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 2022/23

Program Transformation: Motivation

- "User code" is often not very efficient

[9] 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

[9] 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*

[9] 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

[9] 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[9]

[9] 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
   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: 1   %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
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: 1       %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)

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

```
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
}
```

Work list (stack)

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)

br_1

br_2

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)

- $\texttt{br}_1$
- $\texttt{br}_2$
- $\texttt{br}_3$
- %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)

br_1

br_2

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)

br_1

br_2

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)

br_1

br_2

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 @foo(i32 %0, ptr %1, ptr %2) {
  %4 = zext i32 %0 to i64
  %5 = getelementptr inbounds i32, ptr %1, i64 %4
  %6 = load i32, ptr %5, align 4
  %7 = zext i32 %0 to i64
  %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
 %6 = load i32, ptr %5, align 4
 %7 = zext i32 %0 to i64
 %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
 %6 = load i32, ptr %5, align 4
 %7 = zext i32 %0 to i64
 %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
         %6 = load i32, ptr %5, align 4
         %7 = zext i32 %0 to i64
         %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
→ ht   %6 = load i32, ptr %5, align 4
       %7 = zext i32 %0 to i64
       %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
→ ht          %6 = load i32, ptr %5, align 4
dup %4        %7 = zext i32 %0 to i64
              %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4`
→ ht	`%6 = load i32, ptr %5, align 4`
dup %4	`%7 = zext i32 %0 to i64`
	`%8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
→ ht      %6 = load i32, ptr %5, align 4
dup %4    %7 = zext i32 %0 to i64
→ ht      %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
→ ht     %6 = load i32, ptr %5, align 4
dup %4   %7 = zext i32 %0 to i64
→ ht     %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
→ ht          %6 = load i32, ptr %5, align 4
dup %4        %7 = zext i32 %0 to i64
→ ht          %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
→ ht       %6 = load i32, ptr %5, align 4
dup %4     %7 = zext i32 %0 to i64
→ ht       %8 = getelementptr inbounds 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 inbounds i32, ptr %1, i64 %4
→ ht      %6 = load i32, ptr %5, align 4
dup %4    %7 = zext i32 %0 to i64
→ ht      %8 = getelementptr inbounds 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

→ ht

→ 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
→ 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:
  idom $(n) = d : d$ sdom $n \wedge \nexists d'.d$ sdom $d' \wedge d'$ 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:
  $\text{idom } (n) = d : d \text{ sdom } n \wedge \nexists d'.d \text{ sdom } d' \wedge d' \text{ sdom } n$

⇒ All strict dominators are always executed before the block

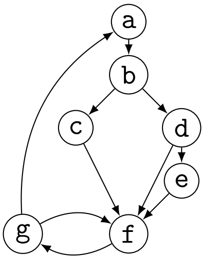
⇒ All values from dominators available/usable

⇒ 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

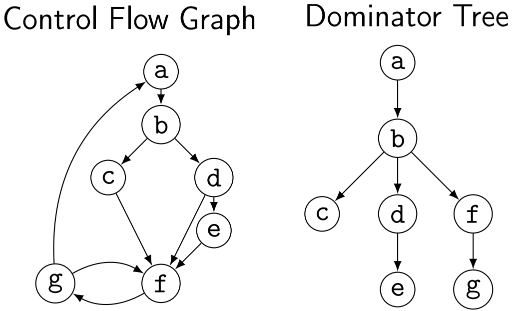
Control Flow Graph



Dominator Tree

Dominator Tree

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



Dominator Tree: Construction

- Naive: inefficient (but reasonably simple)[10]
  - 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

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

Dominator Tree: Construction

- Naive: inefficient (but reasonably simple)[10]
  - 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[11]
  - 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 frequently in compilers[12]

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

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

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

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[13]
  - $a.preNum < b.preNum \ \wedge \ a.postNum > b.postNum$
  - After updates, numbers might be invalid: recompute or walk tree

[13] 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[13]
  - $a.preNum < b.preNum \ \wedge \ a.postNum > b.postNum$
  - After updates, numbers might be invalid: recompute or walk tree
- Problem: dominance of unreachable blocks ill-defined $\rightsquigarrow$ special handling

[13] 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:[14]
  - Do a DFS over dominator tree
  - Use scoped hashmap to track available values

Does this work?

[14] 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:[14]
  - Do a DFS over dominator tree
  - Use scoped hashmap to track available values

Does this work? Yes.

