



# Compilers I

---

Loop Optimizations

Dr William Harrison



# Loop Optimizations

---


- Empirical studies show that much of program execution occurs within loops
  - So, it makes sense to identify loops & concentrate optimization efforts on that code
  - But what is a loop?
    - Within source code, it's obvious
    - Within IR/CFG, however,...
      - Loops aren't precisely what your first guess would be



# Simple loop optimization

---

within the source code, that is...

`for (i = 1; i++ ; i = 10) { c }`        `c ; ... ; c`



10 times

Caveat:

- seems reasonable (removes branches)
- not terribly general: would like to use on while loops but how do you identify the induction variable “i”?
- also, how does this apply at the IR/CFG level?

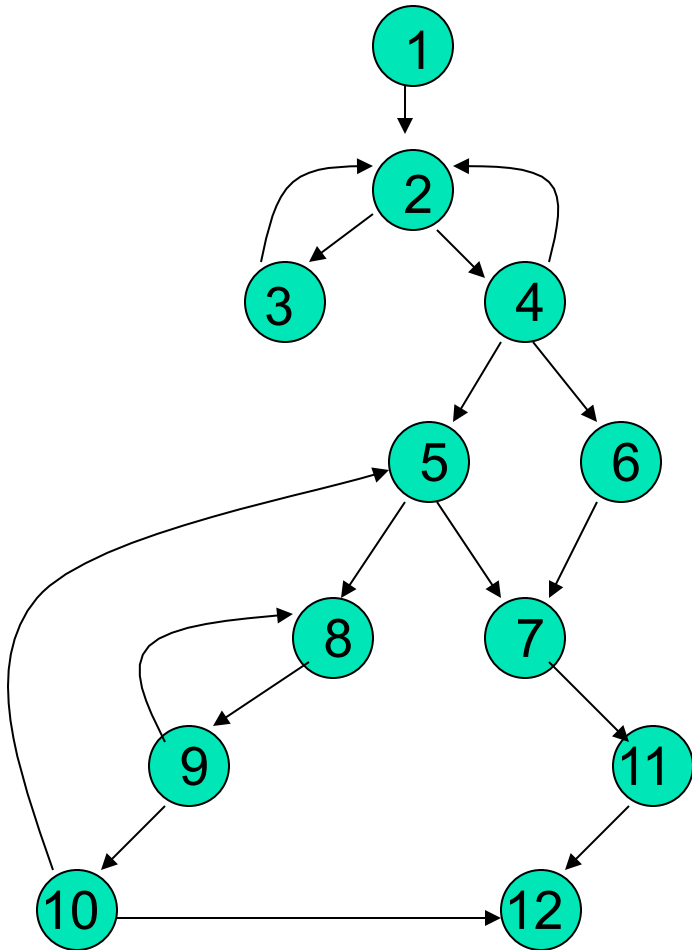


# Ex: Loop invariant code hoisting

---

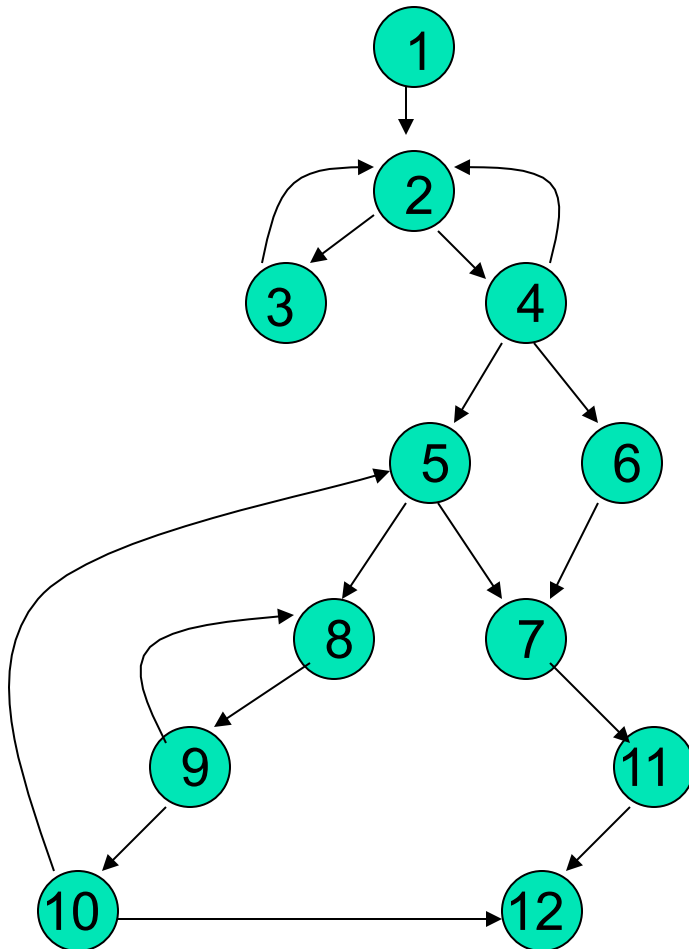
- Say the assignment “ $a \leftarrow b+c$ ” occurs within a loop
  - But “b” and “c” aren't assigned in the loop
  - Would like to move “ $a \leftarrow b+c$ ” somewhere “right in front of” the loop
    - Thereby avoiding redundant reassignments to “a”
- Problem IR/CFG is a directed graph of basic blocks
  - Where, for example, is the “front of” a loop?

# A flow graph



Say “ $a \leftarrow b+c$ ” occurs in 9,  
To where might we hoist it?

# A flow graph



Say “ $a \leftarrow b+c$ ” occurs in 9,  
To where might we hoist it?

- Nodes 5,4,2,1 seem likely
  - how would I determine that automatically?
  - Impact on other optimizations?
  - Identifying loops with “dominator tree”

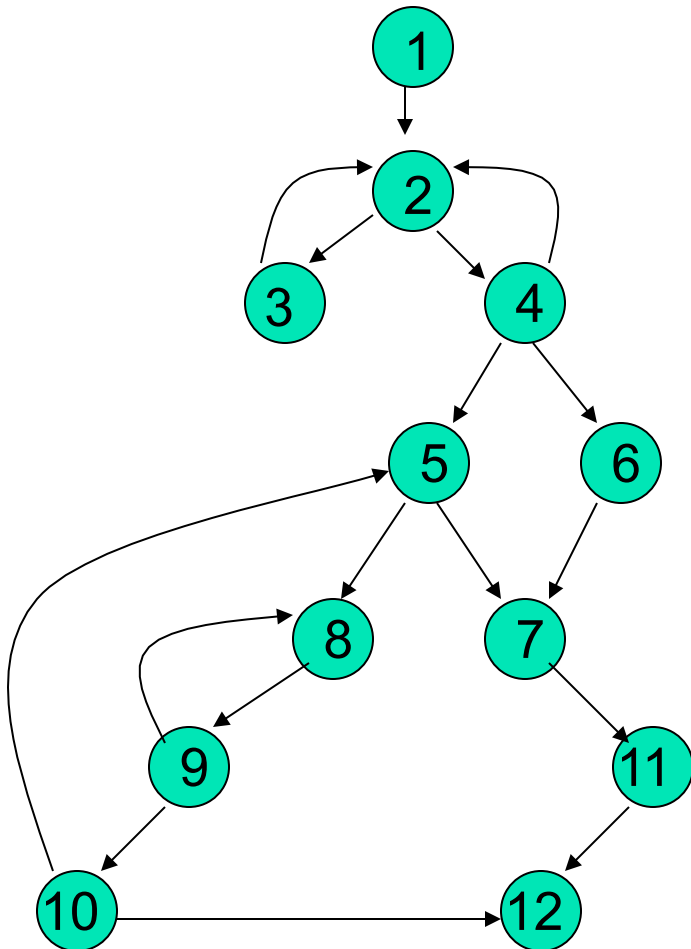


# Dominators

---

- Assume we have a flow graph with entry node  $s_0$  with no predecessors
  - I.e., no edge into  $s_0$
- Node  $d$  **dominates**  $n$  means that
  - $d$  occurs in every path from  $s_0$  to  $n$
- Note that every node dominates itself

# A flow graph



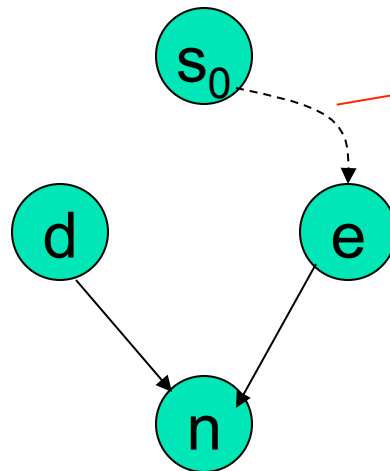
- for each pair of nodes  $d, n$ 
  - does  $d$  dominate  $n$ ?
  - can think of “ $d$  dominates  $n$ ” as a new kind of directed arc in a new graph
- **Question:** is it possible that for  $d \neq n$  that:
  - $d$  dominates  $n$ , and
  - $n$  dominates  $d$ ?



