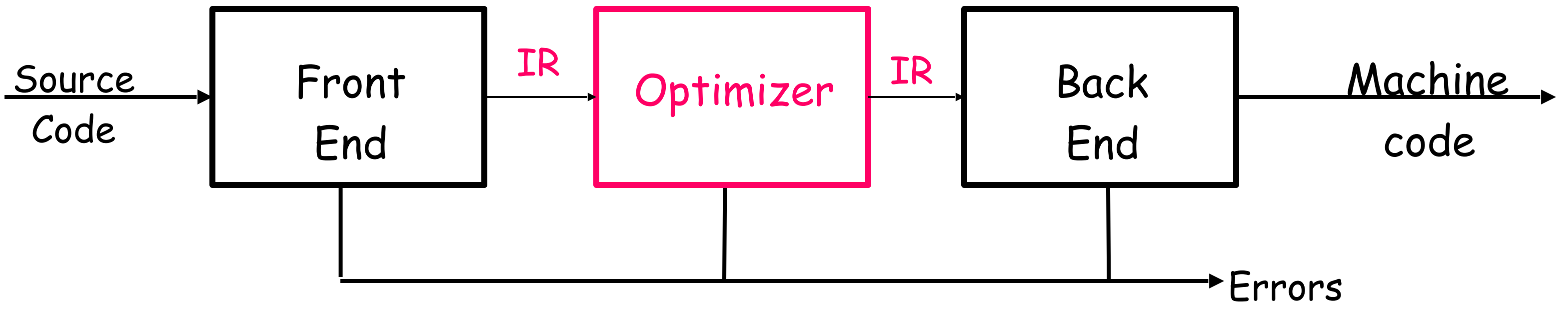
This lecture begins the material from Chapter 8 of EaC

Introduction to Code Optimization

Traditional Three-Phase Compiler



Optimization (or Code Improvement)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …

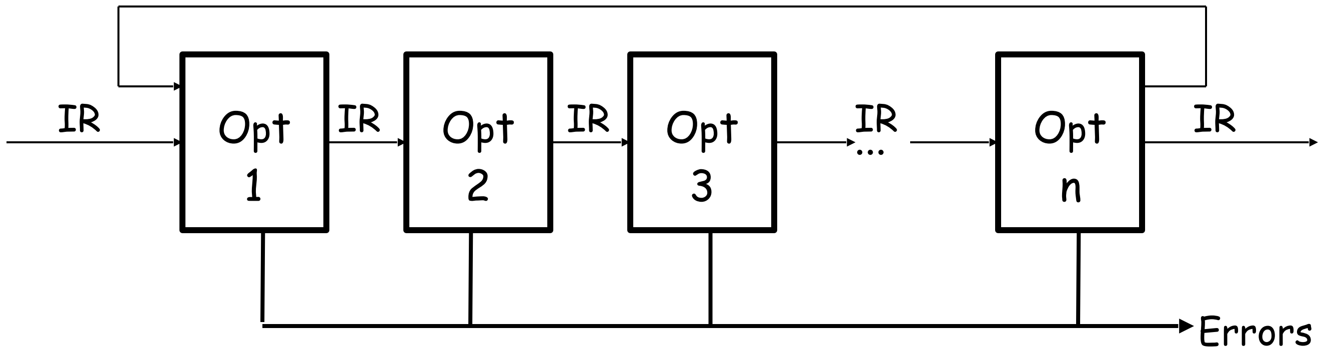
Transformations have to be:

- Safely applied and (it does not change the result of the running program)
- Applied when profit has expected

Background

- Until the early 1980s optimisation was a feature should be added to the compiler only after its other parts were working well
- Debugging compilers vs. optimising compilers
- After the development of RISC processors the demand for support from the compiler had increased

The Optimizer



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code

The Role of the Optimizer

- The compiler can implement a procedure in many ways
- The optimizer tries to find an implementation that is "better"
  - Speed, code size, data space, …

To accomplish this, it

- Analyzes the code to derive knowledge about run-time behavior
  - Data-flow analysis, pointer disambiguation, …
  - General term is "static analysis"
- Uses that knowledge in an attempt to improve the code
  - Literally hundreds of transformations have been proposed
  - Large amount of overlap between them

Nothing "optimal" about optimization

- Proofs of optimality assume restrictive & unrealistic conditions

Scope of Optimization

In scanning and parsing, “scope” refers to a region of the code that corresponds to a distinct name space.

In optimization “scope” refers to a region of the code that is subject to analysis and transformation.