[14] 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
  - 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
  - Use pointer or index for instruction result operands
- Canonicalize commutative operations
  - Order operands deterministically, e.g., by address
- Identities: `a+(b+c)` vs. `(a+b)+c`

Global Value Numbering – or: advanced CSE

- Hash-based approach only catches trivially removable duplicates

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

Global Value Numbering – or: advanced CSE

- 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[15]

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

Simple Transformations: Inlining

- Estimate whether inlining is beneficial
  - Savings of avoided call/computations/branches; cost of increased size

Simple Transformations: Inlining

- Estimate whether inlining is beneficial
  - Savings of avoided call/computations/branches; cost of increased size
- Copy original function in place of the call
  - Split basic block containing function call

Simple Transformations: Inlining

- Estimate whether inlining is beneficial
  - Savings of avoided call/computations/branches; cost of increased size
- Copy original function in place of the call
  - Split basic block containing function call
- Replace returns with branches and ϕ-node to/at continuation point

Simple Transformations: Inlining

- Estimate whether inlining is beneficial
  - Savings of avoided call/computations/branches; cost of increased size
- Copy original function in place of the call
  - Split basic block containing function call
- Replace returns with branches and ϕ-node to/at continuation point
- Move `alloca` to beginning or save stack pointer
  - Prevent unbounded stack growth in loops
  - LLVM provides `stacksave`/`stackrestore` intrinsics
- Exceptions may need special treatment

Simple Transformations: Mem2Reg and SROA

- Mem2reg: promote `alloca` to SSA values/phis
  - Condition: only `load`/`store`, no address taken
  - Essentially just SSA construction
- SROA: scalar replacement of aggregate
  - Separate structure fields into separate variables
  - Also promote them to SSA

Loops

- Loop: maximal SCC L with at least one internal edge[16]

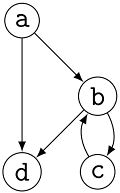
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$

[16] 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[16]

  (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 $L \setminus \{h\}$

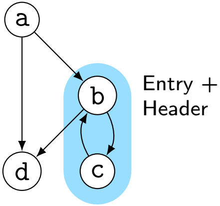


[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[16]

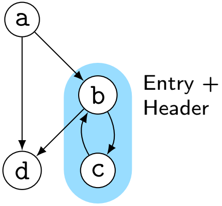
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$

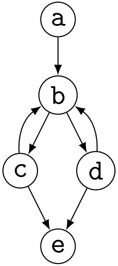


[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$

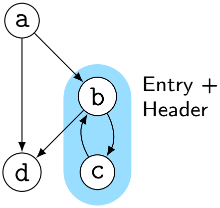


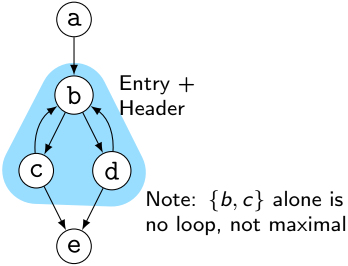


[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$

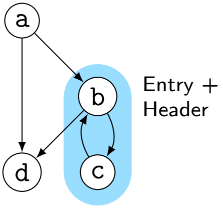


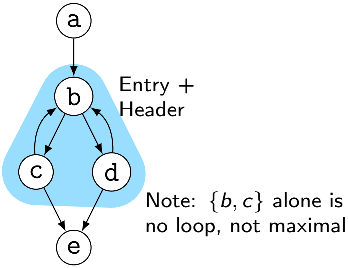


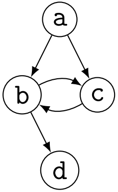
[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$



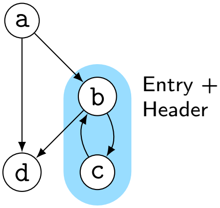


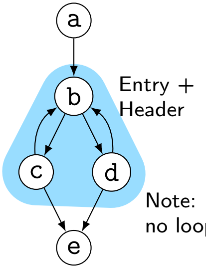


[16] P Havlak. “Nesting of reducible and irreducible loops”. In: *TOPLAS* 19.4 (1997), pp. 557–567.