# Theorem

Let  $d \neq e$  both dominate  $n$ . Then, either  $d$  dominates  $e$  or  $e$  dominates  $d$  (but not both)

**Proof sketch.** Assume neither  $d$  dominates  $e$  nor vice versa.

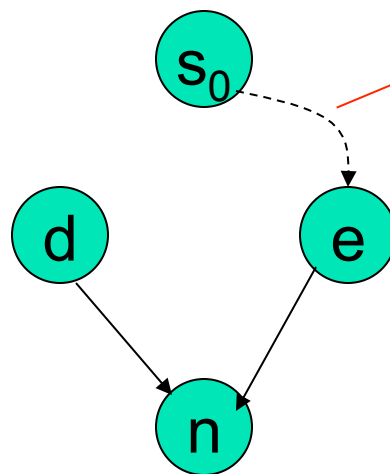


Path to  $e$  not containing  $d$  exists since  $d$  doesn't dominate  $e$

# Theorem

Let  $d \neq e$  both dominate  $n$ . Then, either  $d$  dominates  $e$  or  $e$  dominates  $d$  (but not both)

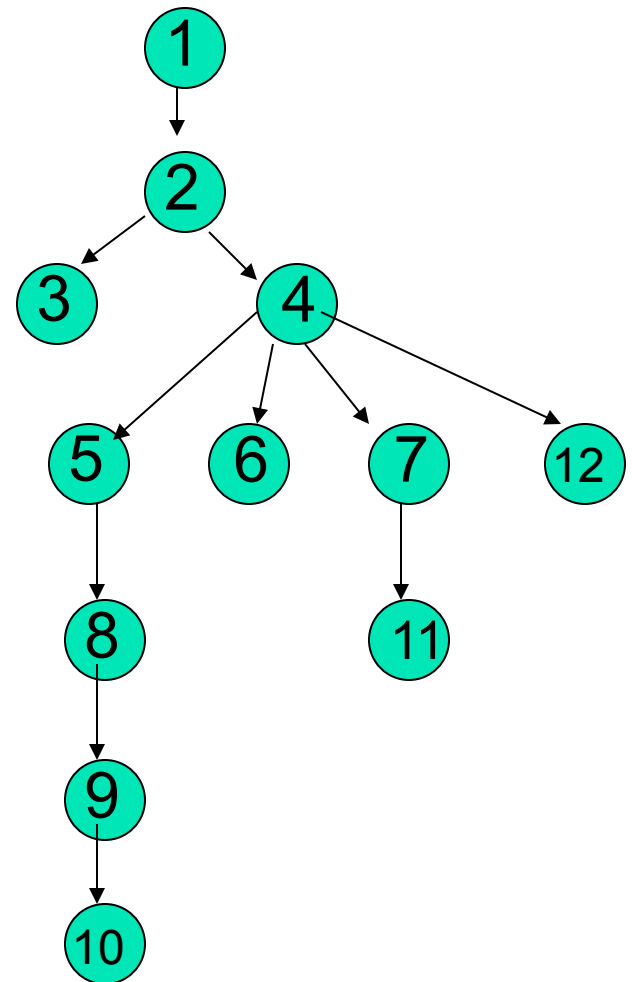
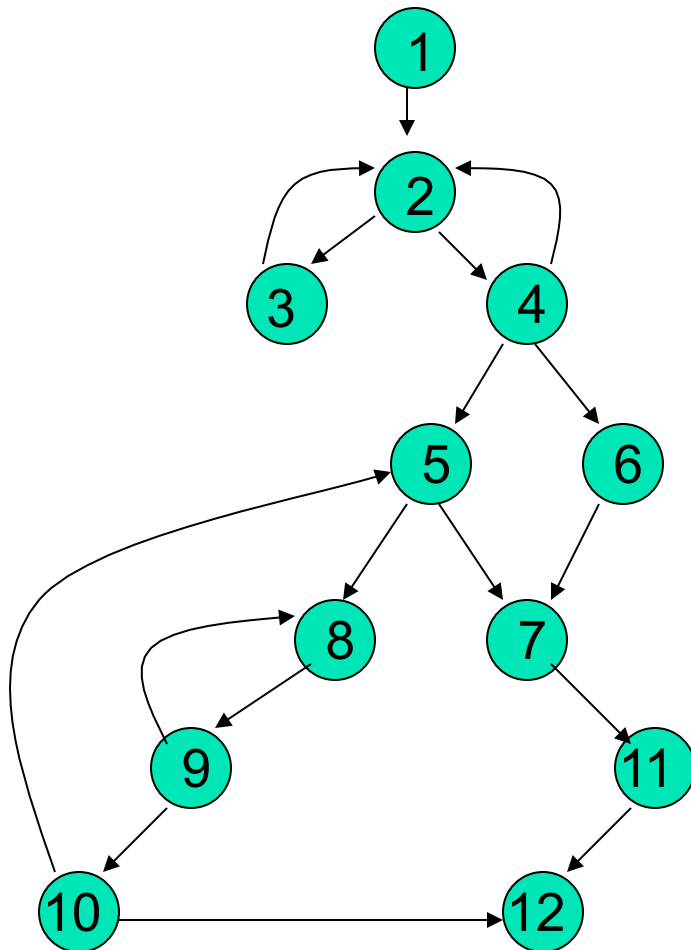
**Proof sketch.** Assume neither  $d$  dominates  $e$  nor vice versa.



