Eliminating Redundant Redundancy

April 11, 2011
CS 132: Compiler Design

7. Most travel will be accomplished using the University Credit Card. Certain cases will require a cash advance that can be obtained at an ATM machine by using your personal pin number.

Optimizations to Remove Repeated Computations

- ✓ Loop-Invariant Code Motion
- ✓ Value Numbering
- ✓ Common Subexpression Elimination (CSE)
- ✓ Partial Redundancy Elimination (PRE)

Value Numbering

Applied most often to (extended) basic blocks.

Can be made "global," but algorithms become more complicated (e.g., requiring SSA form).

Value Numbering

Applied most often to (extended) basic blocks.

Can be made "global," but algorithms become more complicated (e.g., requiring SSA form).

Main idea: associate a (numeric) identifier with

- each variable
- each expression computed so far (in terms of the numbers of its operands).

Example

Show how value numbering finds redundant computations in the following pieces of code:

```
a ← x + y
a ← a + b
b ← x + y
c ← a + b
```

```
g ← x + y
h ← u - v
i ← x + y
x ← u - v
u ← g + h
v ← i + x
w ← u + v
```

Available Expressions

We say that the statement

$$a \leftarrow b + c$$

generates the expression **b+c** and *kills* all expressions involving **a**.

Available Expressions

We say that the statement

```
a ← b + c
```

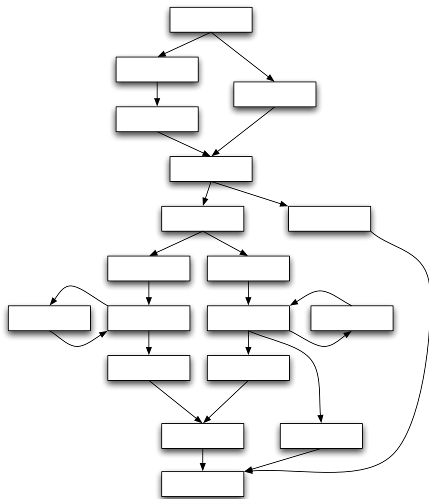
generates the expression **b+c** and *kills* all expressions involving **a**.

$$\begin{aligned} \mathit{AvailIn}(I) &= \bigcap_{P \in \mathit{Pred}(I)} \mathit{AvailOut}(P) \\ \mathit{AvailOut}(I) &= (\mathit{AvailIn}(I) \cup \mathit{ExprsUsed}(I)) \setminus \mathit{ExprsKilled}(I) \end{aligned}$$

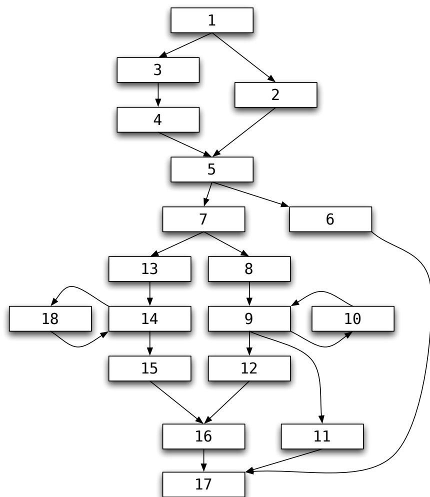
Speeding up Iterative Analysis (1)

Process the blocks in a good order.
E.g., Depth-First

- ✓ Successively visit (unvisited) children
- ✓ Then, number counting down from N



Speeding up Iterative Analysis (1)

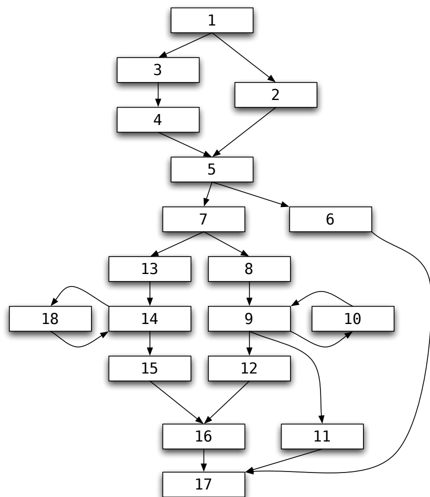


For available expressions, should we work from in depth-first order, or in reverse of the depth-first order?

$$\mathit{in}(\mathrm{I}) = \bigcap_{\mathrm{p} \in \mathit{Pred}(\mathrm{I}_1)} \mathit{out}(\mathrm{p})$$

$$\mathit{out}(\mathrm{I}) = (\mathit{in}(\mathrm{I}) \cup \mathit{gen}(\mathrm{I})) \setminus \mathit{kill}(\mathrm{I})$$

SPEEDING UP ITERATIVE ANALYSIS (1)