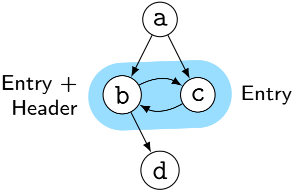
Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$





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



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

Natural Loops

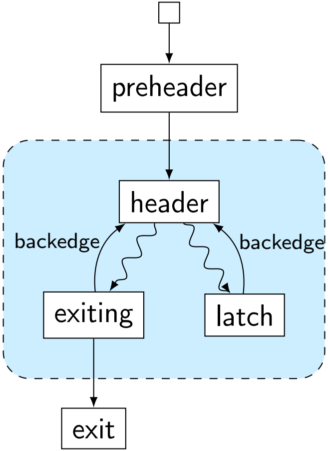
- Natural Loop: loop with single entry
  - ⇒ Header is unique
  - ⇒ Header dominates all block
  - ⇒ Loop is reducible

- 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[17] of Tarjan's algorithm[18]

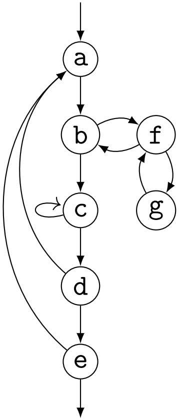
- Iterate over dominator tree in post order
- Each block: find predecessors dominated by the block
  - None $\rightsquigarrow$ no loop header, continue
  - Any $\rightsquigarrow$ loop header, these edges *must* be backedges
- Walk through predecessors until reaching header again
  - All blocks on the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

[17] G Ramalingam. "Identifying loops in almost linear time". In: *TOPLAS* 21.2 (1999), pp. 175–188. .

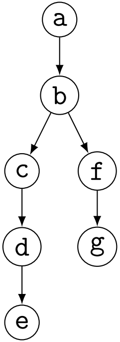
[18] R Tarjan. "Testing flow graph reducibility". In: *STOC*. 1973, pp. 96–107. .

Finding Natural Loops: Example

Control Flow Graph



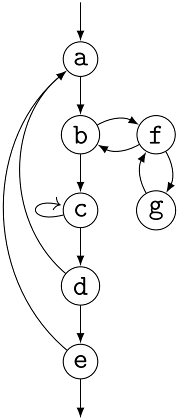
Dominator Tree



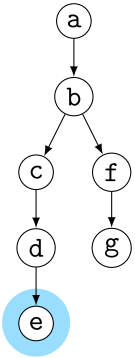
Loop Info

Finding Natural Loops: Example

Control Flow Graph



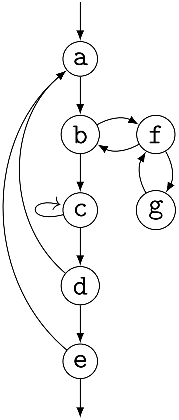
Dominator Tree



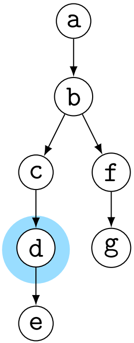
Loop Info

Finding Natural Loops: Example

Control Flow Graph



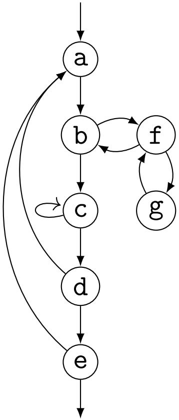
Dominator Tree



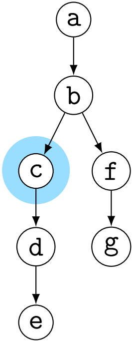
Loop Info

Finding Natural Loops: Example

Control Flow Graph



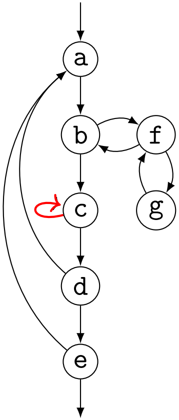
Dominator Tree



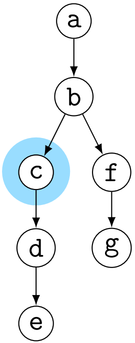
Loop Info

Finding Natural Loops: Example

Control Flow Graph



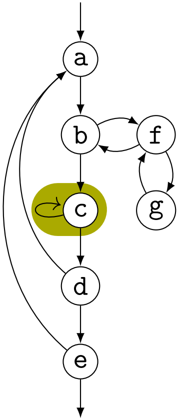
Dominator Tree



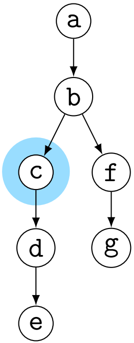
Loop Info

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



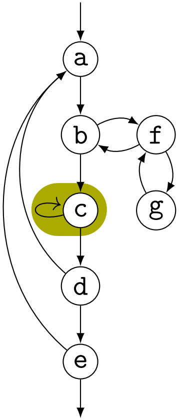
Loop Info

Loop A: {c}

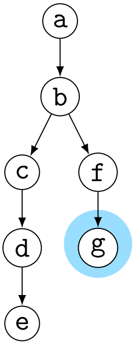
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