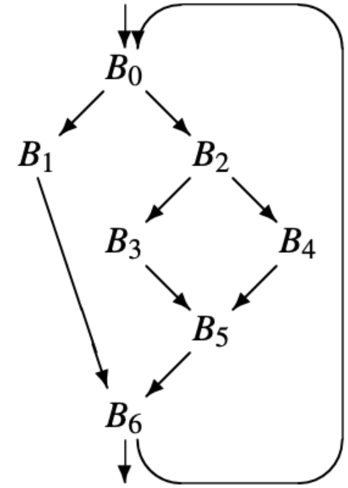
- Notions are somewhat related
- Connection is not necessarily intuitive

Different scopes introduces different challenges & different opportunities

Historically, optimization has been performed at several distinct scopes.

Scope of Optimization

CFG of basic blocks: BB is a maximal length sequence of straightline code.



Local optimization

- Operates entirely within a single basic block
- Properties of block lead to strong optimizations

Regional optimization

- Operate on a region in the CFG that contains multiple blocks
- Loops, trees, paths, extended basic blocks

new opportunities

Whole procedure optimization (intraprocedural)

- Operate on entire CFG for a procedure

Whole program optimization (interprocedural)

- Operate on some or all of the call graph (multiple procedures)
- Must contend with call/return & parameter binding

Assume a low-level, linear IR such as ILOC

Redundancy Elimination as an Example

An expression x+y is redundant if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

- Proving that x+y is redundant, or available
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called value numbering

Rewriting to avoid Redundancy

$$
\begin{aligned}
a &\leftarrow b + c \\
b &\leftarrow a - d \\
c &\leftarrow b + c \\
d &\leftarrow a - d
\end{aligned}
$$

Original Block

$$
\begin{aligned}
a &\leftarrow b + c \\
b &\leftarrow a - d \\
c &\leftarrow b + c \\
d &\leftarrow b
\end{aligned}
$$

Rewritten Block

The resulting code runs more quickly but extend the lifetime of b
This could cause the allocator to spill the value of b

Since the optimiser cannot predict the behaviour of the register allocator, it assumes that rewriting to avoid redundancy is profitable!

Redundancy without textual identity

The problem is more complex that it may seem!

$$a \leftarrow b \times c$$
$$d \leftarrow b$$
$$e \leftarrow d \times c$$

Local Value Numbering

Local algorithm due to Balke (1968) or Ershov (1954)

The key notion

- Assign an identifying number, V(e), to each identifier, constant or expression in general with the following property:
  - V(e1) = V(e2) iff e1 and e2 always have the same value for all possible operand
  - Use hashing over the value numbers to make it efficient
- Use these numbers to improve the code

Improving the code

- Replace redundant expressions
  - Same V(e) $\Rightarrow$ refer rather than recompute

Local Value Numbering

The Algorithm

For each operation o = ‹operator, o_1, o_2› in the block, in order

1. Get value numbers VN(o_1) and VN(o_2) for operands from hash lookup
2. Hash ‹operator,VN(o_1),VN(o_2)› to get a value number for o
3. If o already had a value number, replace o with a reference ‹operator,VN(o_1),VN(o_2)›

If hashing behaves, the algorithm runs in linear time

Local Value Numbering

An example

Original Code	With VNs	Rewritten
$a \leftarrow b + c$	$a^3 \leftarrow b^1 + c^2$	$a \leftarrow b + c$
$b \leftarrow a - d$	$b^5 \leftarrow a^3 - d^4$	$b \leftarrow a - d$
$c \leftarrow b + c$	$c^6 \leftarrow b^5 + c^2$	$c \leftarrow b + c$
* $d \leftarrow a - d$	* $d^5 \leftarrow a^3 - d^4$	* $d \leftarrow b$

One redundancy

- Eliminate stmt with *

Local Value Numbering: the role of naming

An example

Original Code	With VNs	Rewritten
$a \leftarrow x + y$	$a^3 \leftarrow x^1 + y^2$	$a^3 \leftarrow x^1 + y^2$
$b \leftarrow x + y$	$b^3 \leftarrow x^1 + y^2$	* $b^3 \leftarrow a^3$
$a \leftarrow 17$	$a^4 \leftarrow 17$	$a^4 \leftarrow 17$
$c \leftarrow x + y$	$c^3 \leftarrow x^1 + y^2$	* $c^3 \leftarrow a^3$ (oops!)

Two redundancies

- Eliminate stmts with a *

Options

- Use $c^3 \leftarrow b^3$

with a mapping from values to names

- Save a^3 in t^3
- Rename around it

Local Value Numbering: renaming

Example (continued):

Remember the SSA form?