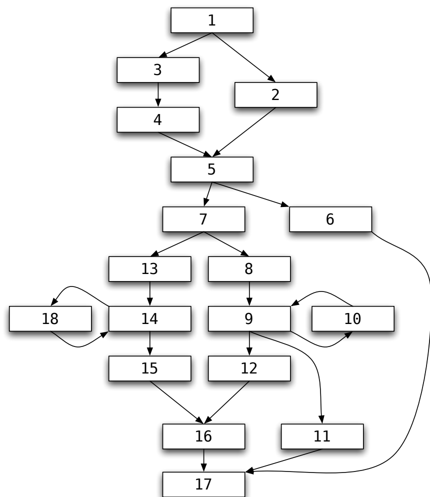
For available expressions, should we work from in depth-first order, or in reverse of the depth-first order?

$$in(\mathrm{I}) = \bigcap_{p \in Pred(\mathrm{I}_1)} out(p)$$

$$out(\mathrm{I}) = (in(\mathrm{I}) \cup gen(\mathrm{I})) \setminus kill(\mathrm{I})$$

✓ This is a *forwards* analysis

Speeding up Iterative Analysis (1)



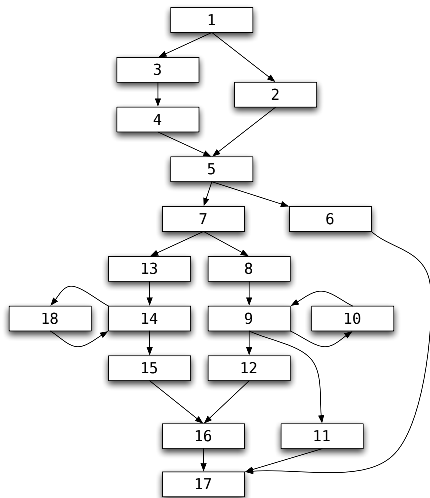
For available expressions, should we work from in depth-first order, or in reverse of the depth-first order?

$$in(\mathrm{I}) = \bigcap_{\mathrm{p} \in Pred(\mathrm{I}_1)} out(\mathrm{p})$$

$$out(\mathrm{I}) = (in(\mathrm{I}) \cup gen(\mathrm{I})) \setminus kill(\mathrm{I})$$

- ✓ This is a *forwards* analysis
- ✓ Liveness is a *backwards* analysis

Speeding up Iterative Analysis (1)



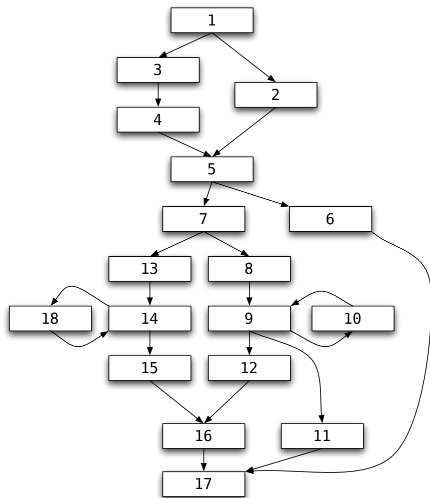
For available expressions, should we work from in depth-first order, or in reverse of the depth-first order?

$$in(I) = \bigcap_{p \in Pred(I_1)} out(p)$$

$$out(I) = (in(I) \cup gen(I)) \setminus kill(I)$$

- ✓ This is a *forwards* analysis
- ✓ Liveness is a *backwards* analysis
- ✓ For some problems, information flows in both directions (*bidirectional* analysis)

Speeding up Iterative Analysis (1)



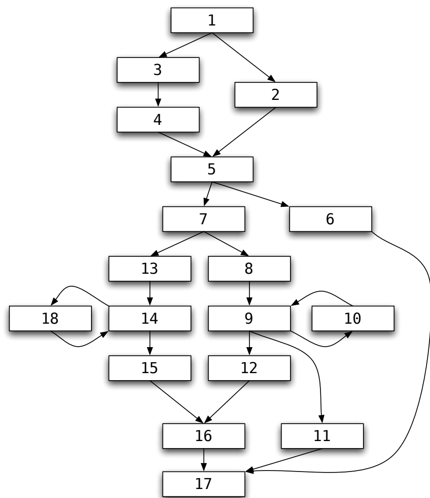
For available expressions, should we work from in depth-first order, or in reverse of the depth-first order?

$$in(I) = \bigcap_{p \in Pred(I_1)} out(p)$$

$$out(I) = (in(I) \cup gen(I)) \setminus kill(I)$$

- ✓ This is a *forwards* analysis
- ✓ Liveness is a *backwards* analysis
- ✓ For some problems, information flows in both directions (*bidirectional* analysis)
  - No good order, so slower

Speeding up Iterative Analysis (1)



For available expressions, should we work from in depth-first order, or in reverse of the depth-first order?