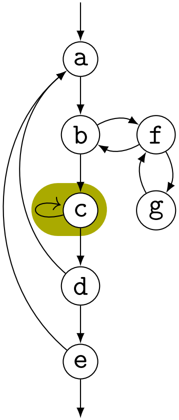
Loop Info

Loop A: {c}

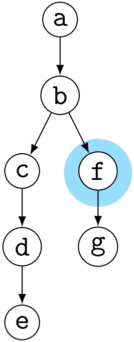
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



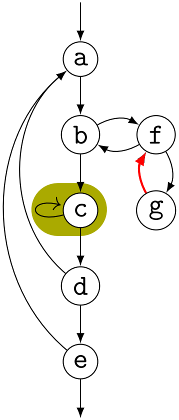
Loop Info

Loop A: {c}

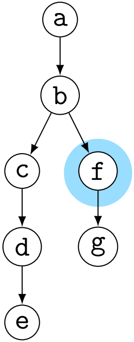
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



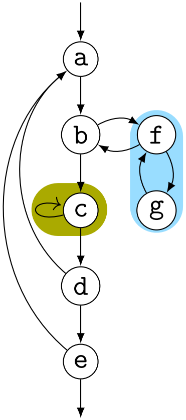
Loop Info

Loop A: {c}

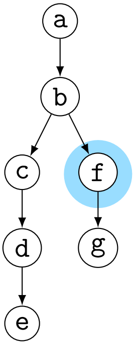
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

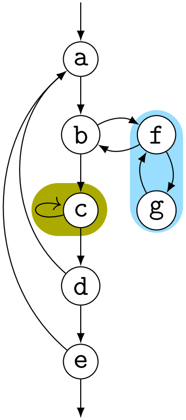
header: c; parent: NULL

Loop B: {f,g}

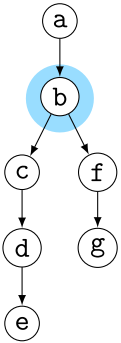
header: f; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

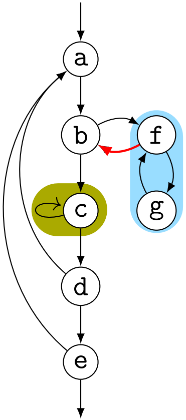
header: c; parent: NULL

Loop B: {f,g}

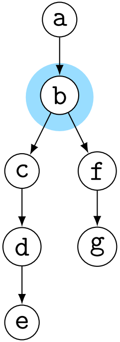
header: f; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

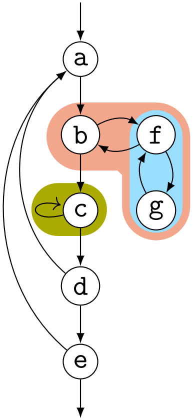
header: c; parent: NULL

Loop B: {f,g}

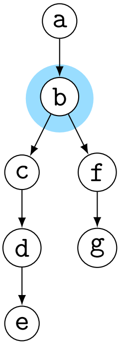
header: f; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

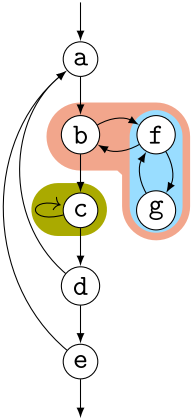
Loop A: `{c}`
header: `c`; parent: NULL

Loop B: `{f,g}`
header: `f`; parent: C

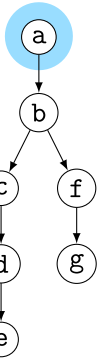
Loop C: `{b,f,g}`
header: `b`; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

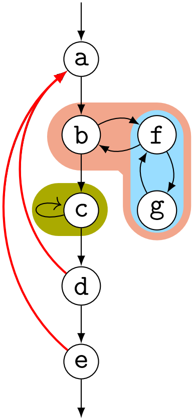
Loop A: {c}
header: c; parent: NULL

Loop B: {f,g}
header: f; parent: C

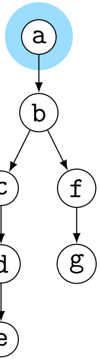
Loop C: {b,f,g}
header: b; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

