Code Generation for Data Processing

Lecture 5: Analyses and Transformations

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2025/26

Program Transformation: Motivation

- "User code" is often not very efficient

[16] FE Allen and J Cocke. *A catalogue of optimizing transformations.* 1971.

Program Transformation: Motivation

- "User code" is often not very efficient
- Also: no need to, compiler can (often?) optimize better
  - More knowledge: e.g., data layout, constants after inlining, etc.
- Allows for more pragmatic/simple code

[16] FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. .

Program Transformation: Motivation

- "User code" is often not very efficient
- Also: no need to, compiler can (often?) optimize better
  - More knowledge: e.g., data layout, constants after inlining, etc.
- Allows for more pragmatic/simple code

- Generating "better" IR code on first attempt is expensive
  - What parts are actually used? How to find out?
- Transformation to "better" code must be done *somewhere*

[16] FE Allen and J Cocke. *A catalogue of optimizing transformations.* 1971. 🌐.

Program Transformation: Motivation

- "User code" is often not very efficient
- Also: no need to, compiler can (often?) optimize better
  - More knowledge: e.g., data layout, constants after inlining, etc.
- Allows for more pragmatic/simple code

- Generating "better" IR code on first attempt is expensive
  - What parts are actually used? How to find out?
- Transformation to "better" code must be done *somewhere*

- Optimization is a misnomer: we don't know whether it improves code!
  - Many transformations are driven by heuristics

[16] FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. .

Program Transformation: Motivation

- "User code" is often not very efficient
- Also: no need to, compiler can (often?) optimize better
  - More knowledge: e.g., data layout, constants after inlining, etc.
- Allows for more pragmatic/simple code

- Generating "better" IR code on first attempt is expensive
  - What parts are actually used? How to find out?
- Transformation to "better" code must be done *somewhere*

- Optimization is a misnomer: we don't know whether it improves code!
  - Many transformations are driven by heuristics
- Many types of optimizations are well-known[16]

[16] FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971.

Dead Block Elimination

- CFG not necessarily connected
- E.g., consequence of optimization
  - Conditional branch $\rightarrow$ unconditional branch

Dead Block Elimination

- CFG not necessarily connected
- E.g., consequence of optimization
  - Conditional branch $\rightarrow$ unconditional branch

- Removing dead blocks is trivial
  1. DFS traversal of CFG from entry, mark visited blocks
  2. Remove unmarked blocks

Optimization Example 1

```
define i32 @fac(i32 %0) {
  br label %for.header
for.header: ; preds = %for.body, %1
  %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
  %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
  %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
  %cond = icmp sle i32 %i, %0
  br i1 %cond, label %for.body, label %exit
for.body: ; preds = %for.header
  %a.new = mul i32 %a, %i
  %b.new = add i32 %b, %i
  %i.new = add i32 %i, 1
  br label %for.header
exit: ; preds = %for.header
  %absum = add i32 %a, %b
  ret i32 %a
}
```

Simple Dead Code Elimination (DCE)

- Look for trivially dead instructions
  - No users or side-effects
  - Calls *might* be removed

1. Add all instructions to work queue
2. While work queue not empty:
   2.1 Check for deadness (zero users, no side-effects)
   2.2 If dead, remove and add all operands to work queue

Warning: Don't implement it this naively, this is inefficient

Applying Simple DCE

```
           define i32 @fac(i32 %0) {
eff.: cf    br label %for.header
           for.header: ; preds = %for.body, %1
users: 3    %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
users: 2    %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
users: 4    %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
users: 1    %cond = icmp sle i32 %i, %0
eff.: cf    br i1 %cond, label %for.body, label %exit
           for.body: ; preds = %for.header
users: 1    %a.new = mul i32 %a, %i
users: 1    %b.new = add i32 %b, %i
users: 1    %i.new = add i32 %i, 1
eff.: cf    br label %for.header
           exit: ; preds = %for.header
users: 0    %absum = add i32 %a, %b
eff.: cf    ret i32 %a
           }
```

Applying Simple DCE

```
         define i32 @fac(i32 %0) {
eff.: cf   br label %for.header
         for.header: ; preds = %for.body, %1
users: 2   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
users: 1   %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
users: 4   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
users: 1   %cond = icmp sle i32 %i, %0
eff.: cf   br i1 %cond, label %for.body, label %exit
         for.body: ; preds = %for.header
users: 1   %a.new = mul i32 %a, %i
users: 1   %b.new = add i32 %b, %i
users: 1   %i.new = add i32 %i, 1
eff.: cf   br label %for.header
         exit: ; preds = %for.header

eff.: cf   ret i32 %a
         }
```

Dead Code Elimination

- Problem: unused value cycles

Dead Code Elimination

- Problem: unused value cycles
- Idea: find “value sinks” and mark all needed values as live
  unmarked values can be removed
  - Sink: instruction with side effects (e.g., store, control flow)
  - Alternative formulation: assume instruction is dead unless proven otherwise

1. Only mark instrs. with side effects as live
2. Populate work list with newly added live instrs.
3. While work list not empty:
   3.1 Mark dead operand instructions as live and add to work list
4. Remove instructions not marked as live

Applying Liveness-based DCE

Work list (stack)

```
define i32 @fac(i32 %0) {
 br₁ label %for.header
for.header: ; preds = %for.body, %1
 %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
 %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
 %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
 %cond = icmp sle i32 %i, %0
 br₂ i1 %cond, label %for.body, label %exit
for.body: ; preds = %for.header
 %a.new = mul i32 %a, %i
 %b.new = add i32 %b, %i
 %i.new = add i32 %i, 1
 br₃ label %for.header
exit: ; preds = %for.header
 %absum = add i32 %a, %b
 ret i32 %a
}
```

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live    br₁ label %for.header
      for.header: ; preds = %for.body, %1
        %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
        %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
        %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
        %cond = icmp sle i32 %i, %0
live    br₂ i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
        %a.new = mul i32 %a, %i
        %b.new = add i32 %b, %i
        %i.new = add i32 %i, 1
live    br₃ label %for.header
      exit: ; preds = %for.header
        %absum = add i32 %a, %b
live    ret i32 %a
      }
```

Work list (stack)

$\mathtt{br}_1$

$\mathtt{br}_2$

$\mathtt{br}_3$

`ret`

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live   br1 label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
       %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
       %cond = icmp sle i32 %i, %0
live   br2 i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
       %a.new = mul i32 %a, %i
       %b.new = add i32 %b, %i
       %i.new = add i32 %i, 1
live   br3 label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

$\mathtt{br}_1$

$\mathtt{br}_2$

$\mathtt{br}_3$

%a

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live   br₁ label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
       %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
       %cond = icmp sle i32 %i, %0
live   br₂ i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
live   %a.new = mul i32 %a, %i
       %b.new = add i32 %b, %i
       %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

$\texttt{br}_1$

$\texttt{br}_2$

$\texttt{br}_3$

`%a.new`

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live   br₁ label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
       %cond = icmp sle i32 %i, %0
live   br₂ i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
live   %a.new = mul i32 %a, %i
       %b.new = add i32 %b, %i
       %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

$\mathtt{br}_1$

$\mathtt{br}_2$

$\mathtt{br}_3$

%i

Applying Liveness-based DCE