Original Code	With VNs	Rewritten
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0^3 \leftarrow a_0^3$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1^4 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0^3 \leftarrow a_0^3$

Renaming:
- Give each value a unique name
- Makes it clear

Notation:
- While complex, the meaning is clear

Result:
- a_0^3 is available
- Rewriting now works

How to reconcile this new subscripted names with the original ones? A clever implementation would map $a_1 \rightarrow a$ $b_0 \rightarrow b$ $c_0 \rightarrow c$ $a_0 \rightarrow t$

The impact of indirect assignments on SSA form

- To manage the subscripted naming the compiler maintain a map from names to the current subscript.
- With a direct assignment a <- b + c, the changes are clear
- With an indirect assignment *p <- 0?
- The compiler can perform static analysis to disambiguate pointer references (to restrict the set of variables to whom p can refer to).

Ambiguous reference

the compiler cannot isolate a single memory location

Simple Extensions to Value Numbering

Commutative operations

- commutative operations that differs only for the order of their operands should receive the same value numbers a x b and b x a

  Impose an order !!

Constant folding

- Add a bit that records when a value is constant
- Evaluate constant values at compile-time
- Replace an operation with load of the immediate value

Algebraic identities

- Must check (many) special cases

  (organize them into operator-specific decision tree)
- Replace result with input VN

Identities (on VNs)

$x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x \div 1$, $x-x$, $x*0$, $x \div x$, $x \vee 0$, $x \wedge x$, ...
max(x,MAXINT), min(x,MININT), max(x,x), min(y,y), and so on ...

Local Value Numbering (Recap)

The LVN Algorithm, with bells & whistles

Block is a sequence of n operations of the form

$T_i \leftarrow L_i \; Op_i \; R_i$

for i ← 0 to n-1

1. get the value numbers V_1 and V_2 for L_i and R_i
2. if L_i and R_i are both constant then *Constant folding*
   evaluate Li Op_i R_i, assign it to T_i and mark T_i as a constant
3. if Li Op_i R_i matches an identity then *Algebraic identities*
   replace it with a copy operation or an assignment
4. if Op_i commutes and $V_1 > V_2$ then *Commutativity*
   swap V_1 and V_2
5. construct a hash key $<V_1, Op_i, V_2>$

- if the hash key is already present in the table then
  replace operation I with a copy into T_i and mark T_i with the VN

  else

  insert a new VN into table for hash key & mark T_i with the VN

Local Value Numbering

The Algorithm

For each operation o = ‹operator, o_1, o_2› in the block, in order

1 Get value numbers for operands from hash lookup

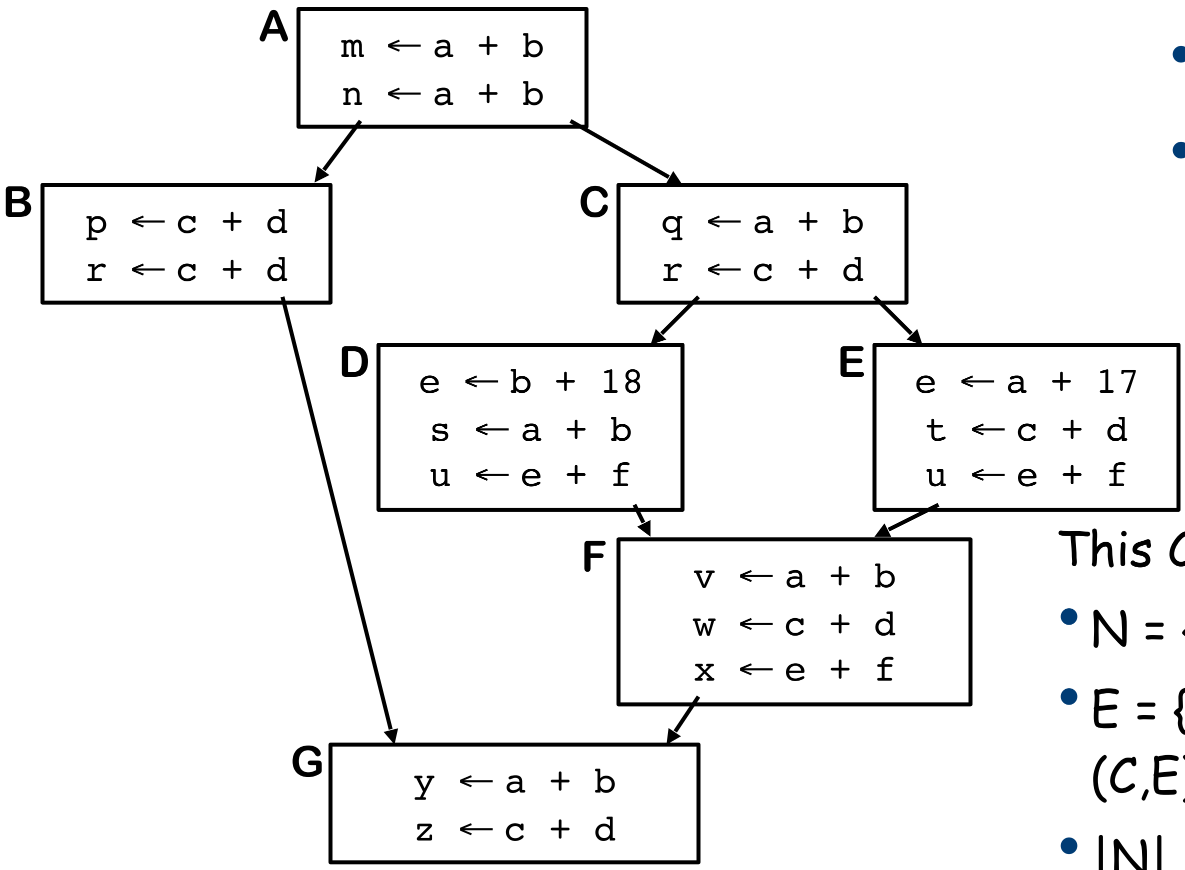
2 Hash ‹operator,VN(o_1),VN(o_2)› to get a value number for o