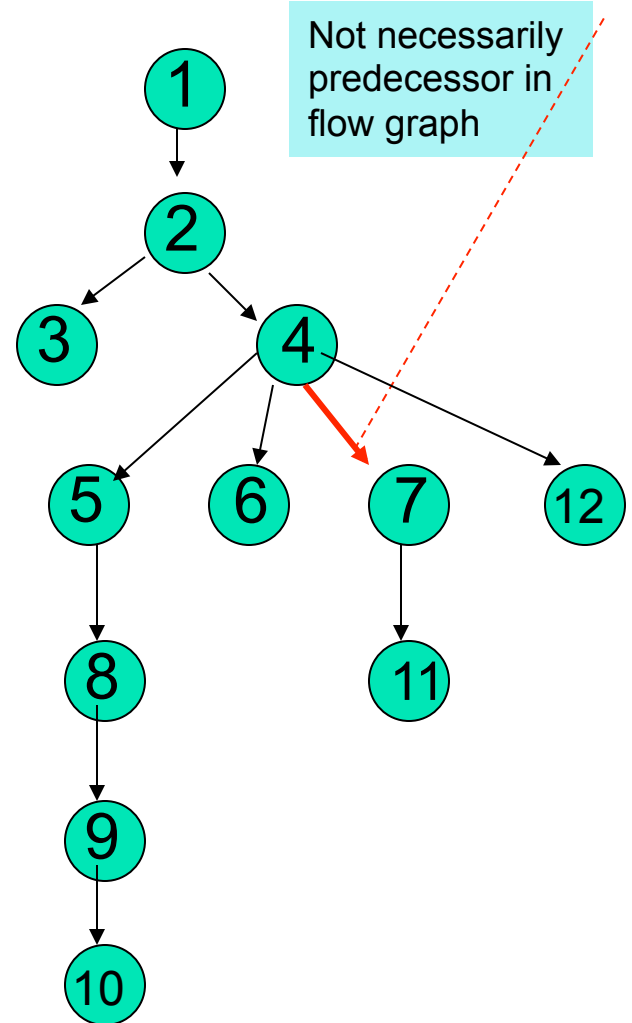
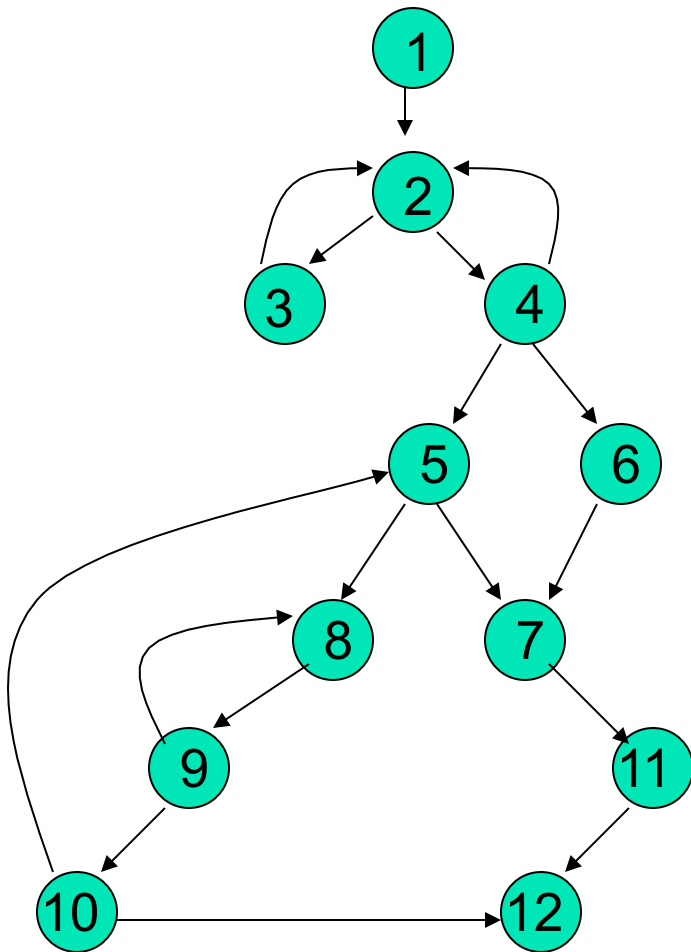
Path to  $e$  not containing  $d$  exists since  $d$  doesn't dominate  $e$

Path from  $s_0$  to  $n$  not through  $d$  exists.  $\therefore d$  doesn't dominate  $n$   $\Rightarrow \Leftarrow$

# flow graph & its dominator tree



# flow graph & its dominator tree





# Immediate Dominators

---

- An **immediate dominator** of node  $n$  is a node  $\text{idom}(n)$  such that
  - $\text{idom}(n) \neq n$
  - $\text{idom}(n)$  dominates  $n$
  - $\text{idom}(n)$  does not dominate any other dominator of  $n$
- Every node has (at most) one immediate dominator
  - How do we know this?



# Calculating Dominators

---

Let  $D[n]$  be the set of nodes that dominate  $n$  in a particular flow graph  $G$ , we get the following two simultaneous equations\*

$$D[s_0] = \{s_0\}$$

$$D[n] = \{n\} \cup \left( \bigcap_{p \in \text{pred}(n)} D[p] \right) \text{ for } n \neq s_0$$

\*I wonder how we'd solve equations like this?



# Iterative solution

---

change := true

$D[s_0] := \{s_0\}$

foreach  $n \in (\text{Nodes}(G) \setminus \{s_0\})$  {  $D[n] := \{n\}$  }

repeat

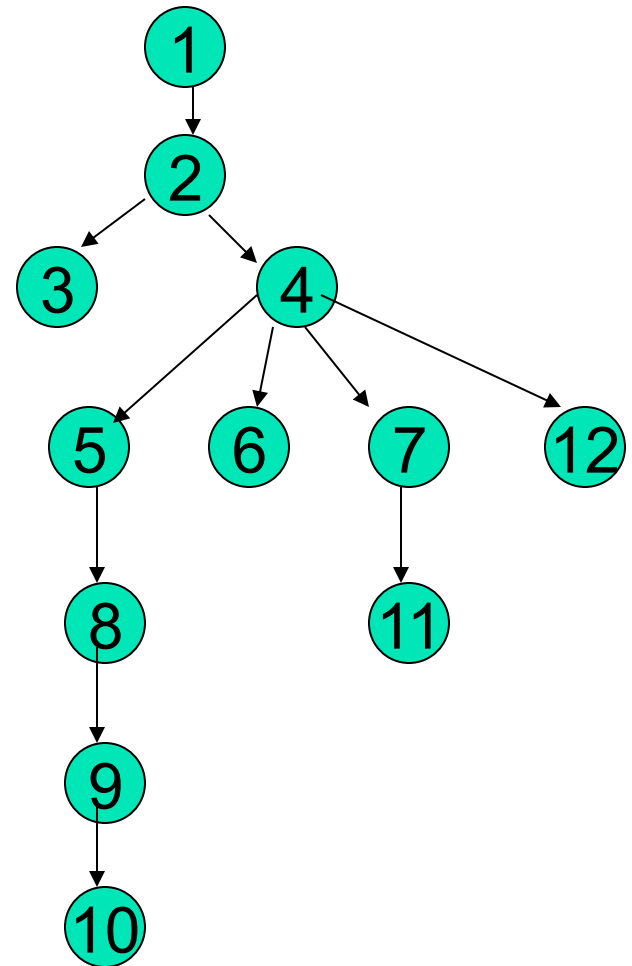
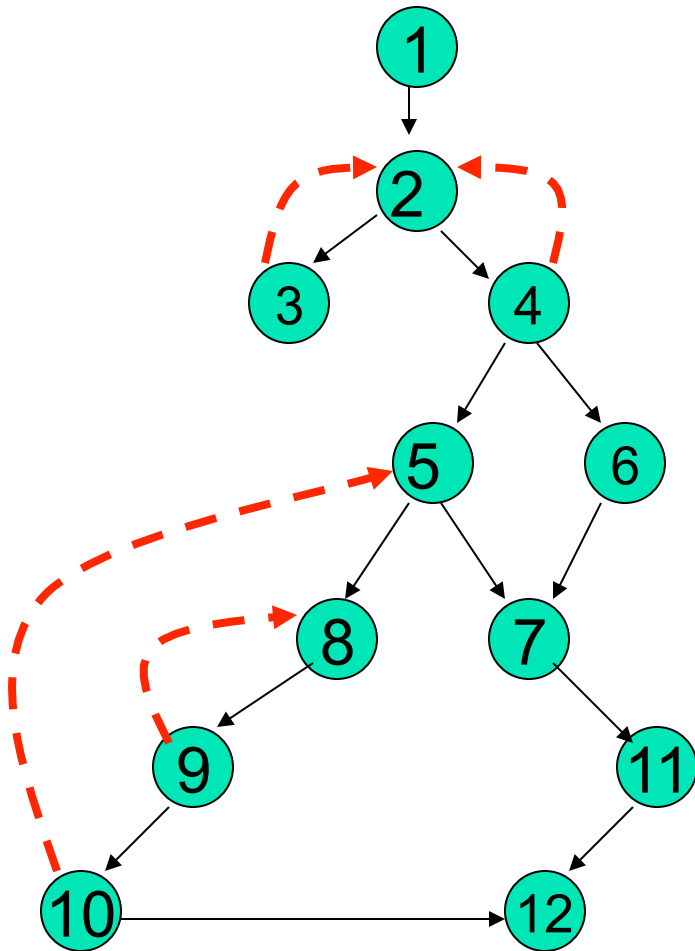
    change := false

    foreach  $n \in (\text{Nodes}(G) \setminus \{s_0\})$

$\left( \begin{array}{l} T := \text{Node} \\ \text{foreach } p \in (\text{pred}(n) \setminus \{s_0\}) \{ T := T \cap D[p] \} \\ X := \{n\} \cup T \\ \text{if } X \neq D[n] \text{ then} \\ \quad \text{change} := \text{true} \\ \quad D[n] = X \end{array} \right)$

until (not change)