```
     define i32 @fac(i32 %0) {
live  br1 label %for.header
     for.header: ; preds = %for.body, %1
live  %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
      %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live  %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
      %cond = icmp sle i32 %i, %0
live  br2 i1 %cond, label %for.body, label %exit
     for.body: ; preds = %for.header
live  %a.new = mul i32 %a, %i
      %b.new = add i32 %b, %i
live  %i.new = add i32 %i, 1
live  br3 label %for.header
     exit: ; preds = %for.header
      %absum = add i32 %a, %b
live  ret i32 %a
     }
```

Work list (stack)

$\texttt{br}_1$

$\texttt{br}_2$

$\texttt{br}_3$

`%i.new`

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live   br₁ label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
       %cond = icmp sle i32 %i, %0
live   br₂ i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
live   %a.new = mul i32 %a, %i
       %b.new = add i32 %b, %i
live   %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

br_1

br_2

br_3

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live   br1 label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
       %cond = icmp sle i32 %i, %0
live   br2 i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
live   %a.new = mul i32 %a, %i
       %b.new = add i32 %b, %i
live   %i.new = add i32 %i, 1
live   br3 label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

br$_1$

br$_2$

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live   br₁ label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
live   %cond = icmp sle i32 %i, %0
live   br₂ i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
live   %a.new = mul i32 %a, %i
       %b.new = add i32 %b, %i
live   %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

br_1

%cond

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live   br₁ label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
live   %cond = icmp sle i32 %i, %0
live   br₂ i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
live   %a.new = mul i32 %a, %i
       %b.new = add i32 %b, %i
live   %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

br_1

Applying Liveness-based DCE

Work list (stack)

```
      define i32 @fac(i32 %0) {
live   br₁ label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
live   %cond = icmp sle i32 %i, %0
live   br₂ i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
live   %a.new = mul i32 %a, %i
       %b.new = add i32 %b, %i
live   %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Applying Liveness-based DCE

Work list (stack)

```
      define i32 @fac(i32 %0) {
live   br₁ label %for.header
      for.header: ; preds = %for.body, %1
live   %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]

live   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
live   %cond = icmp sle i32 %i, %0
live   br₂ i1 %cond, label %for.body, label %exit
      for.body: ; preds = %for.header
live   %a.new = mul i32 %a, %i

live   %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header

live   ret i32 %a
      }
```

Optimization Example 2

```
define i32 @trivial() {
entry:
  br label %for.cond
for.cond:
  %x = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ugt i32 %x, 0
  br i1 %cmp, label %for.inc, label %ret
for.inc:
  %inc = add i32 %x, 1
  br label %for.cond
ret:
  ret i32 %x
}
```

Sparse Conditional Constant Propagation[17]

- Constant folding and dead code elimination separately insufficient
- Idea: be optimistic about reachability and constantness
  - Assume block is dead/value is constant unless proven othewise
  - Ignore edges that are not proven as executable

[17] MN Wegman and FK Zadeck. "Constant propagation with conditional branches". In: *TOPLAS* 13.2 (1991), pp. 181–210.

Sparse Conditional Constant Propagation[17]

- Constant folding and dead code elimination separately insufficient
- Idea: be optimistic about reachability and constantness
  - Assume block is dead/value is constant unless proven othewise
  - Ignore edges that are not proven as executable
- For every value: store lattice element
  - Lattice: $\top$ (any value) $>$ *constant* (single constant value) $> \bot$ (dynamic)
  - $\top \wedge x = x, \bot \wedge x = \bot, x \wedge y = (x \text{ if } x = y \text{ else } \bot)$
- For every edge: store executability
  - ϕ-nodes: edges that are not (yet) marked executable get $\top$

[17] MN Wegman and FK Zadeck. "Constant propagation with conditional branches". In: *TOPLAS* 13.2 (1991), pp. 181–210.

SCCP Algorithm

State:

- EdgeWorklist
- InstrWorklist
- EdgeExecutable
  - Initialize `false`
- InstrLattice
  - Initialize $\top$

- EdgeWorklist $\leftarrow$ edge to entry block
- Get item from any work list; stop when empty
- Edge – if not yet visited, mark visited and:
  - Visited block first time: *visit* all instrs (incl. ϕ)
  - Otherwise: *visit* just ϕ-nodes
- Instr – if block is reachable: *visit*

- At end: replace instrs with const lattice values

SCCP Algorithm II

visit

- ϕ-node: update lattice, meet of visited edge values
- Other instructions: try constant fold

- If lattice value changed:
  - Add all users to instruction worklist
  - Branch with constant/no condition: add selected edge to worklist
  - Branch with $\perp$: add all edges to worklist

```
define i32 @fn() {
entry:
  br label %loop
loop:
  %j2 = phi i32 [ 1, %entry ], [ %j4, %ifmerge ]
  %k2 = phi i32 [ 0, %entry ], [ %k4, %ifmerge ]
  %kcond = icmp slt i32 %k2, 100
  br i1 %kcond, label %loopbody, label %ret
loopbody:
  %jcond = icmp slt i32 %j2, 20
  br i1 %jcond, label %then, label %else
then:
  %k3 = add i32 %k2, 1
  br label %ifmerge
else:
  %k5 = add i32 %k2, 1
  br label %ifmerge
ifmerge:
  %j4 = phi i32 [ 1, %then ], [ %k2, %else ]
  %k4 = phi i32 [ %k3, %then ], [ %k5, %else ]
  br label %loop
ret:
  ret i32 %j2
}
```

Apply SCCP algorithm and fold constants. Then, remove dead instructions and blocks.

SCCP: Misc

- Strictly more powerful than seperate constant folding + DCE
- Runtime:

SCCP: Misc

- Strictly more powerful than seperate constant folding + DCE
- Runtime: $\mathcal{O}(\#\text{CFG-edges} + \#\text{instr.-operands})$
  - Every CFG edge is processed at most once
  - Every def–use relation is processed at most twice
    lattice can lower at most twice ($\top \rightarrow \textit{const}, \textit{const} \rightarrow \bot$)
- Can also be used as inter-procedural optimization
  reaching into module-internal functions

Optimization Example 3

```
define i32 @foo(i32 %0, ptr %1, ptr %2) {
  %4 = zext i32 %0 to i64
  %5 = getelementptr i32, ptr %1, i64 %4
  %6 = load i32, ptr %5, align 4
  %7 = zext i32 %0 to i64
  %8 = getelementptr i32, ptr %2, i64 %7
  %9 = load i32, ptr %8, align 4
  %10 = add nsw i32 %6, %9
  ret i32 %10
}
```

Common Subexpression Elimination (CSE) – Attempt 1

- Idea: find/eliminate redundant computation of same value

Common Subexpression Elimination (CSE) – Attempt 1

- Idea: find/eliminate redundant computation of same value
- Keep track of previously seen values in hash map
- Iterate over all instructions
  - If found in map, remove and replace references
  - Otherwise add to map
- Easy, right?