$$in(\mathrm{I}) = \bigcap_{\mathrm{p} \in Pred(\mathrm{I}_1)} out(\mathrm{p})$$

$$out(\mathrm{I}) = (in(\mathrm{I}) \cup gen(\mathrm{I})) \setminus kill(\mathrm{I})$$

- ✓ This is a *forwards* analysis
- ✓ Liveness is a *backwards* analysis
- ✓ For some problems, information flows in both directions (*bidirectional* analysis)
  - No good order, so slower
  - Preferred: multiple unidirectional analyses

Speeding Up Iterative Analysis (2)

Compute *gen* and *kill* per-basic-block, rather than per-instruction.

$$\begin{aligned} gen(I_1; I_2) &= (gen(I_1) \setminus kill(I_1)) \cup gen(I_2) \\ kill(I_1; I_2) &= kill(I_1) \cup kill(I_2) \end{aligned}$$

Speeding Up Iterative Analysis (2)

Compute *gen* and *kill* per-basic-block, rather than per-instruction.

$$
\begin{aligned}
gen(I_1; I_2) &= (gen(I_1) \setminus kill(I_1)) \cup gen(I_2) \\
kill(I_1; I_2) &= kill(I_1) \cup kill(I_2)
\end{aligned}
$$

What does the following code generate and kill?

```
a ← b + c
b ← a - d
c ← b + c
```

Speeding up Iterative Analysis (3)

Represent the sets efficiently

Possible representations include:

- ✓ lists
- ✓ binary trees
- ✓ hash tables
- ✓ bit vectors

Local CSE

- ✓ Avoids duplication within a basic block.
- ✓ Often applied *during* code generation
  - Before real dataflow analysis

Local CSE

- Avoids duplication within a basic block.
- Often applied *during* code generation
  - Before real dataflow analysis
- Algorithm:
  - Keep track of expressions generated that haven't yet been killed.

Local CSE

- ✓ Avoids duplication within a basic block.
- ✓ Often applied *during* code generation
  - Before real dataflow analysis
- ✓ Algorithm:
  - Keep track of expressions generated that haven't yet been killed.
  - If `b+c` is available because of an instruction

    `a ← b + c`

    and the current instruction involves `b+c`,

Local CSE

- ✓ Avoids duplication within a basic block.
- ✓ Often applied *during* code generation
  - Before real dataflow analysis
- ✓ Algorithm:
  - Keep track of expressions generated that haven't yet been killed.
  - If `b+c` is available because of an instruction

    ```
    a ← b + c
    ```

    and the current instruction involves `b+c`,
    replace the above instruction by

    ```
    t ← b + c
    a ← t
    ```

    (why?) and then emit code referring to `t` instead of `b+c`.

Examples

Apply Local CSE to:

```
a ← x + y
a ← a + b
b ← x + y
c ← a + b
```

```
g ← x + y
h ← u - v
i ← x + y
x ← u - v
u ← g + h
v ← i + x
w ← u + v
```

Global CSE

Algorithm (assuming local CSE was already performed)

- ✓ Compute available expressions (per-block)
- ✓ For each expression used that was available at its block entry:
  - Check that it's still available where it's used (why?)
  - Search backwards for the place(s) the expression was defined
  - Add a new temporary, as in local CSE.

Global CSE

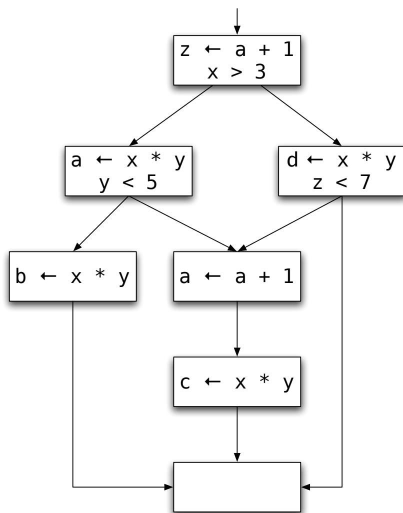
Algorithm (assuming local CSE was already performed)

- ✓ Compute available expressions (per-block)
- ✓ For each expression used that was available at its block entry:
  - Check that it's still available where it's used (why?)
  - Search backwards for the place(s) the expression was defined
  - Add a new temporary, as in local CSE.

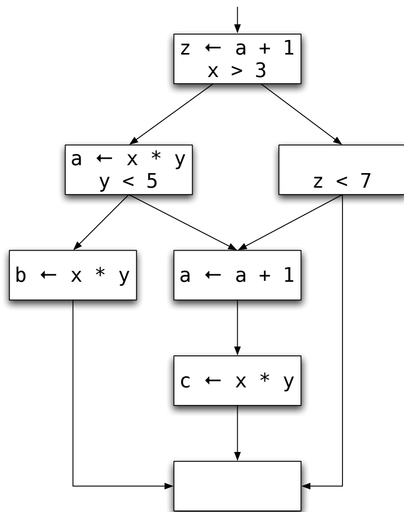
Or, do a more complicated dataflow analysis that tells you what's available *and* where it came from, and skip the search.