**Backedges** are edges  $n \rightarrow h$  where  $h$  dominates  $n$





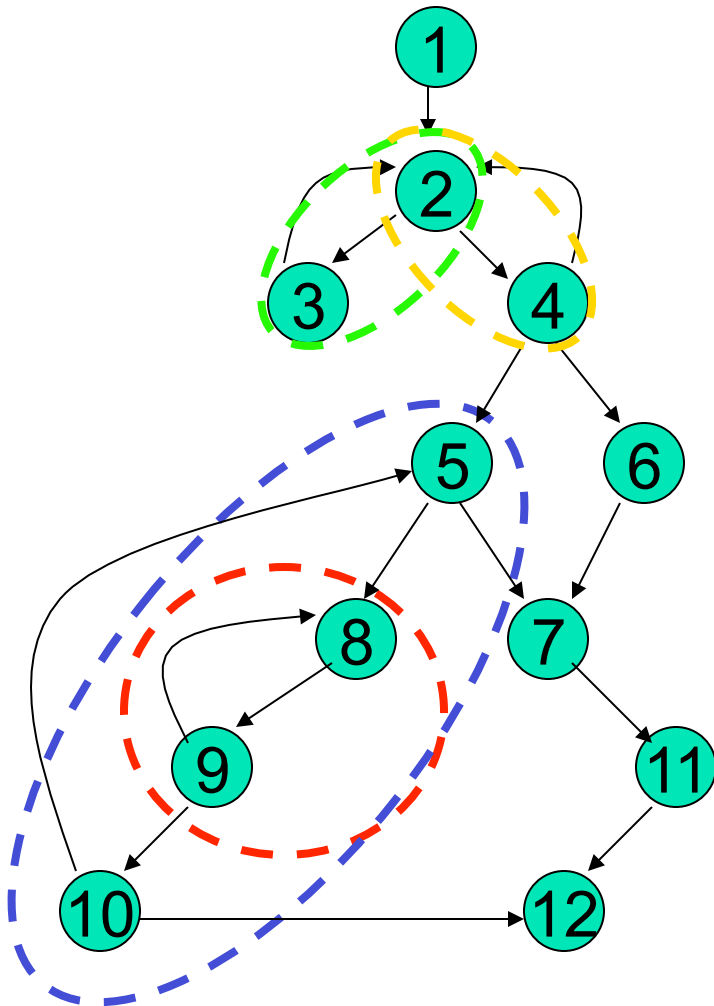


# “Natural” Loops

---

- The natural loop of a backedge  $n \rightarrow h$  is
  - The set of  $x$  such that
    - $h$  dominates  $x$
    - there is a path from  $x$  to  $n$  not containing  $h$
- $h$  is called the **header** of this loop

# Backedges induce natural loops



The four natural loops are:

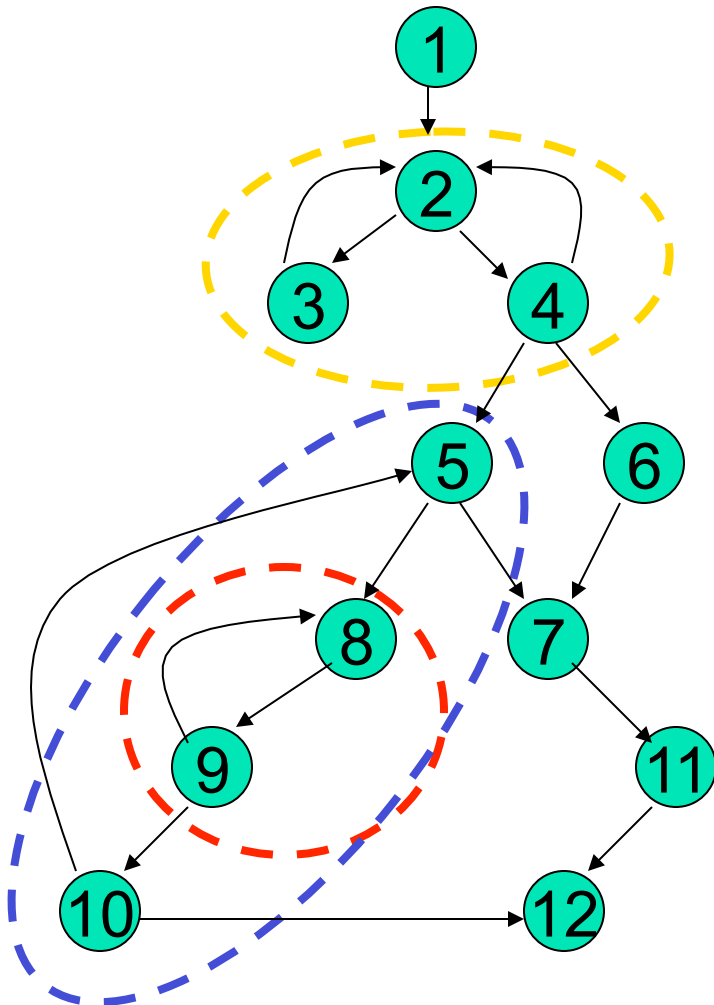
$\{2,3\}$

$\{2,4\}$

$\{5,8,9,10\}$

$\{8,9\}$

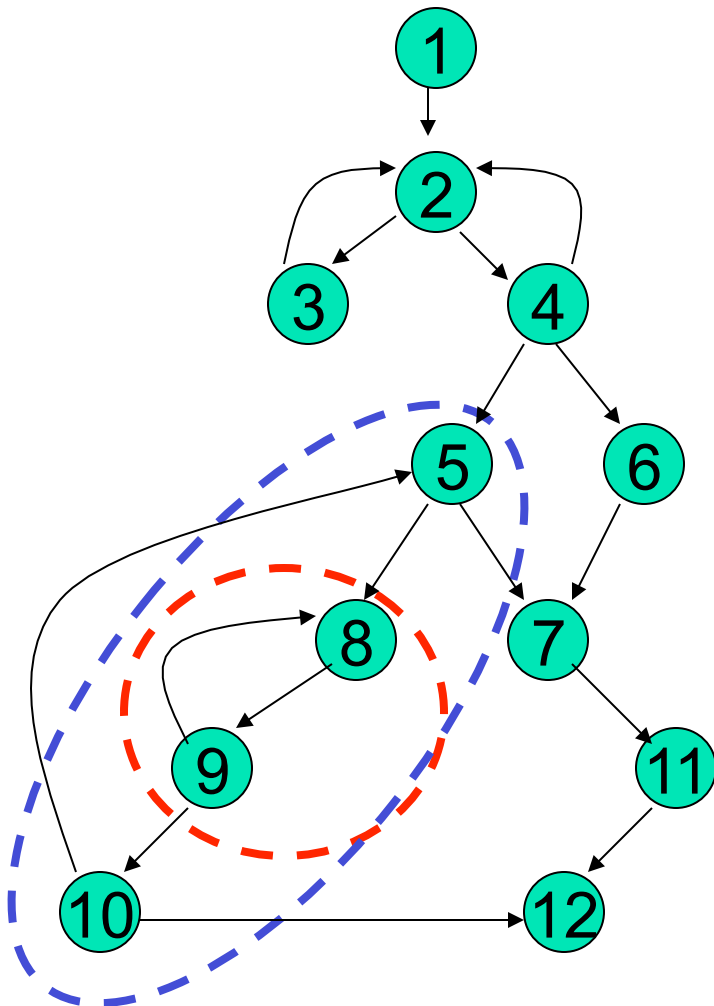
# Identifying loops



By merging natural loops with identical headers, one identifies all **loops**

- note that a natural loop is what you think of as a loop
- while a loop inside a compiler is a slightly different animal

# Identifying nested loops



**Defn:** Let  $A, B$  be loops with headers  $a, b$  s.t.  $a \neq b$ .  
 $B$  is **nested** within  $A$  iff  $B \subset A$

- generally start optimizing inside the innermost loop
- Ex:  $\{8, 9\}$  nested within  $\{5, 8, 9, 10\}$
- Reminder: “ $\subset$ ” means proper subset
  - i.e.,  $A \neq B$