CSE Attempt 1 – Example 1

```
define i32 @foo(i32 %0, ptr %1, ptr %2) {
 %4 = zext i32 %0 to i64
 %5 = getelementptr i32, ptr %1, i64 %4
 %6 = load i32, ptr %5, align 4
 %7 = zext i32 %0 to i64
 %8 = getelementptr i32, ptr %2, i64 %7
 %9 = load i32, ptr %8, align 4
 %10 = add nsw i32 %6, %9
 ret i32 %10
}
```

CSE Attempt 1 – Example 1

→ ht

```
define i32 @foo(i32 %0, ptr %1, ptr %2) {
 %4 = zext i32 %0 to i64
 %5 = getelementptr i32, ptr %1, i64 %4
 %6 = load i32, ptr %5, align 4
 %7 = zext i32 %0 to i64
 %8 = getelementptr i32, ptr %2, i64 %7
 %9 = load i32, ptr %8, align 4
 %10 = add nsw i32 %6, %9
 ret i32 %10
}
```

CSE Attempt 1 – Example 1

→ ht
→ ht

```
define i32 @foo(i32 %0, ptr %1, ptr %2) {
 %4 = zext i32 %0 to i64
 %5 = getelementptr i32, ptr %1, i64 %4
 %6 = load i32, ptr %5, align 4
 %7 = zext i32 %0 to i64
 %8 = getelementptr i32, ptr %2, i64 %7
 %9 = load i32, ptr %8, align 4
 %10 = add nsw i32 %6, %9
 ret i32 %10
}
```

CSE Attempt 1 – Example 1

$\rightarrow$ ht
$\rightarrow$ ht
$\rightarrow$ ht

```
define i32 @foo(i32 %0, ptr %1, ptr %2) {
 %4 = zext i32 %0 to i64
 %5 = getelementptr i32, ptr %1, i64 %4
 %6 = load i32, ptr %5, align 4
 %7 = zext i32 %0 to i64
 %8 = getelementptr i32, ptr %2, i64 %7
 %9 = load i32, ptr %8, align 4
 %10 = add nsw i32 %6, %9
 ret i32 %10
}
```

CSE Attempt 1 – Example 1

```
          define i32 @foo(i32 %0, ptr %1, ptr %2) {
→ ht       %4 = zext i32 %0 to i64
→ ht       %5 = getelementptr i32, ptr %1, i64 %4
→ ht       %6 = load i32, ptr %5, align 4
dup %4     %7 = zext i32 %0 to i64
           %8 = getelementptr i32, ptr %2, i64 %7
           %9 = load i32, ptr %8, align 4
           %10 = add nsw i32 %6, %9
           ret i32 %10
          }
```

CSE Attempt 1 – Example 1

```
        define i32 @foo(i32 %0, ptr %1, ptr %2) {
→ ht      %4 = zext i32 %0 to i64
→ ht      %5 = getelementptr i32, ptr %1, i64 %4
→ ht      %6 = load i32, ptr %5, align 4
dup %4    %7 = zext i32 %0 to i64
          %8 = getelementptr i32, ptr %2, i64 %4
          %9 = load i32, ptr %8, align 4
          %10 = add nsw i32 %6, %9
          ret i32 %10
        }
```

CSE Attempt 1 – Example 1

```
          define i32 @foo(i32 %0, ptr %1, ptr %2) {
→ ht        %4 = zext i32 %0 to i64
→ ht        %5 = getelementptr i32, ptr %1, i64 %4
→ ht        %6 = load i32, ptr %5, align 4
dup %4      %7 = zext i32 %0 to i64
→ ht        %8 = getelementptr i32, ptr %2, i64 %4
            %9 = load i32, ptr %8, align 4
            %10 = add nsw i32 %6, %9
            ret i32 %10
          }
```

CSE Attempt 1 – Example 1

```
        define i32 @foo(i32 %0, ptr %1, ptr %2) {
→ ht      %4 = zext i32 %0 to i64
→ ht      %5 = getelementptr i32, ptr %1, i64 %4
→ ht      %6 = load i32, ptr %5, align 4
dup %4    %7 = zext i32 %0 to i64
→ ht      %8 = getelementptr i32, ptr %2, i64 %4
→ ht      %9 = load i32, ptr %8, align 4
          %10 = add nsw i32 %6, %9
          ret i32 %10
        }
```

CSE Attempt 1 – Example 1

```
         define i32 @foo(i32 %0, ptr %1, ptr %2) {
→ ht       %4 = zext i32 %0 to i64
→ ht       %5 = getelementptr i32, ptr %1, i64 %4
→ ht       %6 = load i32, ptr %5, align 4
dup %4     %7 = zext i32 %0 to i64
→ ht       %8 = getelementptr i32, ptr %2, i64 %4
→ ht       %9 = load i32, ptr %8, align 4
→ ht       %10 = add nsw i32 %6, %9
           ret i32 %10
         }
```

CSE Attempt 1 – Example 1

```
        define i32 @foo(i32 %0, ptr %1, ptr %2) {
→ ht      %4 = zext i32 %0 to i64
→ ht      %5 = getelementptr i32, ptr %1, i64 %4
→ ht      %6 = load i32, ptr %5, align 4
dup %4    %7 = zext i32 %0 to i64
→ ht      %8 = getelementptr i32, ptr %2, i64 %4
→ ht      %9 = load i32, ptr %8, align 4
→ ht      %10 = add nsw i32 %6, %9
→ ht      ret i32 %10
        }
```

CSE Attempt 1 – Example 1

```
        define i32 @foo(i32 %0, ptr %1, ptr %2) {
→ ht     %4 = zext i32 %0 to i64
→ ht     %5 = getelementptr i32, ptr %1, i64 %4
→ ht     %6 = load i32, ptr %5, align 4
dup %4   %7 = zext i32 %0 to i64
→ ht     %8 = getelementptr i32, ptr %2, i64 %4
→ ht     %9 = load i32, ptr %8, align 4
→ ht     %10 = add nsw i32 %6, %9
→ ht     ret i32 %10
        }
```

- Obsolete instr. can be killed immediately, or in a later DCE

CSE Attempt 1 – Example 2

```
define i32 @square(i32 %a, i32 %b) {
entry:
 %cmp = icmp slt i32 %a, %b
 br i1 %cmp, label %if.then, label %if.end
if.then: ; preds = %entry
 %add1 = add i32 %a, %b
 br label %if.end
if.end: ; preds = %if.then, %entry
 %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
 %add2 = add i32 %a, %b
 %res = add i32 %condvar, %add2
 ret i32 %res
}
```