EXAMPLE

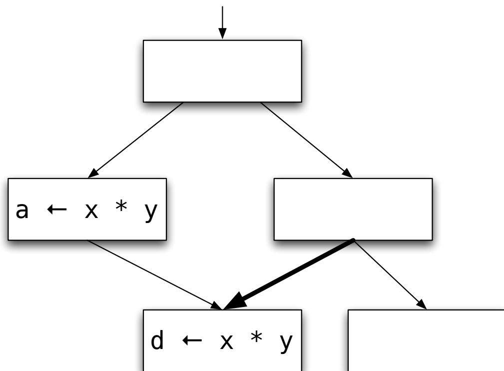
Perform Global CSE on the following program:



Partial Redundancy: An Example



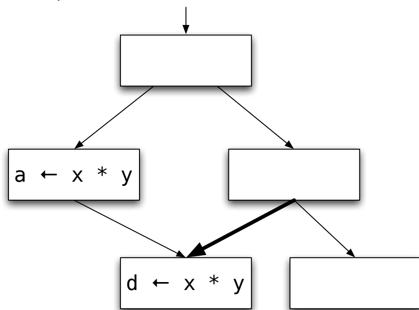
Complication: Critical Edges



Splitting Critical Edges

A critical edge is an edge

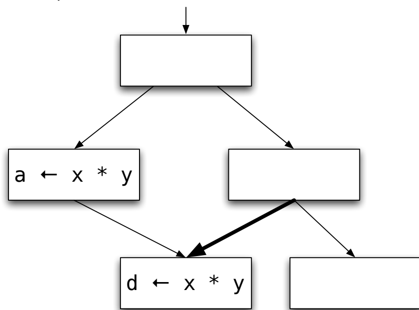
- ✓ from a node with multiple successors
- ✓ to a node with multiple predecessors



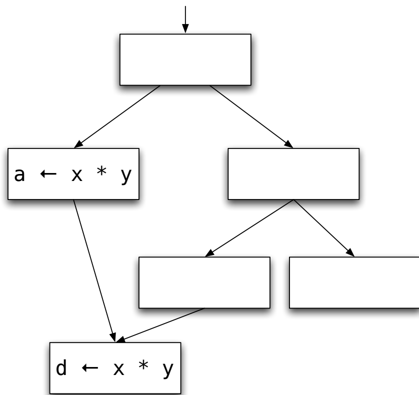
Splitting Critical Edges

A critical edge is an edge

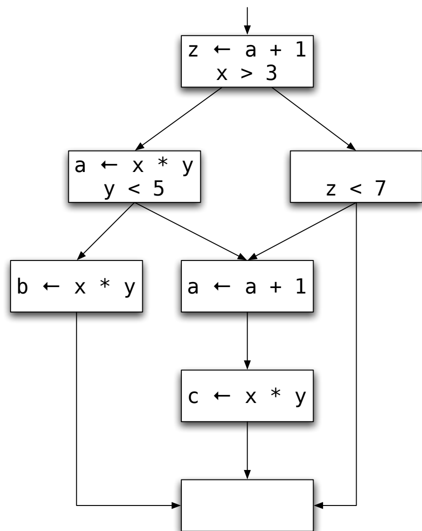
- ✓ from a node with multiple successors
- ✓ to a node with multiple predecessors



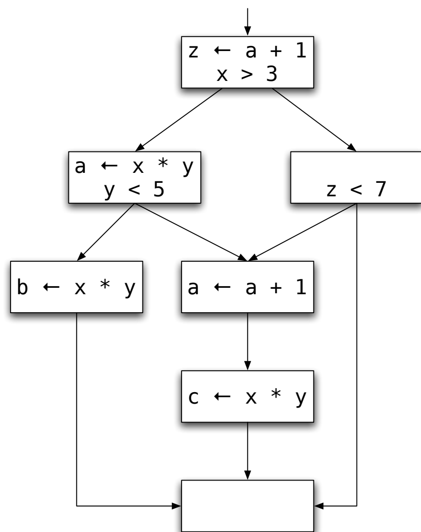
By adding extra nodes,
we can remove all critical edges.