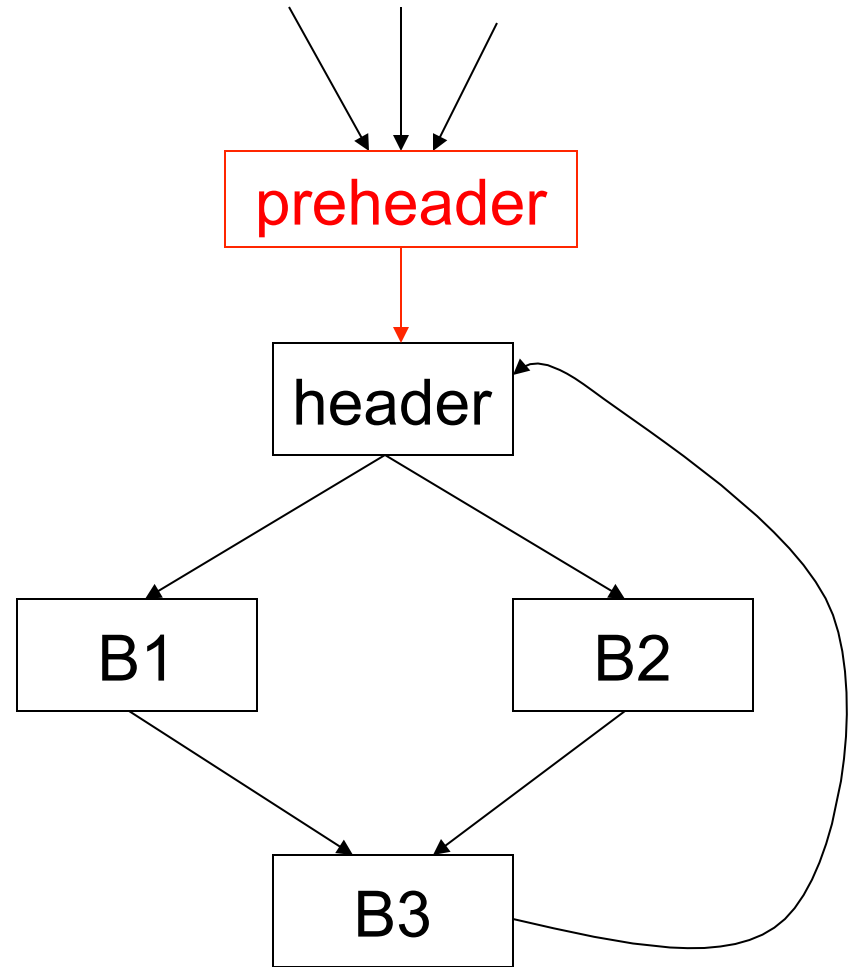
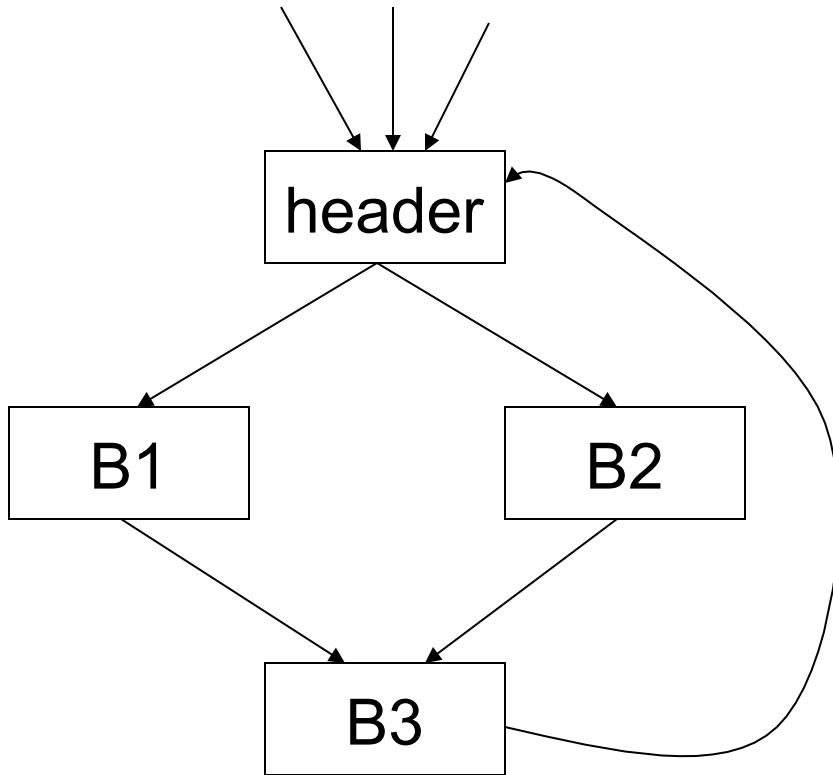


# Loop pre-header

---

- Many loop optimizations require moving code from inside a loop to just before its header
- To guarantee that we uniformly have such a place available, we may insert a **loop pre-header**
  - initially empty basic block with a single edge into the header
  - potentially reduces code duplication, among other things
  - the pre-header block dominates the loop (that's important)

# Inserting a loop pre-header





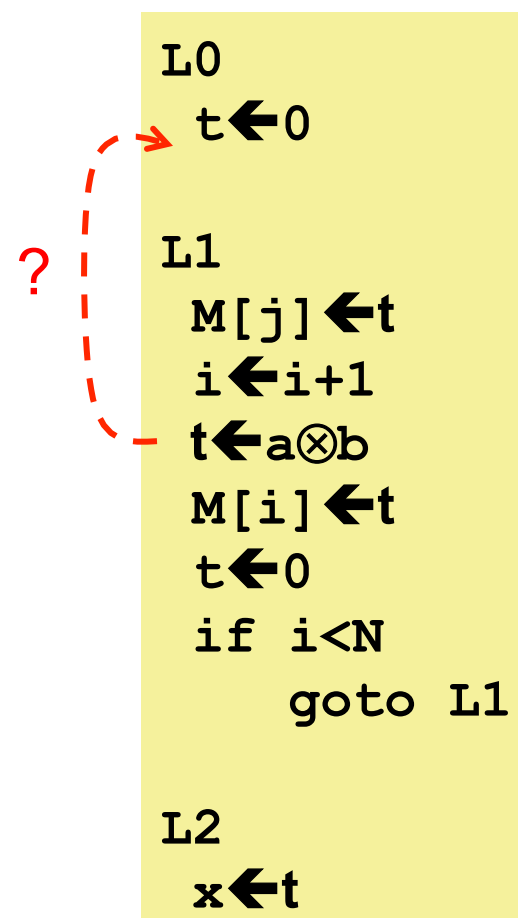
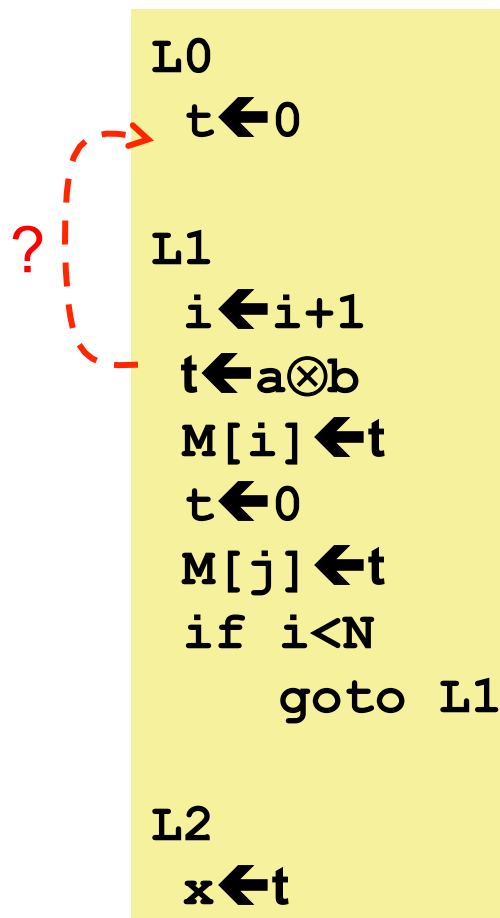
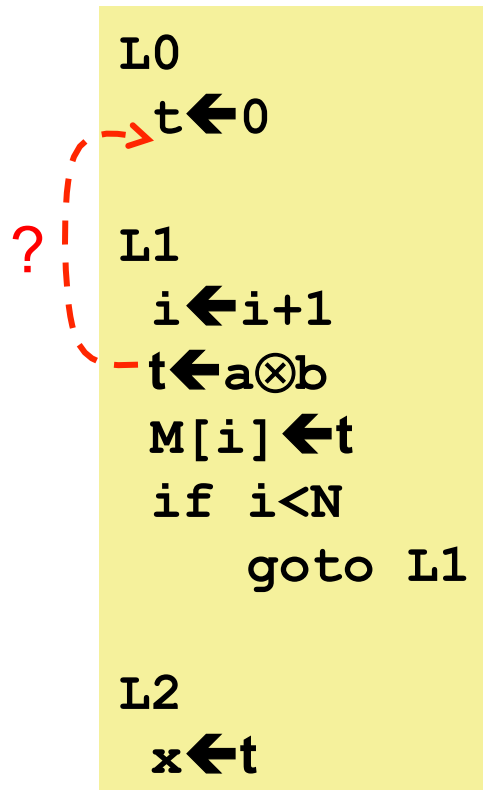
# Loop invariant computations