CSE Attempt 1 – Example 2

→ ht

```
define i32 @square(i32 %a, i32 %b) {
entry:
 %cmp = icmp slt i32 %a, %b
 br i1 %cmp, label %if.then, label %if.end
if.then: ; preds = %entry
 %add1 = add i32 %a, %b
 br label %if.end
if.end: ; preds = %if.then, %entry
 %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
 %add2 = add i32 %a, %b
 %res = add i32 %condvar, %add2
 ret i32 %res
}
```

CSE Attempt 1 – Example 2

```
        define i32 @square(i32 %a, i32 %b) {
        entry:
→ ht     %cmp = icmp slt i32 %a, %b
→ ht     br i1 %cmp, label %if.then, label %if.end
        if.then: ; preds = %entry
         %add1 = add i32 %a, %b
         br label %if.end
        if.end: ; preds = %if.then, %entry
         %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
         %add2 = add i32 %a, %b
         %res = add i32 %condvar, %add2
         ret i32 %res
        }
```

CSE Attempt 1 – Example 2

```
       define i32 @square(i32 %a, i32 %b) {
       entry:
→ ht    %cmp = icmp slt i32 %a, %b
→ ht    br i1 %cmp, label %if.then, label %if.end
       if.then: ; preds = %entry
→ ht    %add1 = add i32 %a, %b
        br label %if.end
       if.end: ; preds = %if.then, %entry
        %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
        %add2 = add i32 %a, %b
        %res = add i32 %condvar, %add2
        ret i32 %res
       }
```

CSE Attempt 1 – Example 2

```
                define i32 @square(i32 %a, i32 %b) {
                entry:
→ ht             %cmp = icmp slt i32 %a, %b
→ ht             br i1 %cmp, label %if.then, label %if.end
                if.then: ; preds = %entry
→ ht             %add1 = add i32 %a, %b
→ ht             br label %if.end
                if.end: ; preds = %if.then, %entry
                 %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
                 %add2 = add i32 %a, %b
                 %res = add i32 %condvar, %add2
                 ret i32 %res
                }
```

CSE Attempt 1 – Example 2

```
        define i32 @square(i32 %a, i32 %b) {
        entry:
→ ht     %cmp = icmp slt i32 %a, %b
→ ht     br i1 %cmp, label %if.then, label %if.end
        if.then: ; preds = %entry
→ ht     %add1 = add i32 %a, %b
→ ht     br label %if.end
        if.end: ; preds = %if.then, %entry
→ ht     %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
         %add2 = add i32 %a, %b
         %res = add i32 %condvar, %add2
         ret i32 %res
        }
```

CSE Attempt 1 – Example 2

```
              define i32 @square(i32 %a, i32 %b) {
              entry:
→ ht            %cmp = icmp slt i32 %a, %b
→ ht            br i1 %cmp, label %if.then, label %if.end
              if.then: ; preds = %entry
→ ht            %add1 = add i32 %a, %b
→ ht            br label %if.end
              if.end: ; preds = %if.then, %entry
→ ht            %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
dup %add1       %add2 = add i32 %a, %b
                %res = add i32 %condvar, %add2
                ret i32 %res
              }
```

CSE Attempt 1 – Example 2

```
             define i32 @square(i32 %a, i32 %b) {
             entry:
→ ht           %cmp = icmp slt i32 %a, %b
→ ht           br i1 %cmp, label %if.then, label %if.end
             if.then: ; preds = %entry
→ ht           %add1 = add i32 %a, %b
→ ht           br label %if.end
             if.end: ; preds = %if.then, %entry
→ ht           %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
dup %add1      %add2 = add i32 %a, %b
               %res = add i32 %condvar, %add1
               ret i32 %res
             }
```

CSE Attempt 1 – Example 2

```
                define i32 @square(i32 %a, i32 %b) {
                entry:
→ ht             %cmp = icmp slt i32 %a, %b
→ ht             br i1 %cmp, label %if.then, label %if.end
                if.then: ; preds = %entry
→ ht             %add1 = add i32 %a, %b
→ ht             br label %if.end
                if.end: ; preds = %if.then, %entry
→ ht             %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
dup %add1        %add2 = add i32 %a, %b
→ ht             %res = add i32 %condvar, %add1
                 ret i32 %res
                }
```

CSE Attempt 1 – Example 2

```
             define i32 @square(i32 %a, i32 %b) {
             entry:
→ ht           %cmp = icmp slt i32 %a, %b
→ ht           br i1 %cmp, label %if.then, label %if.end
             if.then: ; preds = %entry
→ ht           %add1 = add i32 %a, %b
→ ht           br label %if.end
             if.end: ; preds = %if.then, %entry
→ ht           %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
dup %add1      %add2 = add i32 %a, %b
→ ht           %res = add i32 %condvar, %add1
→ ht           ret i32 %res
             }
```

CSE Attempt 1 – Example 2

```
            define i32 @square(i32 %a, i32 %b) {
            entry:
→ ht          %cmp = icmp slt i32 %a, %b
→ ht          br i1 %cmp, label %if.then, label %if.end
            if.then: ; preds = %entry
→ ht          %add1 = add i32 %a, %b
→ ht          br label %if.end
            if.end: ; preds = %if.then, %entry
→ ht          %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
dup %add1     %add2 = add i32 %a, %b
→ ht          %res = add i32 %condvar, %add1
→ ht          ret i32 %res
            }
```

```
Instruction does not dominate all uses!
error: input module is broken!
```

World Domination

Domination

- Remember: CFG $G = (N, E, s)$ with digraph (N, E) and entry $s \in N$
- Dominate: d dom n iff every path from s to n contains d
  - Dominators of n: $DOM(n) = \{d | d \text{ dom } n\}$
- Strictly dominate: $d \text{ sdom } n \Leftrightarrow d \text{ dom } n \wedge d \neq n$
- Immediate dominator:
  $\text{idom}\ (n) = d : d \text{ sdom } n \wedge \nexists d'.d \text{ sdom } d' \wedge d' \text{ sdom } n$

Domination

- Remember: CFG $G = (N, E, s)$ with digraph (N, E) and entry $s \in N$
- Dominate: d dom n iff every path from s to n contains d
  - Dominators of n: $DOM(n) = \{d | d \text{ dom } n\}$
- Strictly dominate: d sdom $n \Leftrightarrow d$ dom $n \wedge d \neq n$
- Immediate dominator:
  $\text{idom}\ (n) = d : d \text{ sdom } n \wedge \nexists d'.d \text{ sdom } d' \wedge d' \text{ sdom } n$

$\Rightarrow$ All strict dominators are always executed before the block

Domination

- Remember: CFG $G = (N, E, s)$ with digraph (N, E) and entry $s \in N$
- Dominate: d dom n iff every path from s to n contains d
  - Dominators of n: $DOM(n) = \{d | d \text{ dom } n\}$
- Strictly dominate: d sdom $n \Leftrightarrow d$ dom $n \wedge d \neq n$
- Immediate dominator:
  idom $(n) = d : d \text{ sdom } n \wedge \nexists d'.d \text{ sdom } d' \wedge d' \text{ sdom } n$