Partial Redundancy Without Critical Edges

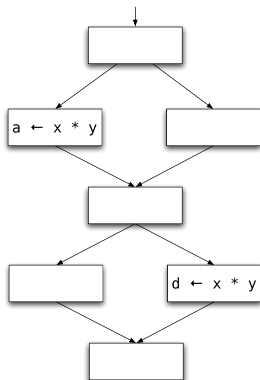


PARTIAL REDUNDANCY WITHOUT CRITICAL EDGES

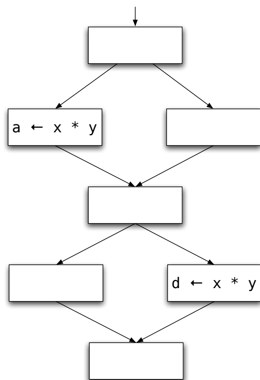


z ← a + 1
x > 3
a ← x * y
y < 5
z < 7
b ← x * y
a ← a + 1
c ← x * y

Removing All Redundancy Can Be Impractical



Removing *All* Redundancy Can Be Impractical



Can't remove all redundancy without duplicating bits of the control graph. In the worst case code blows up exponentially. So we'll just consider optimizations that modify the flowgraph *only* to remove critical edges.

Partial Redundancy Elimination (PRE)

Removes as many partial redundancies as possible

1. Without altering the flowgraph
2. Without adding "unnecessary" work to any path
3. Without moving work later.

PRE can hoist invariant code out of loops (since the path that *goes* around the loop has a redundancy)!

PRE can be applied to any flowgraph, but works better if we have already eliminated critical edges. (Some variants *require* no critical edges.)

Partial Redundancy Elimination (PRE)

Removes as many partial redundancies as possible

1. Without altering the flowgraph
2. Without adding "unnecessary" work to any path
3. Without moving work later.

PRE can hoist invariant code out of loops (since the path that *goes* around the loop has a redundancy)!

PRE can be applied to any flowgraph, but works better if we have already eliminated critical edges. (Some variants *require* no critical edges.)

Originally formulated as a bidirectional analysis.

A more efficient version:

1. Anticipated expressions (backwards)
2. Available expressions (forwards)

Anticipated Expressions

An expression is *anticipated* at a program point if it is guaranteed to be computed later (with unchanged operands).

$$\mathit{AntIn}[\mathrm{B}] = \mathit{ExprsUsed}[\mathrm{B}] \cup (\mathit{AntOut}[\mathrm{B}] \setminus \mathit{ExprsKilled}[\mathrm{B}])$$

$$\mathit{AntOut}[\mathrm{B}] = \bigcap_{\mathrm{S} \in \mathit{Succ}(\mathrm{B})} \mathit{AntIn}[\mathrm{S}]$$

Anticipated Expressions

An *expression* is *anticipated* at a program point if it is guaranteed to be computed later (with unchanged operands).

$$\mathit{AntIn}[\mathrm{B}] = \mathit{ExprsUsed}[\mathrm{B}] \cup (\mathit{AntOut}[\mathrm{B}] \setminus \mathit{ExprsKilled}[\mathrm{B}])$$

$$\mathit{AntOut}[\mathrm{B}] = \bigcap_{\mathrm{S} \in \mathit{Succ}(\mathrm{B})} \mathit{AntIn}[\mathrm{S}]$$

Why bother?

Anticipated Expressions

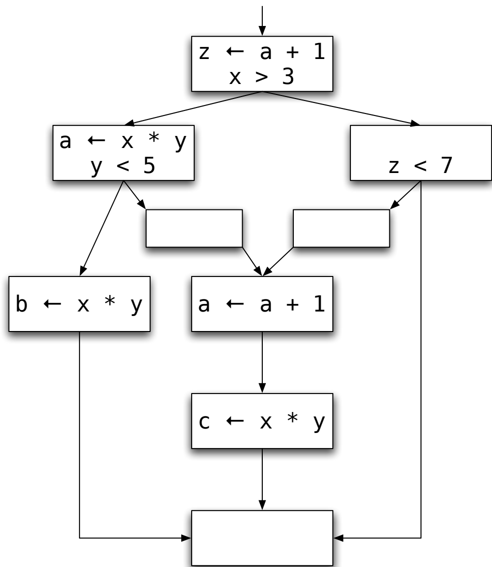
An *expression* is *anticipated* at a program point if it is guaranteed to be computed later (with unchanged operands).

$$\begin{aligned} \mathit{AntIn}[\mathrm{B}] &= \mathit{ExprsUsed}[\mathrm{B}] \cup (\mathit{AntOut}[\mathrm{B}] \setminus \mathit{ExprsKilled}[\mathrm{B}]) \\ \mathit{AntOut}[\mathrm{B}] &= \bigcap_{\mathrm{S} \in \mathit{Succ}(\mathrm{B})} \mathit{AntIn}[\mathrm{S}] \end{aligned}$$

Why bother?

We only want to compute expressions when (and only when) they are anticipated.

Anticipated In/Out?



Earliest Positioning (Motivation)

Simplest practical form of PRE:

- ✓ Compute each expression as soon as it becomes anticipated.