---

- Let  $L$  be a loop and  $d$  be “ $t \leftarrow a \otimes b$ ”
  - $d$  is **loop invariant** for  $L$  iff
    - “ $a$ ” and “ $b$ ” are constant, or
    - definitions reaching “ $a$ ” and “ $b$ ” occur outside  $L$ , or
    - only one definition reaches “ $a$ ” and one reaches “ $b$ ” and they are loop invariant for  $L$
- This is a conservative estimate of loop invariance
  - i.e., it may report  $d$  is not loop invariant when it is
    - but it will never say it is loop invariant when it is not

# To hoist or not to hoist...

Q: when is it safe to hoist  $t \leftarrow a \otimes b$ ?







# To hoist or not to hoist...

---

Before

```
L0
  t ← 0

L1
  i ← i+1
  t ← a ⊗ b
  M[i] ← t
  if i < N
    goto L1

L2
  x ← t
```

After

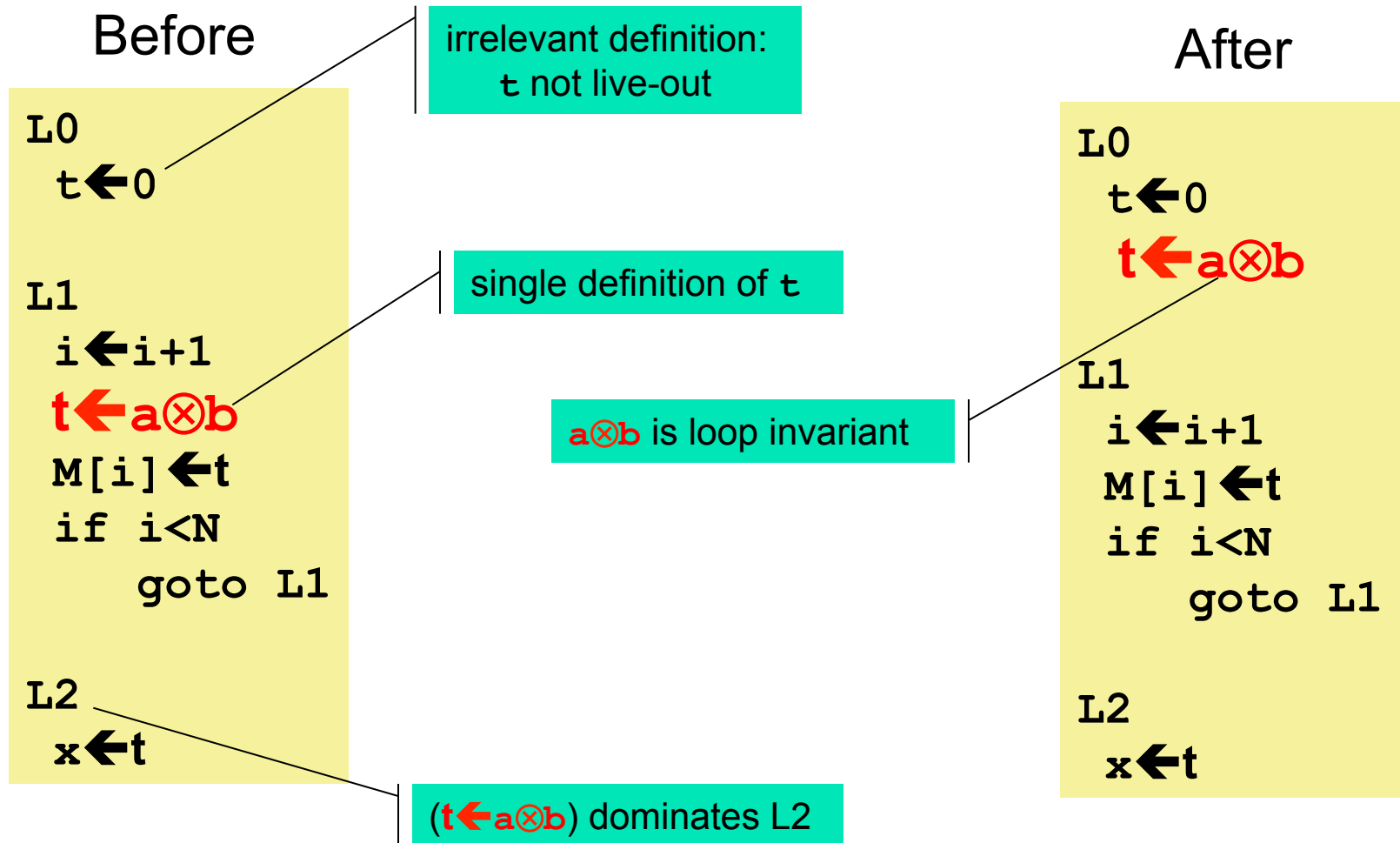
```
L0
  t ← 0
  t ← a ⊗ b

L1
  i ← i+1
  M[i] ← t
  if i < N
    goto L1

L2
  x ← t
```

\* We must determine the criteria for hoisting

# Determining the criteria for safe hoisting





# To hoist or not to hoist...

---

Before

```
L0
  t ← 0

L1
  branch (i ≥ N) L2
  i ← i + 1
  t ← a ⊗ b
  M[i] ← t
  goto L1

L2
  x ← t
```

After

```
L0
  t ← 0
  t ← a ⊗ b

L1
  branch (i ≥ N) L2
  i ← i + 1
  M[i] ← t
  goto L1

L2
  x ← t
```

# To hoist or not to hoist...

Before

```
L0
  t ← 0

L1
  branch (i ≥ N) L2
  i ← i + 1
  t ← a ⊗ b
  M[i] ← t
  goto L1

L2
  x ← t
```

always executes

only executes  
when branch fails

can be  $a \otimes b$  or 0

After

```
L0
  t ← 0
  t ← a ⊗ b

L1
  branch (i ≥ N) L2
  i ← i + 1
  M[i] ← t
  goto L1

L2
  x ← t
```

always  $a \otimes b$



# To hoist or not to hoist...

---

Before

```
L0
  t ← 0

L1
  i ← i+1
  t ← a ⊗ b
  M[i] ← t
  t ← 0
  if i < N
    goto L1

L2
```

After

```
L0
  t ← 0
  t ← a ⊗ b

L1
  i ← i+1
  M[i] ← t
  t ← 0
  if i < N
    goto L1

L2
```