$\Rightarrow$ All strict dominators are always executed before the block

$\Rightarrow$ All values from dominators available/usable

Domination

- Remember: CFG $G = (N, E, s)$ with digraph (N, E) and entry $s \in N$
- Dominate: d dom n iff every path from s to n contains d
  - Dominators of n: $DOM(n) = \{d | d \text{ dom } n\}$
- Strictly dominate: $d \text{ sdom } n \Leftrightarrow d \text{ dom } n \wedge d \neq n$
- Immediate dominator:
  $\text{idom } (n) = d : d \text{ sdom } n \wedge \nexists d'.d \text{ sdom } d' \wedge d' \text{ sdom } n$

$\Rightarrow$ All strict dominators are always executed before the block

$\Rightarrow$ All values from dominators available/usable

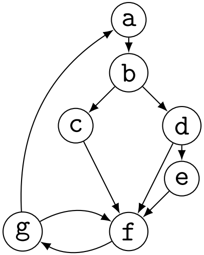
$\Rightarrow$ All values not from dominators **not** usable

Dominator Tree

- Tree of immediate dominators
- Allows to iterate over blocks in pre-order/post-order
- Answer *a* sdom *b* quickly

Control Flow Graph

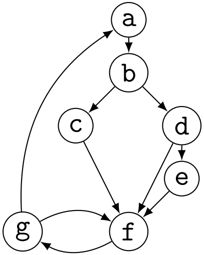
Dominator Tree



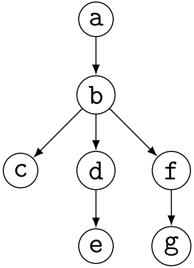
Dominator Tree

- Tree of immediate dominators
- Allows to iterate over blocks in pre-order/post-order
- Answer *a* sdom *b* quickly

Control Flow Graph

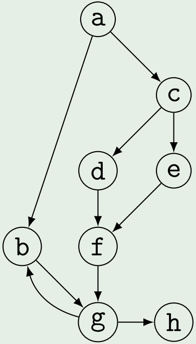


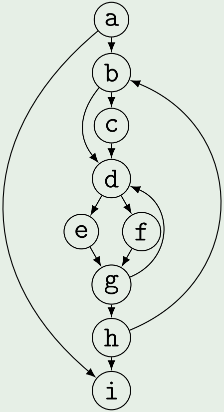
Dominator Tree



Dominator Tree – Example

Construct the dominator tree for the following CFGs (entry at a):





Dominator Tree: Construction

- Naive: inefficient (but reasonably simple)[18]
  - For each block: find a path from the root – superset of dominators
  - Remove last block on path and check for alternative path
  - If no alternative path exists, last block is idom

[18] ES Lowry and CW Medlock. "Object code optimization". In: *CACM* 12.1 (1969), pp. 13–22.

Dominator Tree: Construction

- Naive: inefficient (but reasonably simple)[18]
  - For each block: find a path from the root – superset of dominators
  - Remove last block on path and check for alternative path
  - If no alternative path exists, last block is idom
- Lengauer–Tarjan: more efficient methods[19]
  - Simple method in $\mathcal{O}(m \log n)$; sophisticated method in $\mathcal{O}(m \cdot \alpha(m, n))$
    ($\alpha(m, n)$ is the inverse Ackermann function, grows *extremely* slowly)
  - Used in some compilers[20]
- Semi-NCA: $\mathcal{O}(n^2)$, but lower constant factors[21]

[18] ES Lowry and CW Medlock. "Object code optimization". In: *CACM* 12.1 (1969), pp. 13–22. .

[19] T Lengauer and RE Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *TOPLAS* 1.1 (1979), pp. 121–141.

[20] Example: `https://github.com/WebKit/WebKit/blob/aabfacb/Source/WTF/wtf/Dominators.h`

[21] L Georgiadis. "Linear-Time Algorithms for Dominators and Related Problems". PhD thesis. Princeton University, Nov. 2005.

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator:

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator: ...lookup *idom*
- Iterate over all dominators/dominated by:

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator: ...lookup *idom*
- Iterate over all dominators/dominated by: ...trivial
- Check whether *a* sdom *b*

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator: ...lookup *idom*
- Iterate over all dominators/dominated by: ...trivial
- Check whether a sdom b[23]
  - $a.preNum < b.preNum \;\wedge\; a.postNum > b.postNum$
  - After updates, numbers might be invalid: recompute or walk tree

[23] PF Dietz. "Maintaining order in a linked list". In: *STOC*. 1982, pp. 122–127.

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator: ...lookup *idom*
- Iterate over all dominators/dominated by: ...trivial
- Check whether a sdom b[23]
  - $a.preNum < b.preNum \ \wedge \ a.postNum > b.postNum$
  - After updates, numbers might be invalid: recompute or walk tree
- Problem: dominance of unreachable blocks ill-defined $\rightsquigarrow$ special handling

[23] PF Dietz. "Maintaining order in a linked list". In: *STOC*. 1982, pp. 122–127.

CSE Attempt 2

- Option 1:
  - For identical instructions, store all
  - Add dominance check before replacing
  - Visit nodes in reverse post-order (i.e., topological order)
- Option 2 (Global Value Numbering):[24]
  - Do a DFS over dominator tree
  - Use scoped hashmap to track available values

Does this work?

[24] P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997.

CSE Attempt 2

- Option 1:
  - For identical instructions, store all
  - Add dominance check before replacing
  - Visit nodes in reverse post-order (i.e., topological order)
- Option 2 (Global Value Numbering):[24]
  - Do a DFS over dominator tree
  - Use scoped hashmap to track available values

Does this work? Yes.

[24] P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997.

CSE: Hashing an Instruction (and Beyond)

- Needs hash function *and* "relaxed" equality
- Idea: combine opcode and operands/constants into hash value
  - E.g. assign number to values (hence "value numbering")
  - Alternatively: use pointer or index for instruction result operands

CSE: Hashing an Instruction (and Beyond)

- Needs hash function *and* “relaxed” equality
- Idea: combine opcode and operands/constants into hash value
  - E.g. assign number to values (hence “value numbering”)
  - Alternatively: use pointer or index for instruction result operands
- Canonicalize commutative operations
  - Order operands deterministically, e.g., by number/address
- Identities: `a+(b+c)` vs. `(a+b)+c`

More Complex Global Value Numbering

- Hash-based approach only catches trivially removable duplicates

[25] K Gargi. "A sparse algorithm for predicated global value numbering". In: *PLDI*. 2002, pp. 45–56.

More Complex Global Value Numbering

- Hash-based approach only catches trivially removable duplicates
- Alternative: partition values into *congruence classes*
  - Congruent values are guaranteed to always have the same value
- Optimistic approach: values are congruent unless proven otherwise
- Pessimistic approach: values are not congruent unless proven
- Combinable with: reassociation, DCE, constant folding
- Rather complex, but can be highly beneficial[25]

[25] K Gargi. "A sparse algorithm for predicated global value numbering". In: *PLDI*. 2002, pp. 45–56.