Earliest Positioning (Motivation)

Simplest practical form of PRE:

- ✓ Compute each expression as soon as it becomes anticipated.
  - Place computations as early as possible

Earliest Positioning (Motivation)

Simplest practical form of PRE:

- ✓ Compute each expression as soon as it becomes anticipated.
  - Place computations as early as possible
  - ...without introducing unnecessary computations on any path;

Earliest Positioning (Motivation)

Simplest practical form of PRE:

- ✓ Compute each expression as soon as it becomes anticipated.
  - Place computations as early as possible
  - ...without introducing unnecessary computations on any path;
  - Make redundant as many expressions as possible.

Earliest Positioning (Motivation)

Simplest practical form of PRE:

✓ Compute each expression as soon as it becomes anticipated.

- Place computations as early as possible
- ...without introducing unnecessary computations on any path;
- Make redundant as many expressions as possible.

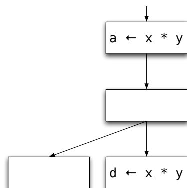
Earliest Positioning (Motivation)

Simplest practical form of PRE:

✓ Compute each expression as soon as it becomes anticipated.

- Place computations as early as possible
- …without introducing unnecessary computations on any path;
- Make redundant as many expressions as possible.

Careful: anticipation may oscillate:



Placing Expressions

To avoid partial redundancies, we want to move expressions only to positions where they are anticipated.

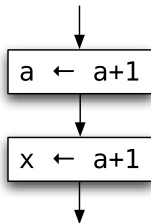
Tricky bit: of all the places where an expression is anticipated, which should actually evaluate the expression?

Placing Expressions

To avoid partial redundancies, we want to move expressions only to positions where they are anticipated.

Tricky bit: of all the places where an expression is anticipated, which should actually evaluate the expression?

Wrong Answer: whenever an expression changes from not-anticipated to anticipated.



A Method for Earliest Positioning (Intuition)

1. **Make sure we cover the right positions:**
   add a copy of each expression at the start of *every* block where it is anticipated!

2. **Eliminate resulting redundancy:**
   do "global" CSE.

Streamlining the Algorithm

1. Add anticipated expressions to blocks
2. Do CSE
   - 2.1 Compute available expressions (at the block level)
   - 2.2 Remove redundant computations in each block

Streamlining the Algorithm

1. Add anticipated expressions to blocks
2. Do CSE
   2.1 Compute available expressions (at the block level)
   2.2 Remove redundant computations in each block

First optimization: Combine 1 and 2(a) "hypothetically available expressions"

$$\mathit{HAvailIn}[\mathrm{B}] = \bigcap_{\mathrm{P} \in \mathit{Pred}(\mathrm{B})} \mathit{HAvailOut}[\mathrm{P}]$$

$$\mathit{HAvailOut}[\mathrm{B}] = (\mathit{HAvailIn}[\mathrm{B}] \cup \mathit{AntIn}[\mathrm{B}]) \setminus \mathit{ExprsKilled}(\mathrm{B})$$

Streamlining the Algorithm

1. Add anticipated expressions to blocks
2. Do CSE
   2.1 Compute available expressions (at the block level)
   2.2 Remove redundant computations in each block

First optimization: Combine 1 and 2(a) "hypothetically available expressions"

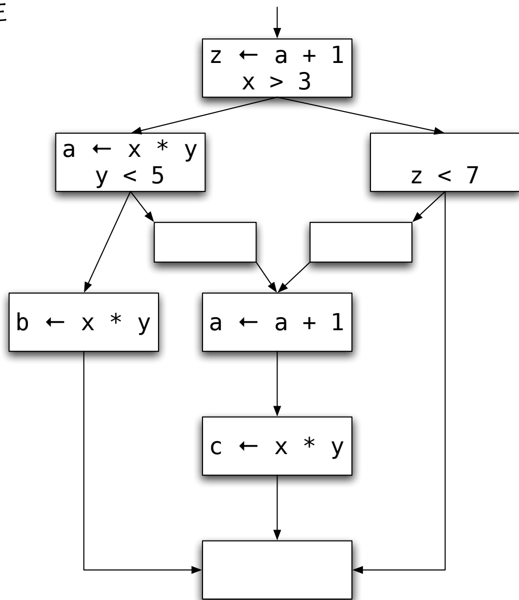
$$
\begin{aligned}
\mathit{HAvailIn}[\mathrm{B}] &= \bigcap_{\mathrm{P} \in \mathit{Pred}(\mathrm{B})} \mathit{HAvailOut}[\mathrm{P}] \\
\mathit{HAvailOut}[\mathrm{B}] &= (\mathit{HAvailIn}[\mathrm{B}] \cup \mathit{AntIn}[\mathrm{B}]) \setminus \mathit{ExprsKilled}(\mathrm{B})
\end{aligned}
$$

Simplification: directly compute the *non-redundant* expressions to keep

$$\mathit{EarliestIn}[\mathrm{B}] = \mathit{AntIn}[\mathrm{B}] \setminus \mathit{HAvailIn}[\mathrm{B}].$$