# To hoist or not to hoist...

Before

```
L0
  t ← 0

L1
  i ← i+1
  t ← a ⊗ b
  M[i] ← t
  t ← 0
  if i < N
    goto L1
```

always  
a ⊗ b

After

```
L0
  t ← 0
  t ← a ⊗ b

L1
  i ← i+1
  M[i] ← t
  t ← 0
  if i < N
    goto L1
```

sometimes  
a ⊗ b

**Observe:** multiple definitions of  $t$  inside  $L1$  complicate matters

# To hoist or not to hoist...

Before

```
L0
  t ← 0

L1
  M[j] ← t
  i ← i+1
  t ← a ⊗ b
  M[i] ← t
  if i < N
    goto L1

L2
  x ← t
```

sometimes  $a \otimes b$

After

```
L0
  t ← 0
  t ← a ⊗ b

L1
  M[j] ← t
  i ← i+1
  M[i] ← t
  if i < N
    goto L1

L2
  x ← t
```

always  $a \otimes b$



# Safe hoisting of $d: t \leftarrow a \otimes b$

---

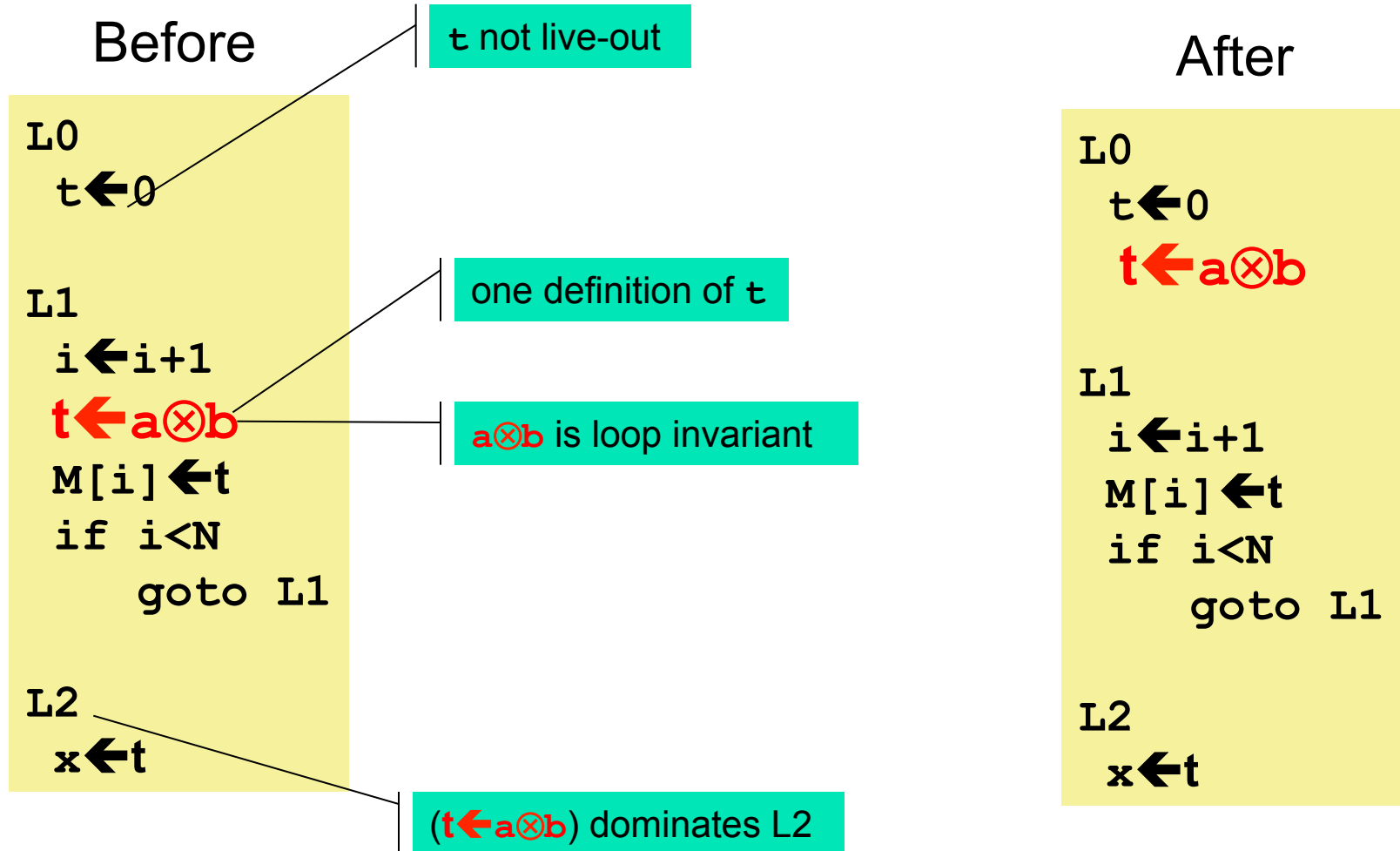
- $d$  dominates all loop exits where  $t$  is live, and
- only one definition of  $t$  in the loop, and
- $t$  is not live-out in the preheader
- assumes  $d$  is loop invariant

This is only a conservative approximation!

- will say some hoistings are unsafe when they are safe
- will not say a hoisting is safe when it isn't



# To hoist or not to hoist...



\*Therefore, hoist!



# Induction Variable Analysis

---

- Some loops have an induction variable
  - a variable “i” that is incremented by a constant or loop invariant amount each iteration
    - for loop-inv. “c”, only definitions of the form:
      - “ $i \leftarrow i + c$ ” or “ $i \leftarrow i - c$ ”
- Other variables may depend entirely on “i”
  - these are called **derived induction variables**
- Identifying induction variables within a loop enables a variety of loop optimizations
  - **strength reduction**: replacing an expensive operation by a less expensive one
  - **induction variable elimination**: removing the variable, thereby (perhaps) shortening the code and reducing register pressure

# An Example

```
s ← 0
i ← 0
L1
  branch (i > n) L2
  j ← i * 4
  k ← j + a
  x ← M[k]
  s ← s + x
  i ← i + 1
  goto L1
L2
```

Before

*i* is an induction variable

- *i*\*4 has values 0,4,8,12,...
- can perform **strength reduction**

```
s ← 0
i ← 0
j ← 0
L1
  branch (i > n) L2
  j ← j + 4
  k ← j + a
  x ← M[k]
  s ← s + x
  i ← i + 1
  goto L1
L2
```

After



# Basic Induction Variables

---

- Given a loop  $L$  with header  $h$ 
  - “ $i$ ” is a basic induction variable within  $L$  iff
    - the only definitions of “ $i$ ” have the form:
      - “ $i \leftarrow i + c$ ” or “ $i \leftarrow i - c$ ”
      - for loop-invariant “ $c$ ”
- Detection of basic induction variables is done by inspecting their form



## Derived Induction Variables in the family of “i”

---

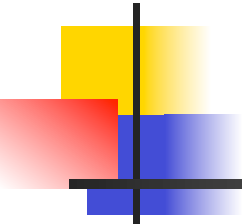
- If “i” is an induction variable (basic or otherwise) for loop L, then
  - “j” is a derived induction variable in the family of “i” means
    - all definitions of “j” in L are of the form
      - $j \leftarrow c*i + d$ 
        - where c,d are loop invariant
        - may be more than one instruction
        - Lingo: j is **determined** by (i,c,d)



## Derived induction variables in the family of “i”

---

- definitions of such a “j” may be rewritten without reference to “i”
  - That is, replace definition(s) with the effect of “ $j \leftarrow c*i + d$ ” with “ $j \leftarrow j + c*a$ ”
    - where “i” is incremented by “a”
    - c,d --- loop invariant for L
    - N.b., “c\*a” is either
      - constant: in which case, calculate it
      - loop-inv, but not constant
        - in which case compute in the pre-header



Example:  $i \leftarrow i+4, j \leftarrow 2*i+5$

---

$i$	$2*i + 5$
0	5
4	13
8	21
12	29
...	...