Inlining

Inlining

- Inlining: copy function body in place of the call
  - Historically also referred to as “procedure integration”
- Benefits:

Inlining

- Inlining: copy function body in place of the call
  - Historically also referred to as "procedure integration"
- Benefits:
  - More optimization opportunities: constant propagation, dead code elimination,
    better register allocation, etc.
  - Avoids call overhead: stack frame setup, register saving, etc.
- Inlining is crucial for performance in many programming languages

Inlining

- Inlining: copy function body in place of the call
  - Historically also referred to as "procedure integration"

- Benefits:
  - More optimization opportunities: constant propagation, dead code elimination,
    better register allocation, etc.
  - Avoids call overhead: stack frame setup, register saving, etc.

- Inlining is crucial for performance in many programming languages
  - "Zero-cost" abstractions typically rely on small functions
  - Critical in e.g. C++/Rust; not too important for C

Inlining: Decision

- First: determine whether function is inlinable
- Easy way: delegate to programmer (`__attribute__((always_inline))`)
- Otherwise: very difficult problem

- Problem 1: ordering

Inlining: Decision

- First: determine whether function is inlinable
- Easy way: delegate to programmer (`__attribute__((always_inline))`)
- Otherwise: very difficult problem

- Problem 1: ordering
  - Inlining big `a()` into `b()` might make `b()` too big to inline
    but inlining `b()` into `c()` might provide more opportunities
- Problem 2: cost model and thresholds
  - Estimate cost of inlining, e.g. additional code size, more branches
  - Estimate optimization gains and hotness
  - Some attempts to use machine learning here

Inlining Transformation

- Copy original function in place of the call
  - Split basic block containing function call
  - Needs map of old to new value
  - Constant fold/simplify instructions along the way
  - Replace constant conditional branches $\rightsquigarrow$ fewer copying

Inlining Transformation

- Copy original function in place of the call
  - Split basic block containing function call
  - Needs map of old to new value
  - Constant fold/simplify instructions along the way
  - Replace constant conditional branches $\rightsquigarrow$ fewer copying
- Replace returns with branches and ϕ-node to/at continuation point

Inlining Transformation

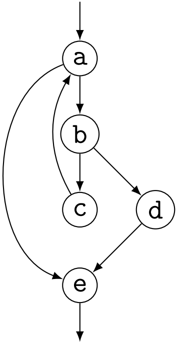
- Copy original function in place of the call
  - Split basic block containing function call
  - Needs map of old to new value
  - Constant fold/simplify instructions along the way
  - Replace constant conditional branches $\rightsquigarrow$ fewer copying
- Replace returns with branches and ϕ-node to/at continuation point
- Move static `alloca` to beginning
- Dynamic `alloca`: save/restore stack pointer
  - Prevent unbounded stack growth in loops
  - LLVM provides `stacksave`/`stackrestore` intrinsics
- Exceptions may need special treatment

What is a Loop?

```
void func() {
  while (a()) {
    if (b()) {
      d();
      break;
    }
    c();
  }
  e();
}
```

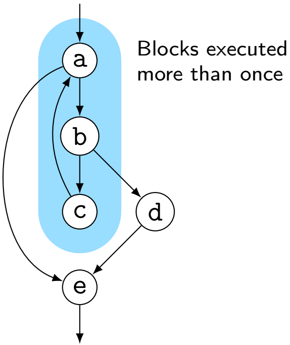
What is a Loop?

```
void func() {
  while (a()) {
    if (b()) {
      d();
      break;
    }
    c();
  }
  e();
}
```



What is a Loop?

```
void func() {
  while (a()) {
    if (b()) {
      d();
      break;
    }
    c();
  }
  e();
}
```



- Loops in source code $\neq$ loops in CFG
- `d` is *not* part of loop: executed at most once

$\rightsquigarrow$ Need algorithm to find loops in CFG

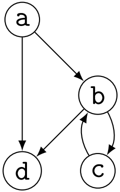
Loops

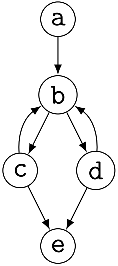
- Loop: maximal SCC L with at least one internal edge[27]
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$

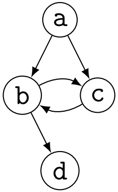
[27] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567. .

Loops

- Loop: maximal SCC L with at least one internal edge[27]
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$



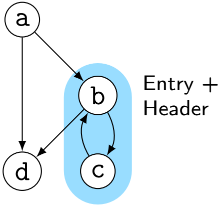


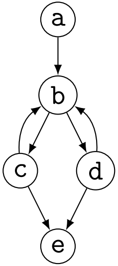


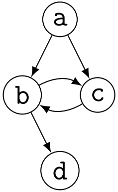
[27] P Havlak. “Nesting of reducible and irreducible loops”. In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[27]
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$



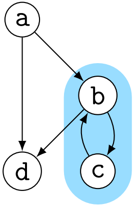




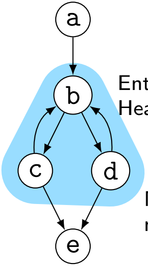
[27] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[27]
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$

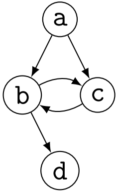


Entry +
Header



Entry +
Header

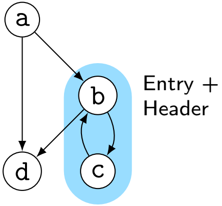
Note: $\{b, c\}$ alone is
no loop, not maximal

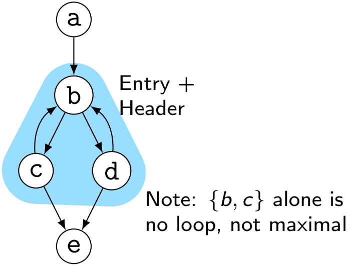


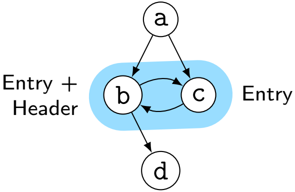
[27] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[27]
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$





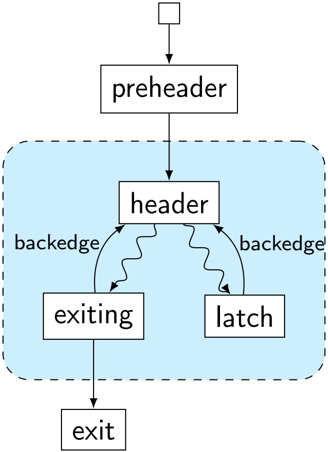


[27] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Natural Loops

- Natural Loop: loop with single entry
  - $\Rightarrow$ Header is unique
  - $\Rightarrow$ Header dominates all block
  - $\Rightarrow$ Loop is reducible

- Backedge: edge from block to header
- Predecessor: block with edge into loop
- Preheader: unique predecessor



Formal Definition

Loop L is reducible iff $\exists h \in L \,.\, \forall n \in L \,.\, h \text{ dom } n$

