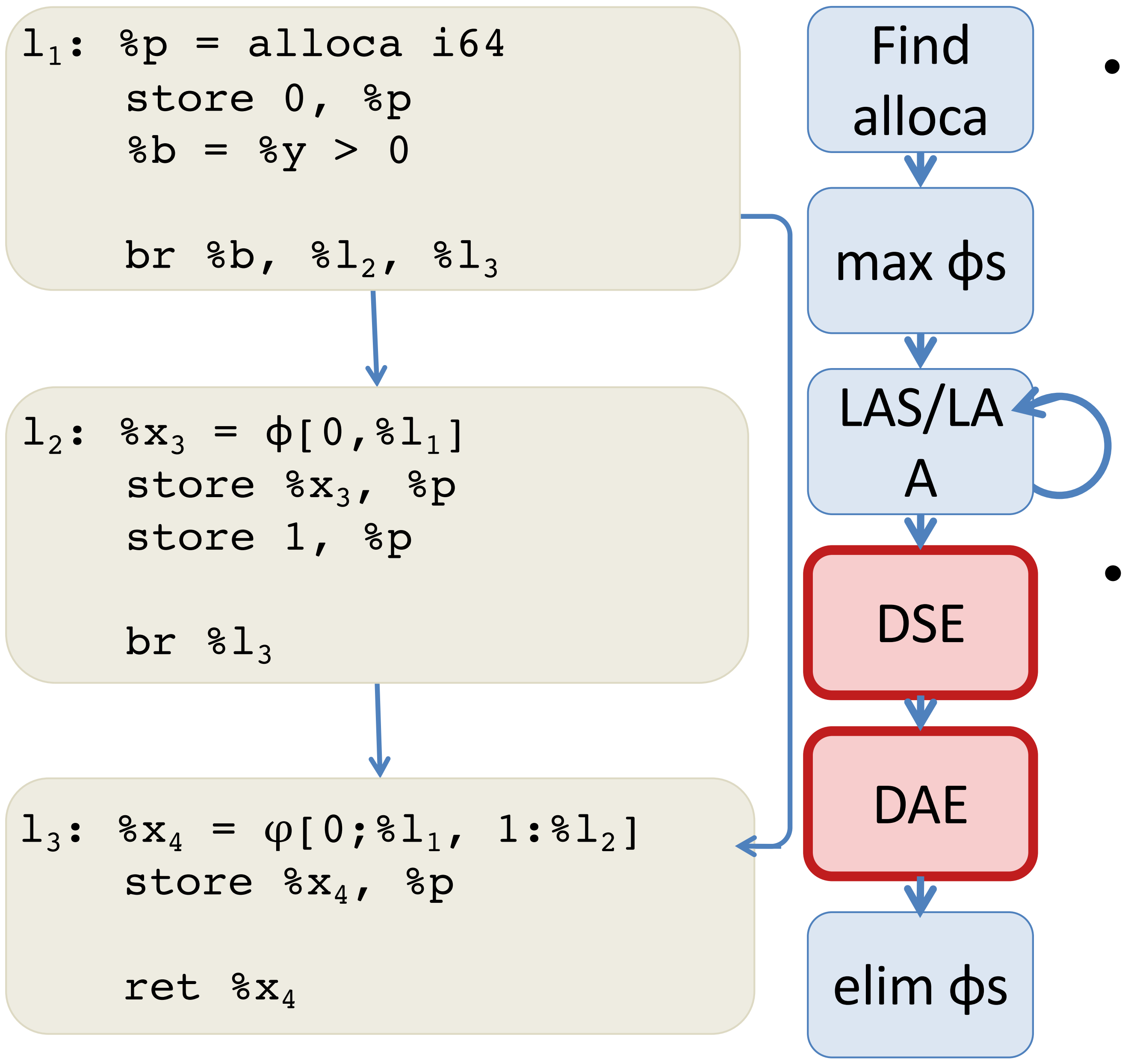
```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0:%l1, 1:%l2]
    store %x4, %p
    ~~%x = load %p~~
    ret %x4
```