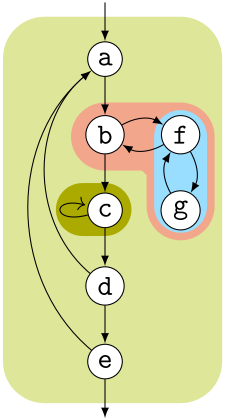
Loop A: `{c}`
header: `c`; parent: NULL

Loop B: `{f,g}`
header: `f`; parent: C

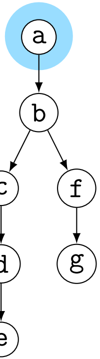
Loop C: `{b,f,g}`
header: `b`; parent: NULL

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 Invariant Code Motion (LICM)

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

[19] 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:[19] 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

[19] 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:[19] 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)

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

Transformations and Analyses in LLVM: Passes

- Transformations and analyses organized in *passes*
- Pass can operate on Module/(CGSCC)/Function/Loop

Transformations and Analyses in LLVM: Passes

- Transformations and analyses organized in *passes*
- Pass can operate on Module/(CGSCC)/Function/Loop

- Analysis pass: takes input IR and returns analysis result
  - May also use results of other analyses; results are cached
- Transformation pass: takes input IR and returns preserved analyses
  - Can use analyses, which are re-run when outdated

Transformations and Analyses in LLVM: Passes

- Transformations and analyses organized in *passes*
- Pass can operate on Module/(CGSCC)/Function/Loop

- Analysis pass: takes input IR and returns analysis result
  - May also use results of other analyses; results are cached
- Transformation pass: takes input IR and returns preserved analyses
  - Can use analyses, which are re-run when outdated

- Pass manager executes passes on same granularity
  - Otherwise, use adaptor: `createFunctionToLoopPassAdaptor` (and preferably combine multiple smaller passes into a separate pass manager)

Using LLVM (New) Pass Manager

```
void optimize(llvm::Function* fn) {
  llvm::PassBuilder pb;
  llvm::LoopAnalysisManager lam{};
  llvm::FunctionAnalysisManager fam{};
  llvm::CGSCCAnalysisManager cgam{};
  llvm::ModuleAnalysisManager mam{};
  pb.registerModuleAnalyses(mam);
  pb.registerCGSCCAnalyses(cgam);
  pb.registerFunctionAnalyses(fam);
  pb.registerLoopAnalyses(lam);
  pb.crossRegisterProxies(lam, fam, cgam, mam);

  llvm::FunctionPassManager fpm{};
  fpm.addPass(llvm::DCEPass());
  fpm.addPass(llvm::createFunctionToLoopPassAdaptor(llvm::LoopRotatePass()));
  fpm.run(*fn, fam);
}
```

Writing a Pass for LLVM's New PM – Part 1

```
#include "llvm/IR/PassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"

class TestPass : public llvm::PassInfoMixin<TestPass> {
public:
  llvm::PreservedAnalyses run(llvm::Function &F,
                              llvm::FunctionAnalysisManager &AM) {
    // Do some magic
    llvm::DominatorTree *DT = &AM.getResult<llvm::DominatorTreeAnalysis>(F);
    // ...
    llvm::errs() << F.getName() << "\n";
    return llvm::PreservedAnalyses::all();
  }
};
// ...
```

Writing a Pass for LLVM's New PM – Part 2

```
extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
  return { LLVM_PLUGIN_API_VERSION, "TestPass", "v1",
    [] (llvm::PassBuilder &PB) {
      PB.registerPipelineParsingCallback(
        [] (llvm::StringRef Name, llvm::FunctionPassManager &FPM,
            llvm::ArrayRef<llvm::PassBuilder::PipelineElement>) {
          if (Name == "testpass") {
            FPM.addPass(TestPass());
            return true;
          }
          return false;
        });
    } };
}
```

```
c++ -shared -o testpass.so testpass.cc -lLLVM -fPIC
opt -load-pass-plugin=$PWD/testpass.so -passes=testpass input.ll | llvm-dis
```

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, common sub-expression elimination, loop-invariant code motion
- Compilers often implement transformations as passes
- Analyses may be invalidated by transformations, needs tracking

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?
- 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?