CFG is reducible iff all loops are reducible

Finding Natural Loops

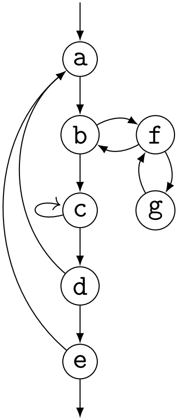
- Modified version[28] of Tarjan's algorithm[29]

- Iterate over dominator tree in post order
- Each block: find predecessors dominated by the block
  - None $\rightsquigarrow$ no loop header, continue
  - Any $\rightsquigarrow$ loop header, these edges *must* be backedges
- Walk through predecessors until reaching header again
  - All blocks on the way must be part of the loop body
  - Might encounter nested loops, update loop parent

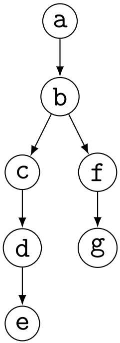
[28] G Ramalingam. "Identifying loops in almost linear time". In: *TOPLAS* 21.2 (1999), pp. 175–188.
[29] R Tarjan. "Testing flow graph reducibility". In: *STOC*. 1973, pp. 96–107.

Finding Natural Loops: Example

Control Flow Graph



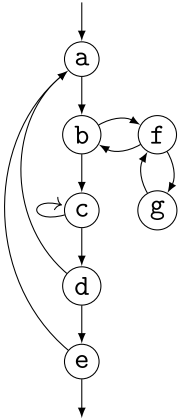
Dominator Tree



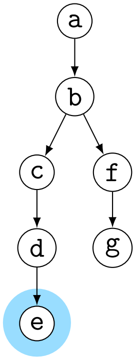
Loop Info

Finding Natural Loops: Example

Control Flow Graph



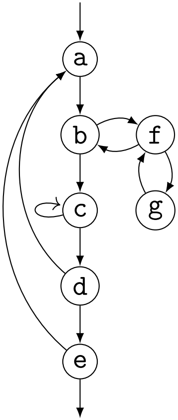
Dominator Tree



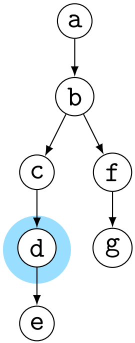
Loop Info

Finding Natural Loops: Example

Control Flow Graph



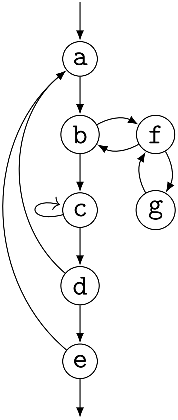
Dominator Tree



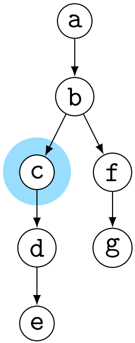
Loop Info

Finding Natural Loops: Example

Control Flow Graph



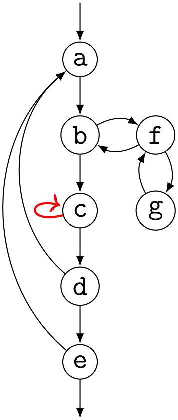
Dominator Tree



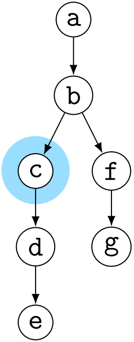
Loop Info

Finding Natural Loops: Example

Control Flow Graph



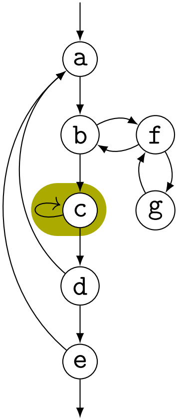
Dominator Tree



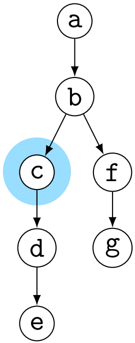
Loop Info

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



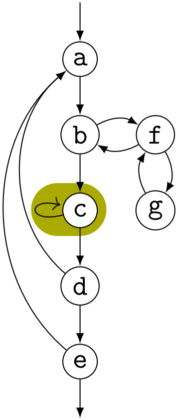
Loop Info

Loop A: {c}

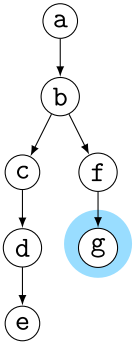
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



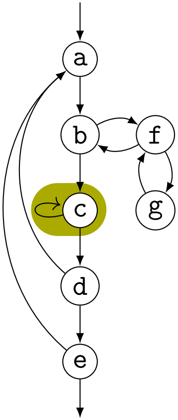
Loop Info

Loop A: {c}

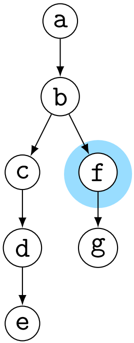
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



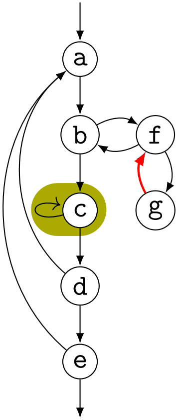
Loop Info

Loop A: `{c}`

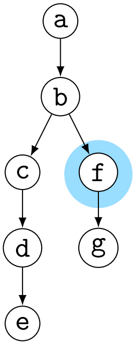
header: `c`; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



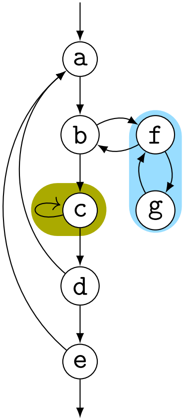
Loop Info

Loop A: {c}

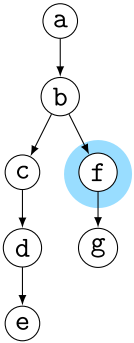
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: `{c}`

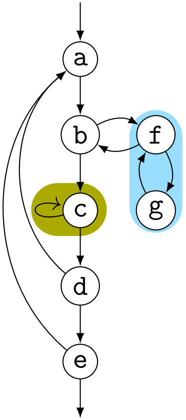
header: `c`; parent: NULL

Loop B: `{f,g}`

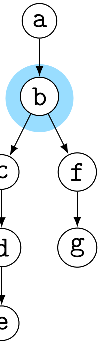
header: `f`; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

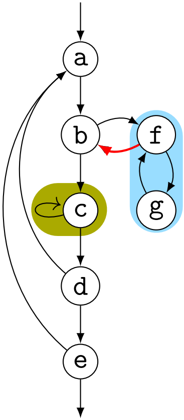
header: c; parent: NULL

Loop B: {f,g}

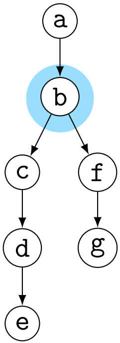
header: f; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