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations



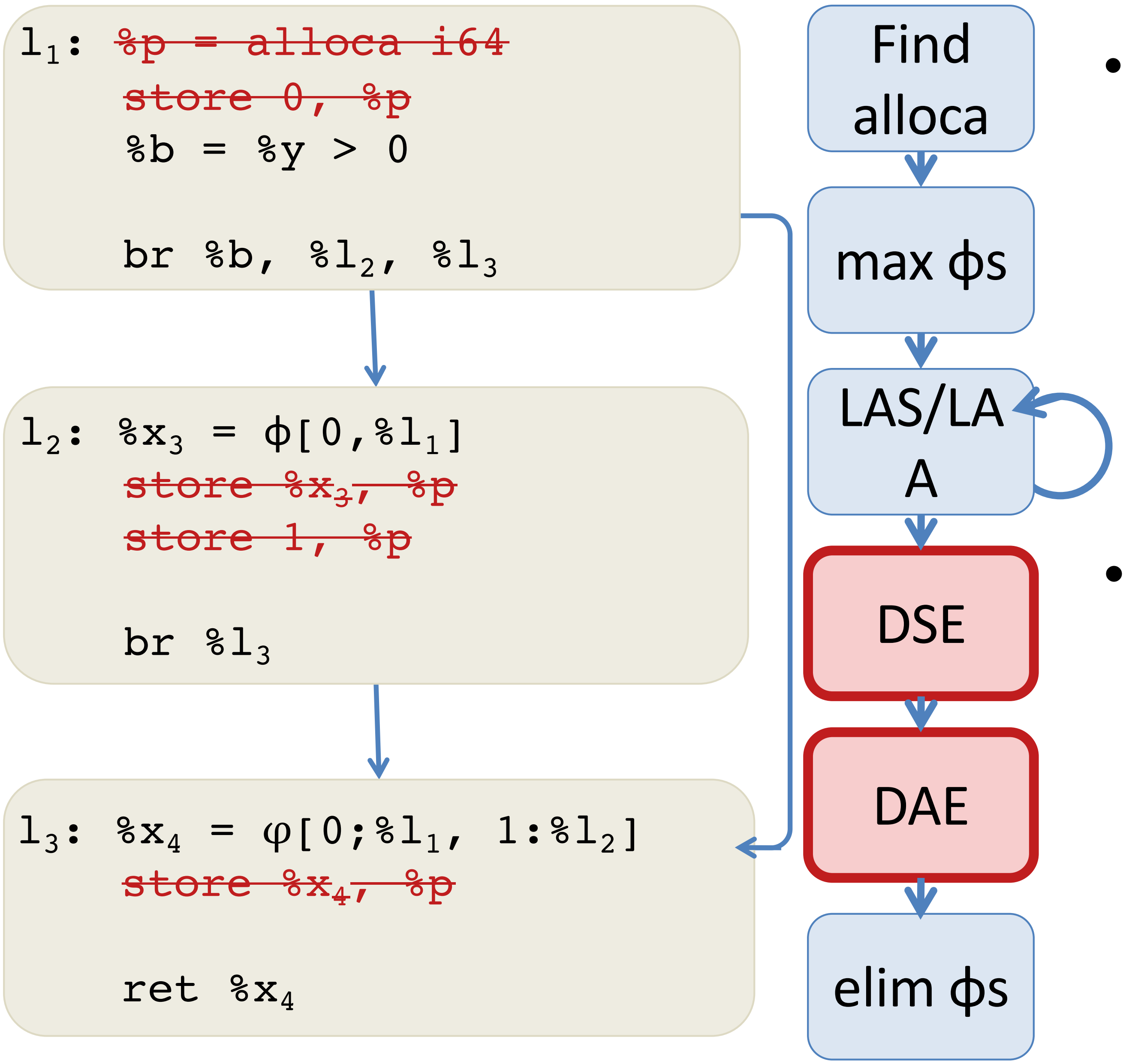
```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p

    ret %x4
```

- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

Example SSA Optimizations



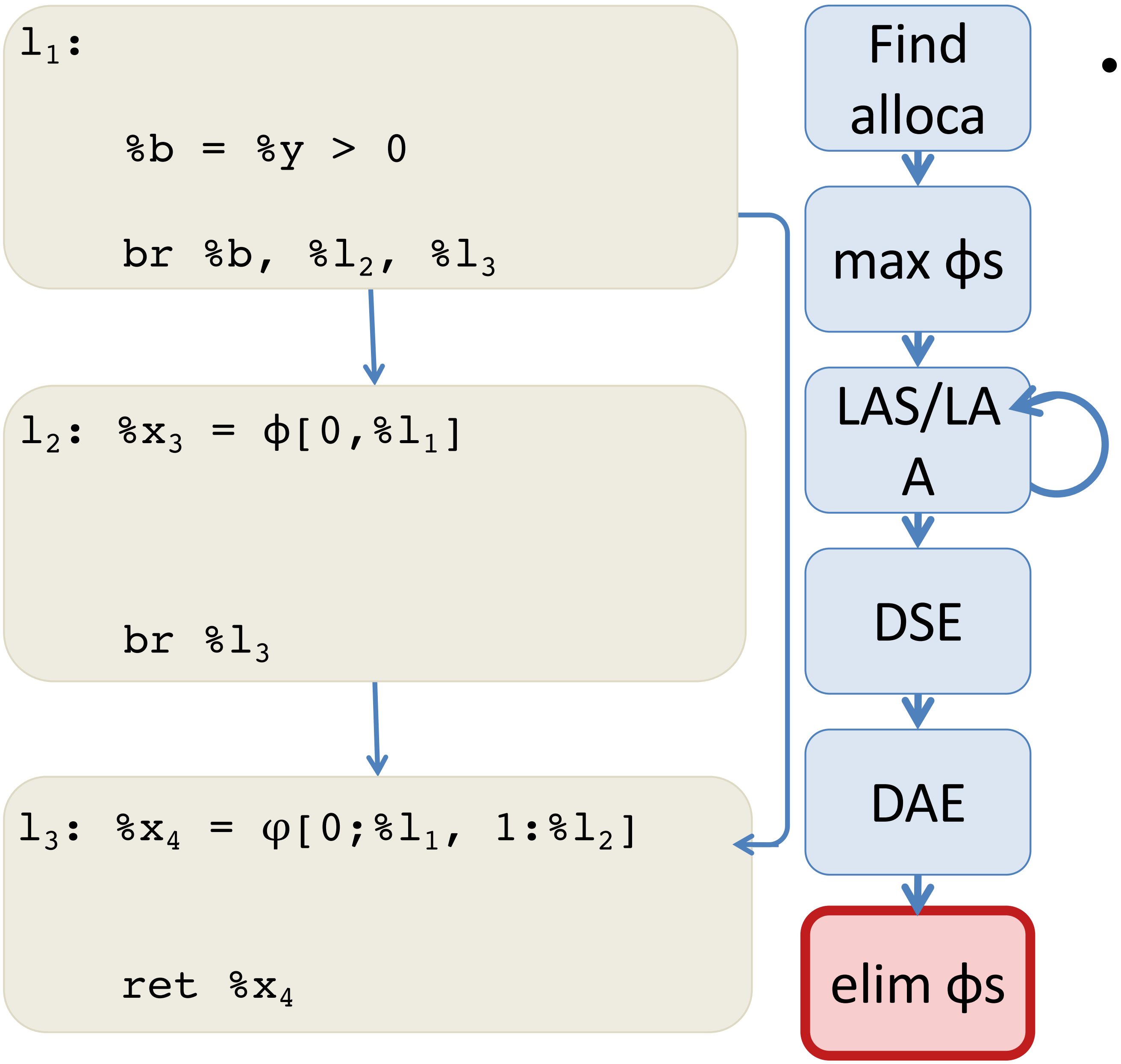
```
l₁: ~~%p = alloca i64~~
    ~~store 0, %p~~
    %b = %y > 0

    br %b, %l₂, %l₃
```

```
l₂: %x₃ = ϕ[0,%l₁]
    ~~store %x₃, %p~~
    ~~store 1, %p~~

    br %l₃
```

```
l₃: %x₄ = φ[0;%l₁, 1:%l₂]
    ~~store %x₄, %p~~

    ret %x₄
```

- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.