3 If o already had a value number, replace o with a reference

Complexity & Speed Issues

- "Get value numbers" — linear search versus hash
- "Hash ‹op,VN(o_1),VN(o_2)›" — linear search versus hash
- Copy folding — set value number of result
- Commutative ops — double hash versus sorting the operands

Terminology Control-flow graph (CGF)

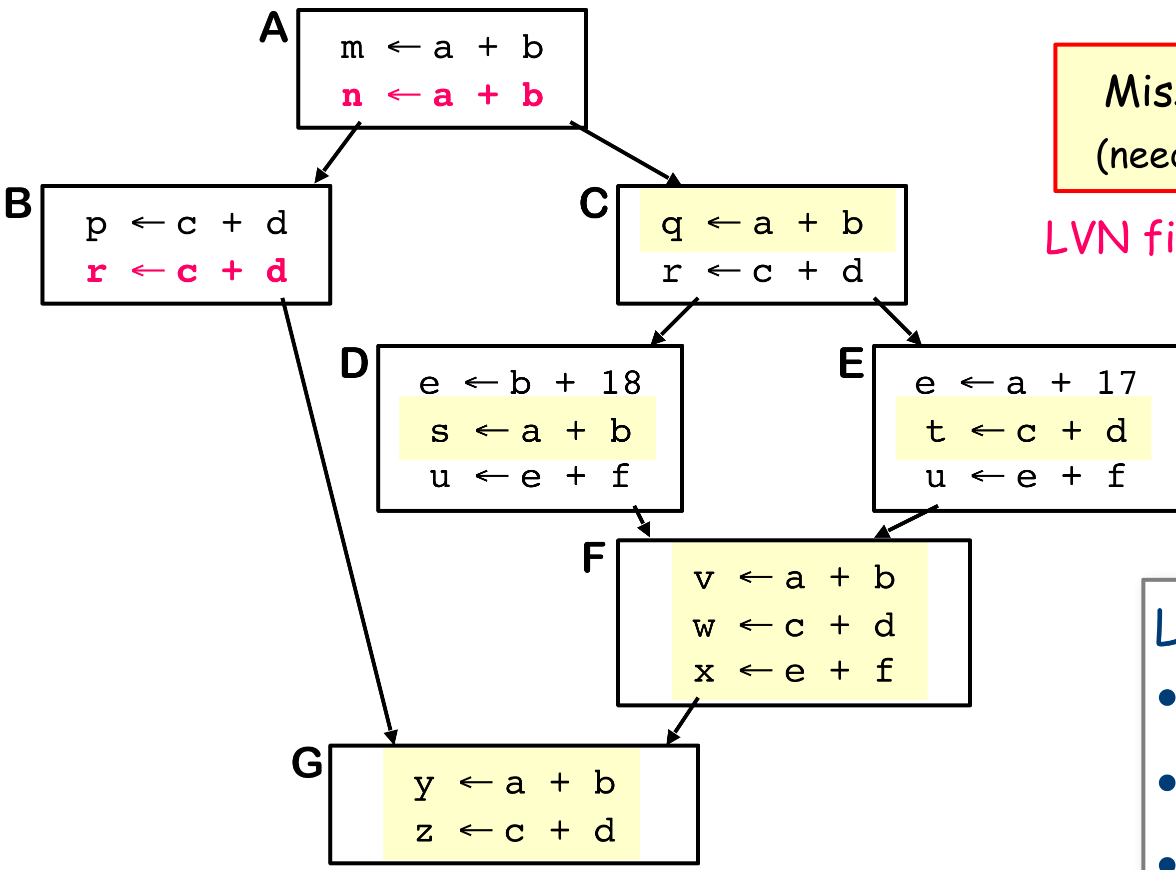


- Nodes for basic blocks
- Edges for branches
- Basis for much of program analysis & transformation