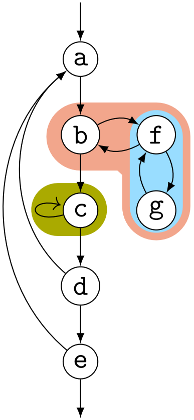
header: c; parent: NULL

Loop B: {f,g}

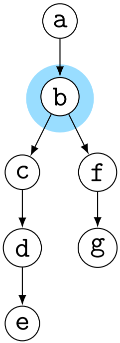
header: f; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: `{c}`

header: `c`; parent: NULL

Loop B: `{f,g}`

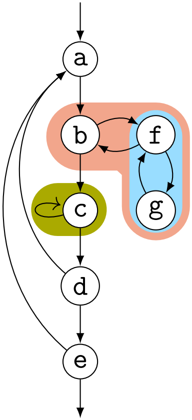
header: `f`; parent: C

Loop C: `{b,f,g}`

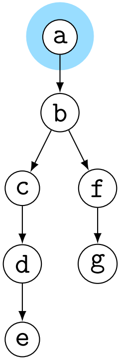
header: `b`; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

header: c; parent: NULL

Loop B: {f,g}

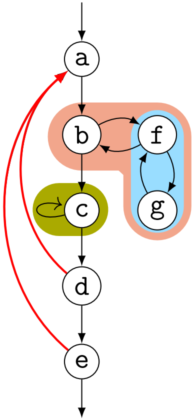
header: f; parent: C

Loop C: {b,f,g}

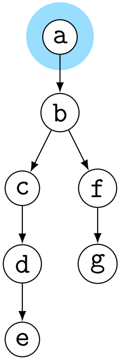
header: b; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

header: c; parent: NULL

Loop B: {f,g}

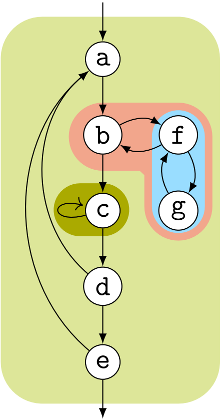
header: f; parent: C

Loop C: {b,f,g}

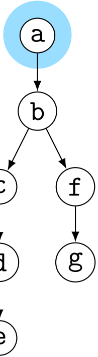
header: b; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: `{c}`

header: `c`; parent: D

Loop B: `{f,g}`

header: `f`; parent: C

Loop C: `{b,f,g}`

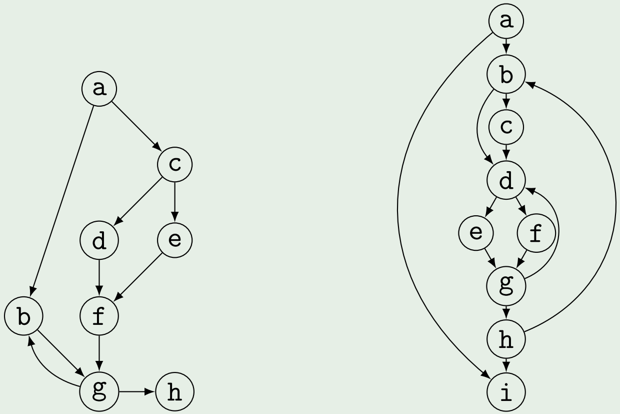
header: `b`; parent: D

Loop D: `{a,b,c,d,e,f,g}`

header: `a`; parent: NULL

Loop Analysis – Example

Apply the previous algorithm to find loops in the following CFGs (entry at a):



Loop Invariant Code Motion (LICM)

- Analyze loops, iterate over loop tree in post-order
  - I.e., visit inner loops first

[30] https://github.com/bytecodealliance/wasmtime/blob/bd6fe11/cranelift/codegen/src/licm.rs

Loop Invariant Code Motion (LICM)

- Analyze loops, iterate over loop tree in post-order
  - I.e., visit inner loops first

↑ Hoist:[30] iterate over blocks of loop in reverse post-order
  - For each movable inst., check for loop-defined operands
  - If not, move to preheader (create one, if not existent)
  - Otherwise, add inst. to set of values defined inside loop

[30] `https://github.com/bytecodealliance/wasmtime/blob/bd6fe11/cranelift/codegen/src/licm.rs`

Loop Invariant Code Motion (LICM)

- Analyze loops, iterate over loop tree in post-order
  - I.e., visit inner loops first

↑ Hoist:[30] iterate over blocks of loop in reverse post-order

- For each movable inst., check for loop-defined operands
- If not, move to preheader (create one, if not existent)
- Otherwise, add inst. to set of values defined inside loop

↓ Sink: Iterate over blocks of loop in post-order

- For each movable inst., check for users inside loop
- If none, move to unique exit (if existent)

[30] `https://github.com/bytecodealliance/wasmtime/blob/bd6fe11/cranelift/codegen/src/licm.rs`

Loop Unswitch

- Hoist loop-invariant conditionals outside loop

```
while (...)
  if (loopInvariantCond)
    A
  else
    B
  C
```

$\rightarrow$

```
if (loopInvariantCond)
  while (...)
    A
    C
else
  while (...)
    B
    C
```

Loop Unswitch

- Hoist loop-invariant conditionals outside loop

```
while (...)
  if (loopInvariantCond)
    A
  else
    B
  C
```

$\rightarrow$

```
if (loopInvariantCond)
  while (...)
    A
    C
else
  while (...)
    B
    C
```

- Might duplicate parts of the loop $\rightsquigarrow$ needs heuristics
- Might lead to exponential code size increase

Loop Rotation

- Rotate loop condition to bottom
  - Essentially, convert `while{}` to `if{do{}while}`

```
define void @src() {
  ; ...
loopheader:
  %iv = phi ...
  %cond = ...
  br i1 %cond, %body, %exit
body:
  ; ...
  br %loopheader
exit: ; ...
}
```

$\rightarrow$

```
define void @tgt() {
  ; ...
  %cond1 = ...
  br i1 %cond1, %body, %exit
body:
  %iv = phi ...
  ; ...
  %cond2 = ...
  br i1 %cond2, %body, %exit
exit: ; ...
}
```

Loop Strength Reduction

- Loop induction variables (e.g., loop counters) are often multiplied
  - E.g., for array indexing
- Strength reduction: replace "strong" multiply with "weak" addition

- Needs: analysis of loop induction variables and their recurrence
- Rewrite through replaced/extra loop induction variable
  - E.g., replace scaled index operation with pointer addition

- Information about ISA addressing modes beneficial

Analyses and Transformations – Summary

- Program Transformation critical for performance improvement
- Code not necessarily better
- Analyses are important to drive transformations
  - Dominator tree, loop detection, value liveness
- Important optimizations
  - Dead code elimination, sparse conditional constant propagation common sub-expression elimination, loop-invariant code motion, loop unswitching, loop rotation, loop strength reduction

Analyses and Transformations – Questions

- Why is "optimization" a misleading name for a transformation?
- How to find unused code sections in a function's CFG?
- Why is a liveness-based DCE better than a simple, user-based DCE?
- Why is SCCP more powerful than separate DCE/constant prop.?
- What is a dominator tree useful for?
- What is the difference between an irreducible and a natural loop?
- How to find natural loops in a CFG?
- How does the algorithm handle irreducible loops?
- Why is sinking a loop-invariant inst. harder than hoisting?