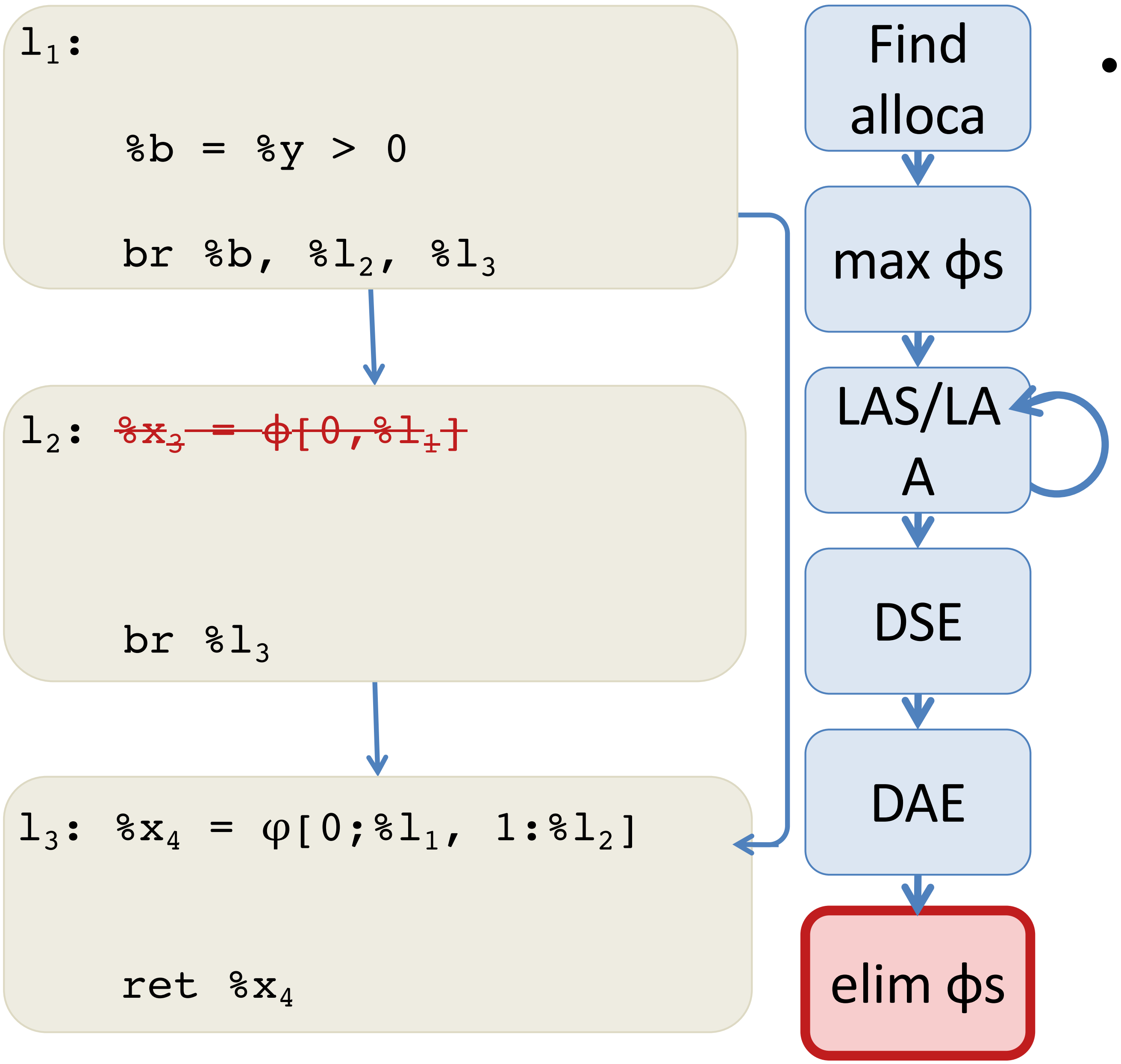
Example SSA Optimizations



```
l1:
    %b = %y > 0
    br %b, %l2, %l3

l2: %x3 = ϕ[0,%l1]
    br %l3

l3: %x4 = φ[0;%l1, 1:%l2]
    ret %x4
```

- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002

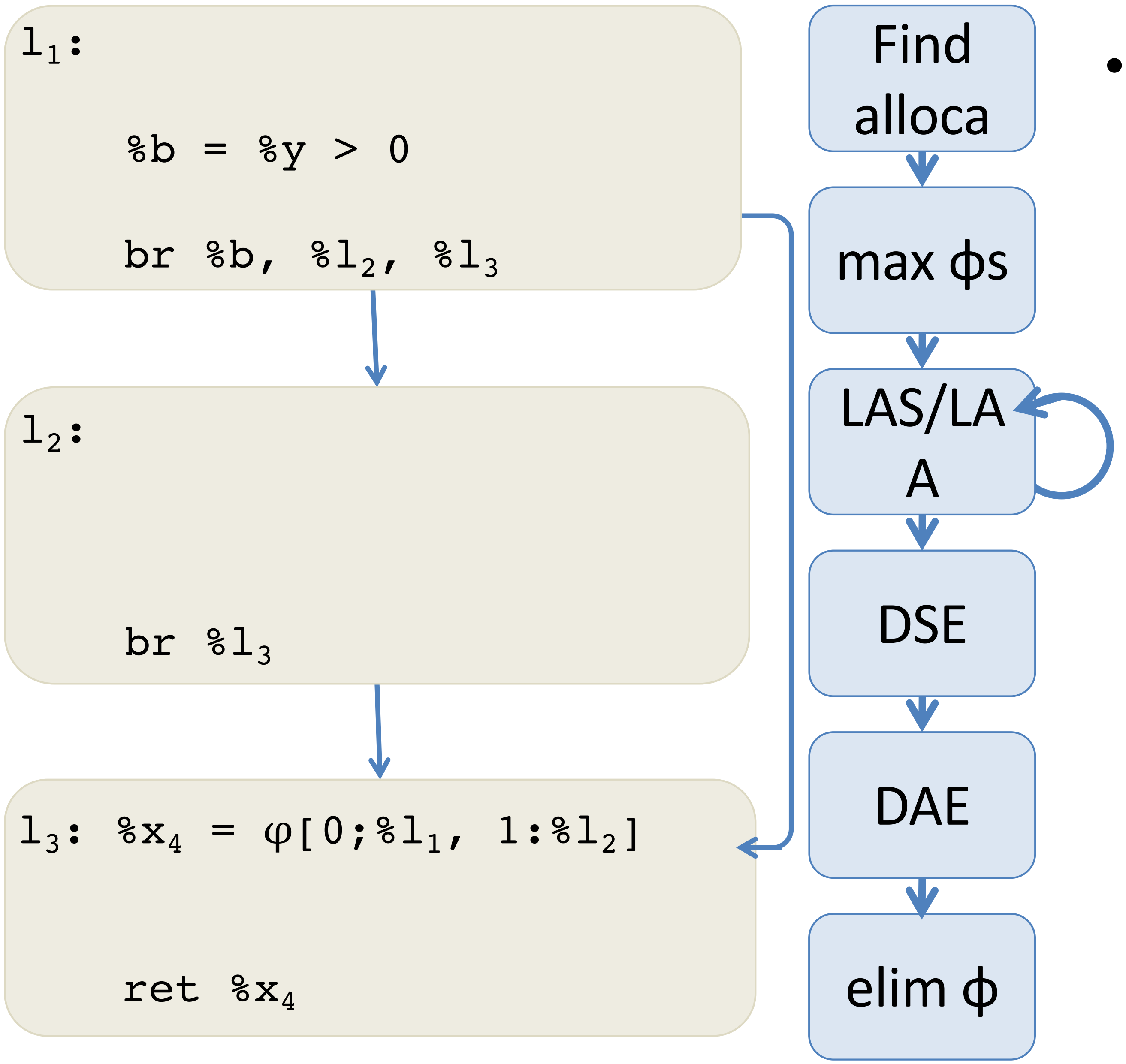
Example SSA Optimizations



```
l_1:
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: ~~%x_3 = ϕ[0,%l_1]~~

    br %l_3

l_3: %x_4 = φ[0;%l_1, 1:%l_2]

    ret %x_4
```

- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor

Example SSA Optimizations



```
l₁:
    %b = %y > 0
    br %b, %l₂, %l₃
```

```
l₂:



    br %l₃
```

```
l₃: %x₄ = φ[0;%l₁, 1:%l₂]

    ret %x₄
```

- Done!

LLVM Phi Placement

- This transformation is also sometimes called register promotion
  - older versions of LLVM called this “mem2reg” memory to register promotion
- In practice, LLVM combines this transformation with *scalar replacement of aggregates* (SROA)
  - i.e. transforming loads/stores of structured data into loads/stores on register-sized data
- These algorithms are (one reason) why LLVM IR allows annotation of predecessor information in the .ll files
  - Simplifies computing the DF