This CFG, G = (N,E)

- N = {A,B,C,D,E,F,G}
- E = {(A,B),(A,C),(B,G),(C,D),(C,E),(D,F),(E,F),(F,E)}
- |N| = 7, |E| = 8

Local Value Numbering



Missed opportunities
(need stronger methods)

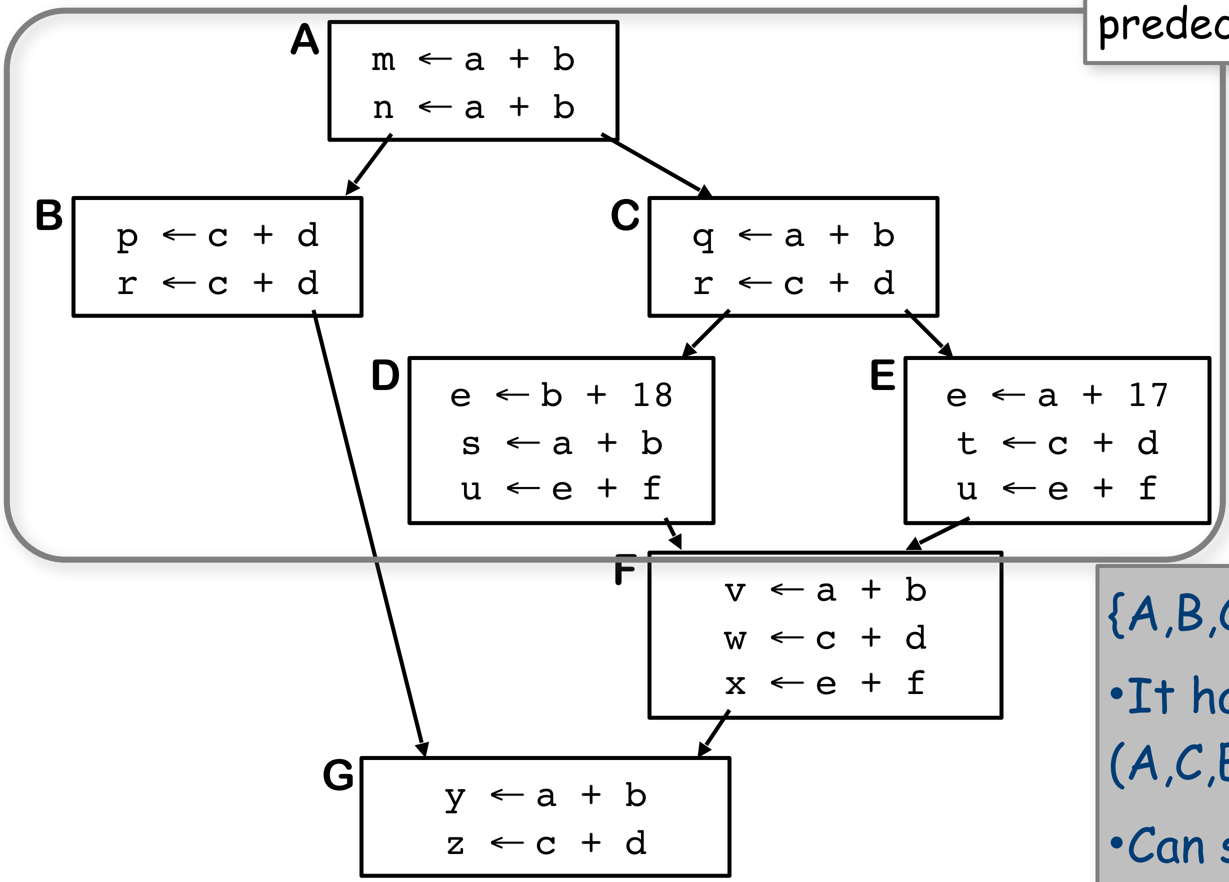
LVN finds these redundant ops

Local Value Numbering

- 1 block at a time
- Strong local results
- No cross-block effects

A Regional Technique

Superlocal Value Numbering



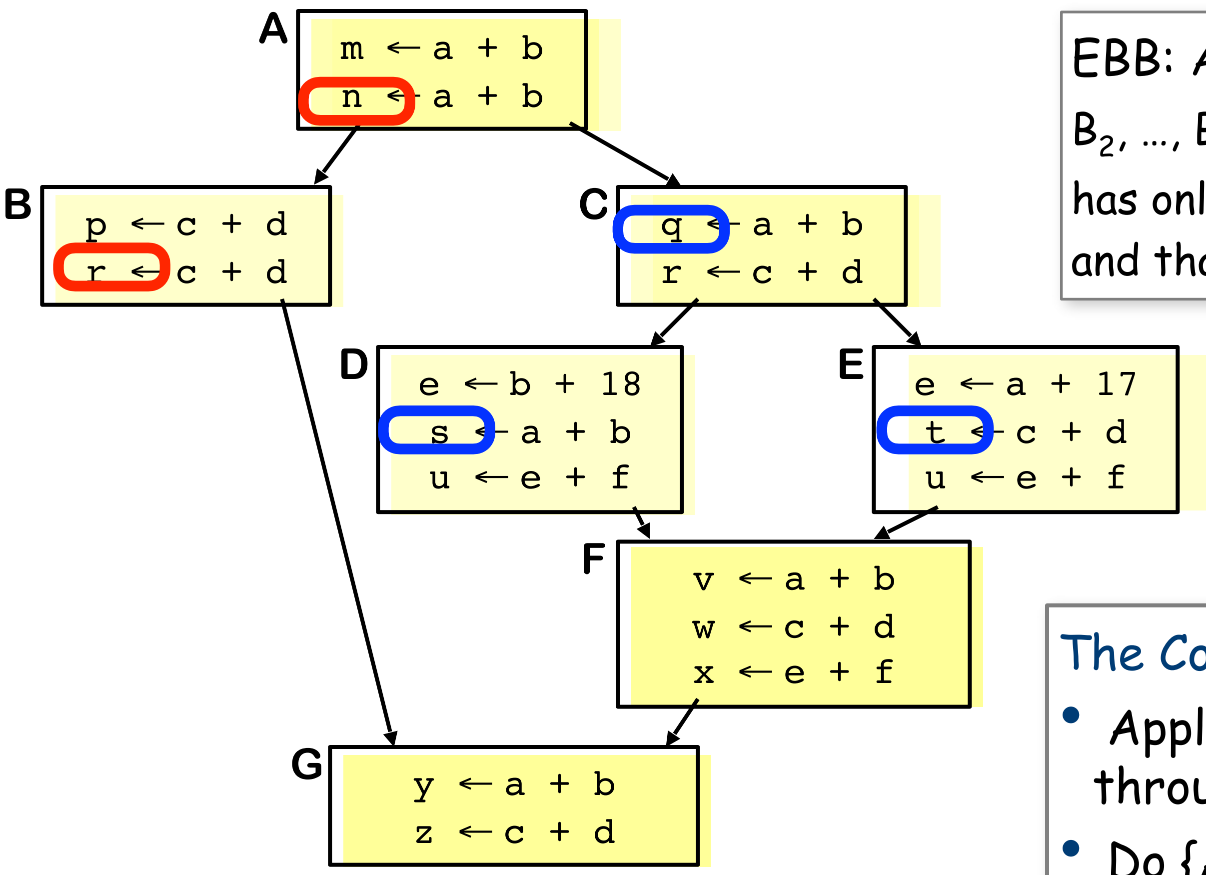
Extended Basic Block: maximal set of blocks $B_1, B_2, \ldots, B_n$ where each B_i, except B_1, has exactly one predecessor in the EBB itself.

{A,B,C,D,E} is an EBB

- It has 3 paths: (A,B), (A,C,D), & (A,C,E)
- Can sometimes treat each path as if it were a block

{F} & {G} are degenerate EBBs

Superlocal Value Numbering



EBB: A maximal set of blocks B_1, B_2, …, B_n where each B_i, except B_1, has only exactly one predecessor and that block is in the EBB.