"expressions anticipated here, but not (even hypothetically) already available"

PRE Example

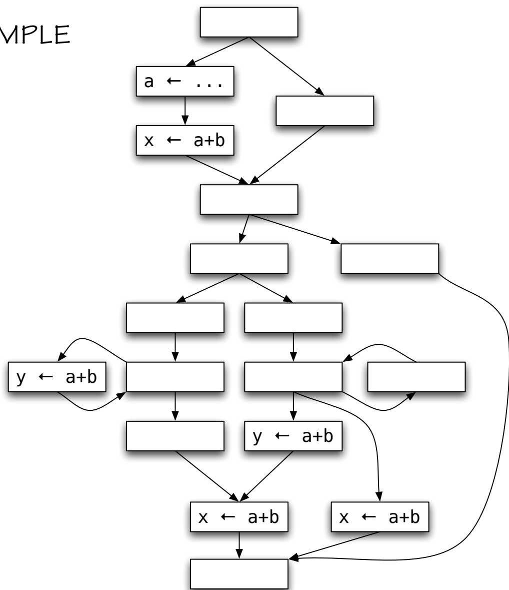


Towards Better PRE

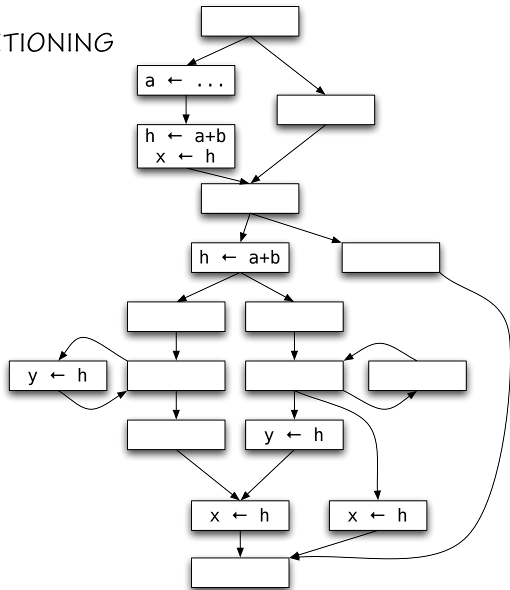
This strategy is “optimal” for eliminating redundancy by code motion.

Could we do even better?

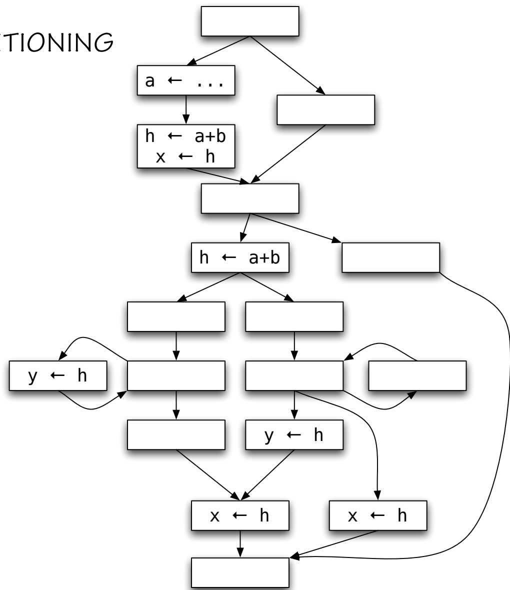
Another Example



Earliest Positioning



Earliest Positioning



Big live ranges. Could we *postpone* some of these computations?

Postponement

Formally, an *expression* $\mathbf{e}$ *could be postponed* to a program point if *every* path to here (from start)

- ✓ goes through an earliest block for $\mathbf{e}$
- ✓ ...without passing through an (original) use of $\mathbf{e}$

Postponement

Formally, an expression $\mathbf{e}$ could be *postponed* to a program point if every path to here (from start)

- ✓ *goes through an earliest block for* $\mathbf{e}$
- ✓ …without passing through an (original) use of $\mathbf{e}$

$$\begin{aligned}
\mathit{PostponeIn}[\mathrm{B}] &= \bigcap_{\mathrm{P} \in \mathit{Pred}(\mathrm{B})} \mathit{PostponeOut}[\mathrm{P}] \\
\mathit{PostponeOut}[\mathrm{B}] &= (\mathit{PostponeIn}[\mathrm{B}] \cup \mathit{EarliestIn}[\mathrm{B}]) \setminus \mathit{ExprsUsed}(\mathrm{B})
\end{aligned}$$

Latest Positioning

An *expression* can be placed at the start of block B if it is in *EarliestIn*[B] or in *PostponeIn*[B].

Latest Positioning

An expression can be placed at the start of block B if
it is in *EarliestIn*[B] or in *PostponeIn*[B].