Initialize	$j \leftarrow 5$
iteration	$j \leftarrow j+2*4$
0 <sup>th</sup>	5
1 <sup>st</sup>	13
2 <sup>nd</sup>	21
3 <sup>rd</sup>	29
...	...

after each  
iteration

$a=4, c=2$



# Detecting derived induction variables

---

Let “j” be an induction variable for L in the family of “i”

- “k” is a derived induction variable for “j” in loop L when:
  - **(Case 1)** there is only one definition of “k” in L
    - and that definition is of the form:
      - “ $k \leftarrow c * j$ ”
      - “ $k \leftarrow j + d$ ”
        - where c,d are loop invariant





# Detecting derived induction variables

---

Let “j” be an induction variable for L in the family of “i”

- “k” is a derived induction variable for “j” in loop L when:
  - **(Case 2)** same as Case 1 AND:
    - the only definition of “j” that reaches “k” is in the loop
      - i.e., “k” depends entirely on a single definition of “j”
    - and, no definition of “i” occurs on any path between the definitions of “j” and “k”
      - i.e., the dependence of “k” on “i” via “j” maintained



# Array bounds checking

---

“Safe” programming languages  
insert dynamic checks to array references  
to avoid “out of bounds” references

```
array m[1..100] of int;  
...  
m[i] := 77;  
...
```



```
r1 ← fp + offseti  
branch (r1 > 100) Lerror  
branch (r1 < 1) Lerror  
<code for assignment>  
...
```



# Array bounds checking

Under certain circumstances, may be able to determine that the array reference is safe and eliminate the check

- relies on induction variable analysis

```
array m[1..100] of int;  
...  
m[i] := 77;  
...
```



```
r1 ← fp + offseti  
branch (r1 > 100) Lerror  
branch (r1 < 1) Lerror  
<code for assignment>  
...
```



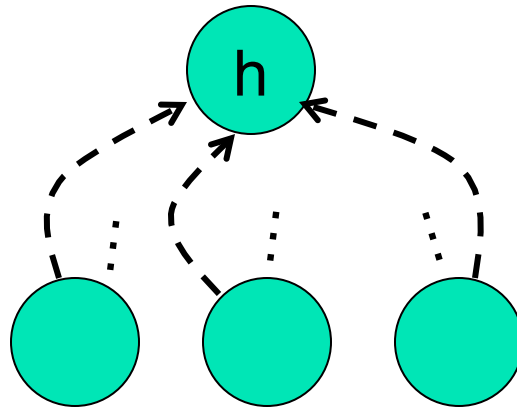
# Loop Unrolling Overview

---

- If the body of a loop  $L$  is small, it may be that it spends most of its execution “looping” rather than “computing”; i.e.,
  - incrementing induction variables
  - branching
- Simple example at source level; replace
  - “for ( $i=0$ ;  $i++$ ;  $i<2$ ) {  $c$  }” with
  - “ $i=0$ ;  $c$ ;  $i=1$ ;  $c$ ”
    - avoids branching, etc.
- The Problem: how do you do this at the machine code/IR level?
- **AKA Software Pipelining**

# Unrolling a loop (brute force)

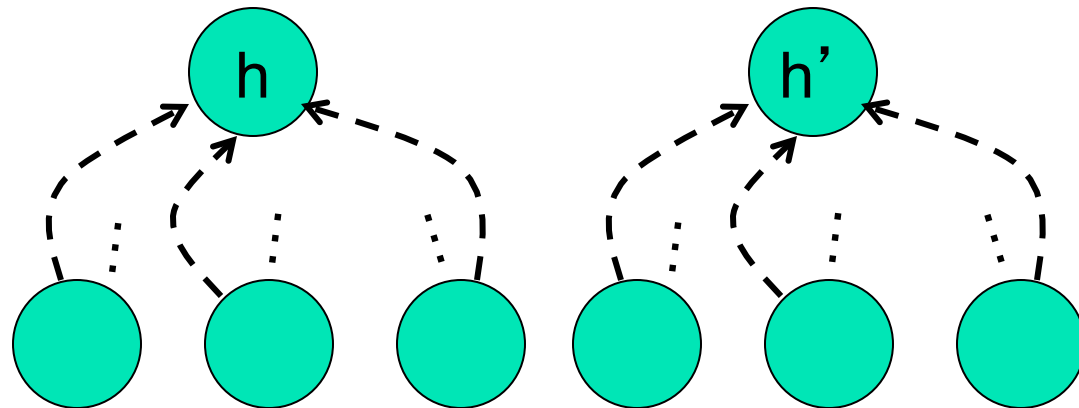
Let  $L$  be the loop:



where  $h$  is the header and  $\cdots\rightarrow$  are backedges

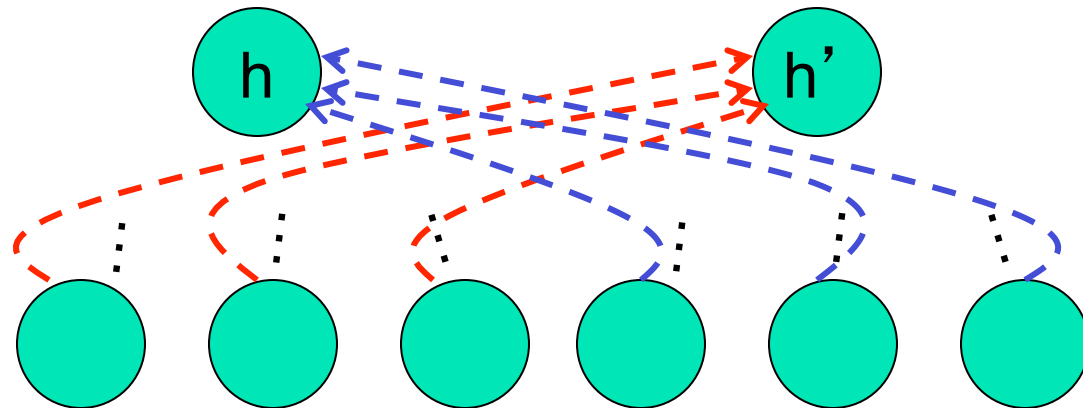
# Unrolling a loop (brute force)

Make a copy of L



# Unrolling a loop (brute force)

Reroute the backedges





# Effects of “brute force” unrolling

Before

```
L1:  x ← M[i]
      s ← s+x
      i ← i+4
      branch (i < n) L1
L2:
```

After

```
L1:  x ← M[i]
      s ← s+x
      i ← i+4
      branch (i ≥ n) L2
L1': x ← M[i]
      s ← s+x
      i ← i+4
      branch (i < n) L1
L2:
```

Q: Does this constitute an improvement?





Empirical Studies show: loops with the following are good candidates for unrolling

---

- **Single Basic Blocks**

- i.e., straight line code
- with a limited number of floating point & memory operations
  - why? Limits unrolling to loops that are most likely to benefit from instruction scheduling (i.e., ordering code w.r.t. architectural features such as data caches)

- **Small in Length**

- otherwise unrolling may have negative impact on instruction cache performance

- **Simple Loop Control**

- simplifies the unrolling transformation



# Typical unrolling candidate

---

- Note that half of the work is in “looping”
- It's a loop within a single basic block
- few instructions
- has a “repeat-until” structure

```
L1:  x ← M[i]
      s ← s + x
      i ← i + 4
      branch (i < n) L1
L2:
```



# Fragile\* unrolling (K=2)

Before

```
L1:  x ← M[i]
      s ← s+x
      i ← i+4
      branch (i<n) L1
L2:
```

After

```
L1:  x ← M[i]
      s ← s+x
      i ← i+4
      x ← M[i]
      s ← s+x
      i ← i+4
      branch (i<n) L1
L2:
```

\**Fragile*: must establish that middle branch may be removed and that loop iterates an even number of times



# “Fragile” unrolling (K=2)

Before

```
L1:  x ← M[i]
      s ← s+x
      i ← i+4
      branch (i<n) L1
L2:
```

After

```
L1:  x ← M[i]
      s ← s+x
      i ← i+4
      x ← M[i+4]
      s ← s+x
      i ← i+8
      branch (i<n) L1
L2:
```

...can do better with induction variable analysis



# Robust\* unrolling (K=2)

Before

```
L1:  x ← M[i]
      s ← s+x
      i ← i+4
      if (i<n) L1 else L2
L2:
```

*\*Robust: i.e., works for any number of iterations.*

After

```
      if (i<n-8) L1 else L2
L1:  x ← M[i]
      s ← s+x
      x ← M[i+4]
      s ← s+x
      i ← i+8
      if (i<n-8) L1 else L2
L2:  x ← M[i]
      s ← s+x
      i ← i+4
      if (i<n) L2 else L3
L3:
```

# Robust\* unrolling (K=2)

Before

```
L1:  x ← M[i]
      s ← s+x
      i ← i+4
      if (i < n) L1 else L2
L2:
```

\**Robust*: i.e., works for any number of iterations.

After

```
L1:  if (i < n-8) L1 else L2
      x ← M[i]
      s ← s+x
      x ← M[i+4]
      s ← s+x
      i ← i+8
      if (i < n-8) L1 else L2
L2:  x ← M[i]
      s ← s+x
      i ← i+4
      if (i < n) L2 else L3
L3:
```

*“twice” block*

*“once” block*

That's all, folks!