The Concept

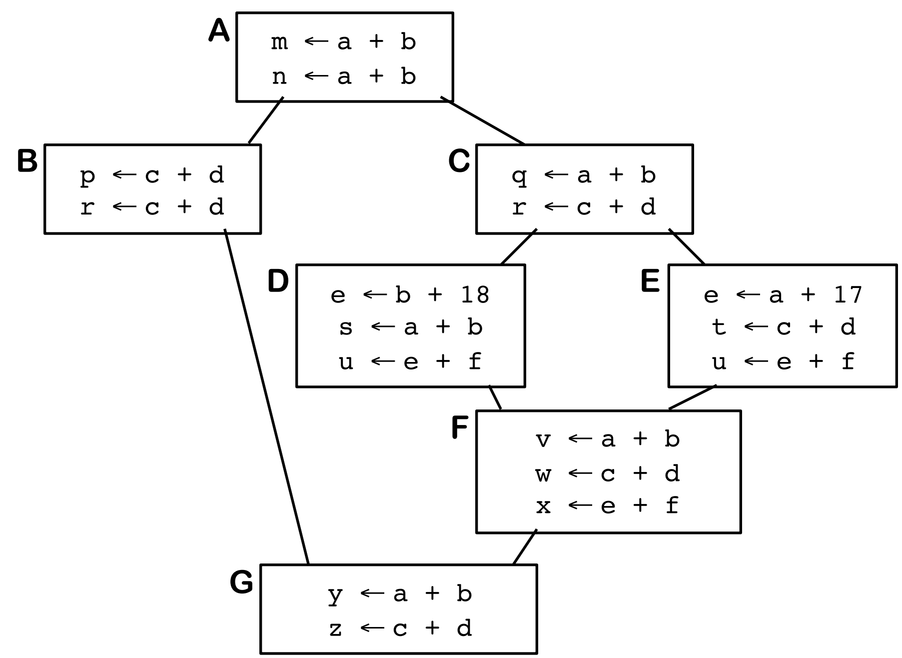
- Apply local method to paths through the EBBs
- Do {A,B}, {A,C,D}, & {A,C,E}
- Obtain reuse from ancestors
- Avoid re-analyzing A & C
- Does not help with F or G

Superlocal Value Numbering

Efficiency

- Use A's table to initialize tables for B & C
- To avoid duplication, use a scoped hash table
  - A, AB, A, AC, ACD, AC, ACE, F, G
- Need a VN→name mapping to handle kills
  - Must restore map with scope
  - Adds complication, not cost

"kill" is a re-definition of some name



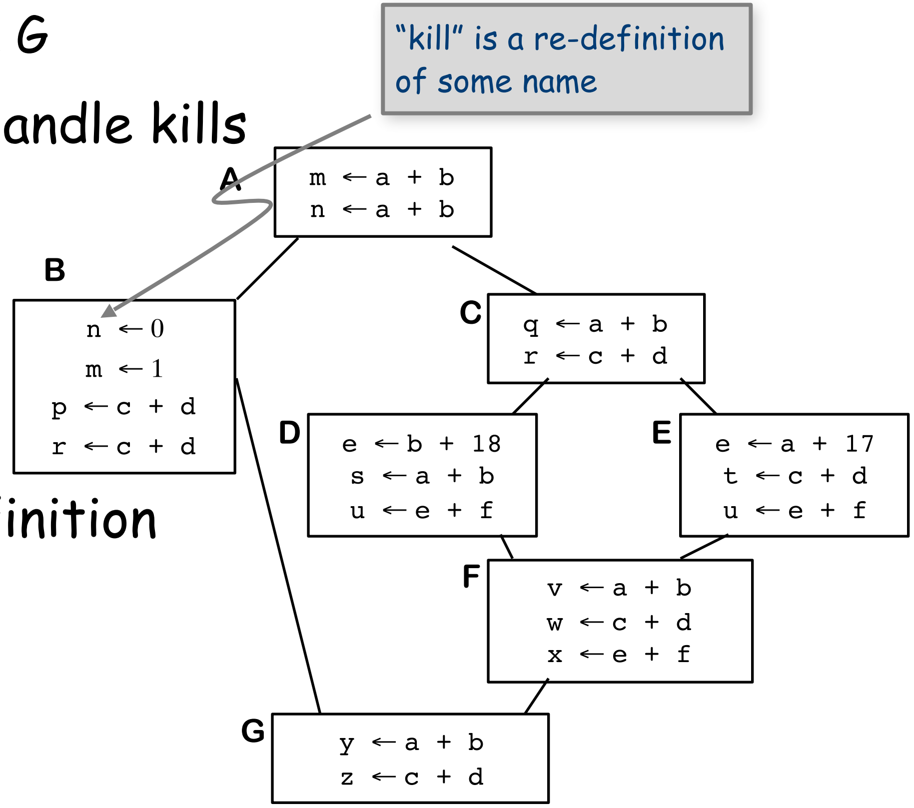
Superlocal Value Numbering

Efficiency

- Use A's table to initialize tables for B & C
- To avoid duplication, use a scoped hash table
  - A, AB, A, AC, ACD, AC, ACE, F, G
- Need a VN→name mapping to handle kills
  - Must restore map with scope
  - Adds complication, not cost

To simplify THE PROBLEM

- Need unique name for each definition
- Use the SSA name space



SSA Name Space (locally)

Example (from earlier):