Ideally, compute expressions as late as possible: the "frontier", where we cannot postpone it any further.

Latest Positioning

An expression can be placed at the start of block B if it is in *EarliestIn*$[B]$ or in *PostponeIn*$[B]$.

Ideally, compute expressions as late as possible: the "frontier", where we cannot postpone it any further.

We say that $e \in$ *LatestIn*$[B]$ if

- ✓ $e \in$ *EarliestIn*$[B] \cup$ *PostponeIn*$[B]$
- ✓ **and**
  - e is used in (the original) B
  - Or, there is a successor S of B such that $e \notin$ (*EarliestIn*$[S] \cup$ *PostponeIn*$[S]$)

Latest Positioning

An expression can be placed at the start of block B if it is in *EarliestIn*$[B]$ or in *PostponeIn*$[B]$.

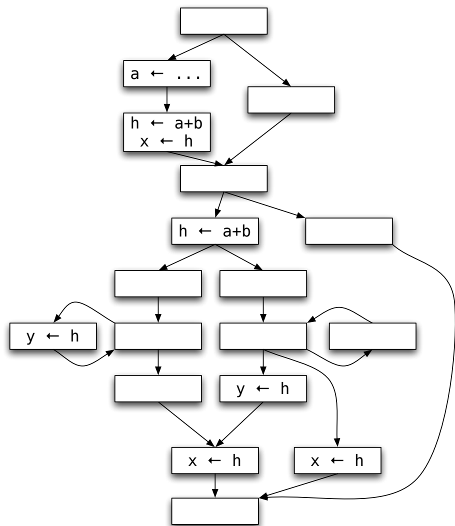
Ideally, compute expressions as late as possible: the "frontier", where we cannot postpone it any further.

We say that $e \in$ *LatestIn*$[B]$ if

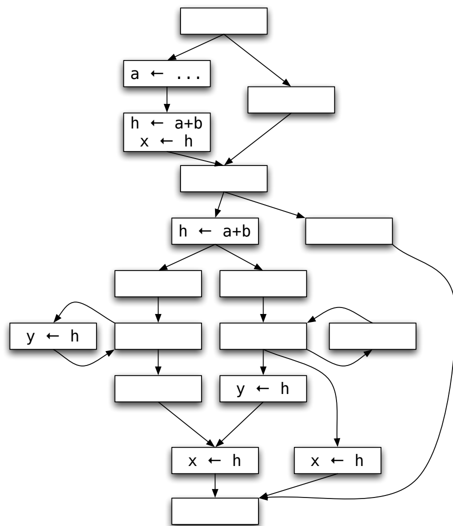
- ✓ $e \in$ *EarliestIn*$[B] \cup$ *PostponeIn*$[B]$
- ✓ **and**
  - e is used in (the original) B
  - Or, there is a successor S of B such that $e \notin$ (*EarliestIn*$[S] \cup$ *PostponeIn*$[S]$)

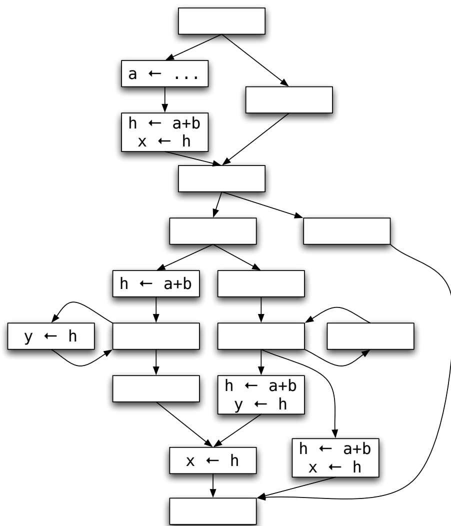
Improved Algorithm: use latest sets instead of earliest sets to place expressions.

Earliest vs. Latest Positioning



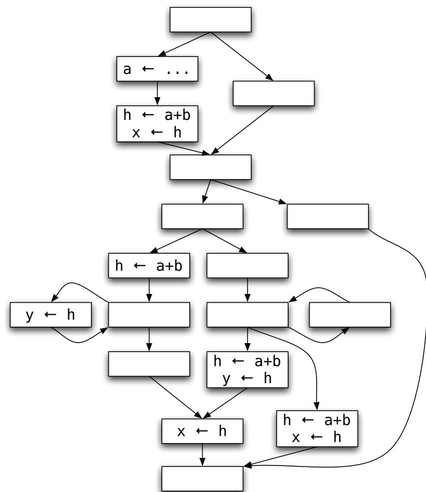
Earliest vs. Latest Positioning





Lazy Code Motion

Adds 4th dataflow to eliminate "pointless" temporaries.



Lazy Code Motion

Adds 4th dataflow to eliminate "pointless" temporaries.