Original Code	With VNs	Rewritten
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0^3 \leftarrow a_0^3$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1^4 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0^3 \leftarrow a_0^3$

Renaming:
- Give each value a unique name
- Makes it clear

Notation:
- While complex, the meaning is clear

Result:
- a_0^3 is available
- Rewriting just works

SSA Name Space (in general)

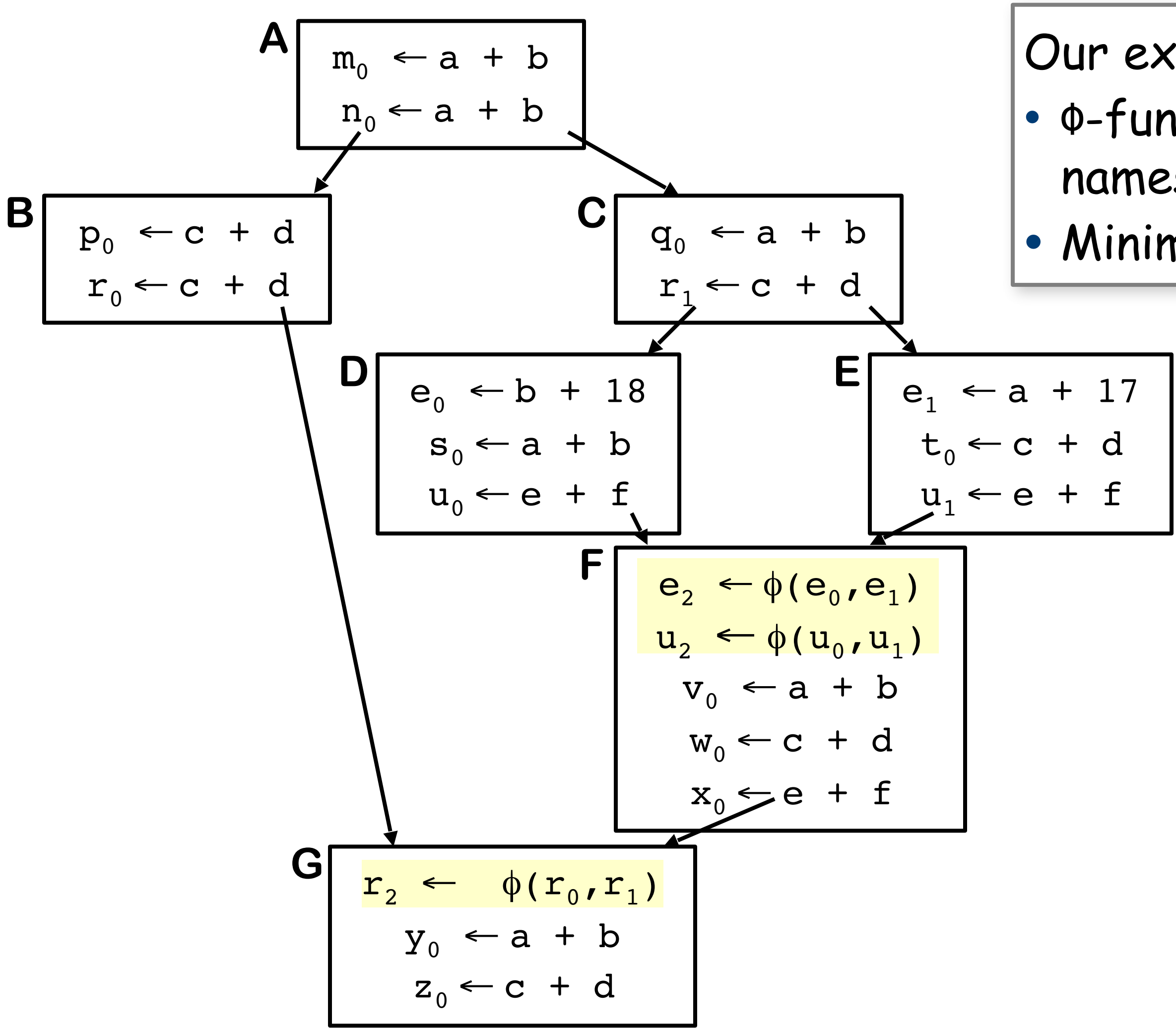
Two principles

- Each name is defined by exactly one operation
- Each operand refers to exactly one definition

To reconcile these principles with real code

- Insert ϕ-functions at merge points to reconcile name space
- Add subscripts to variable names for uniqueness

Superlocal Value Numbering



Our example in SSA form

- Φ-functions at join points for names that need them
- Minimal set of Φ-functions

Superlocal Value Numbering

The SVN Algorithm

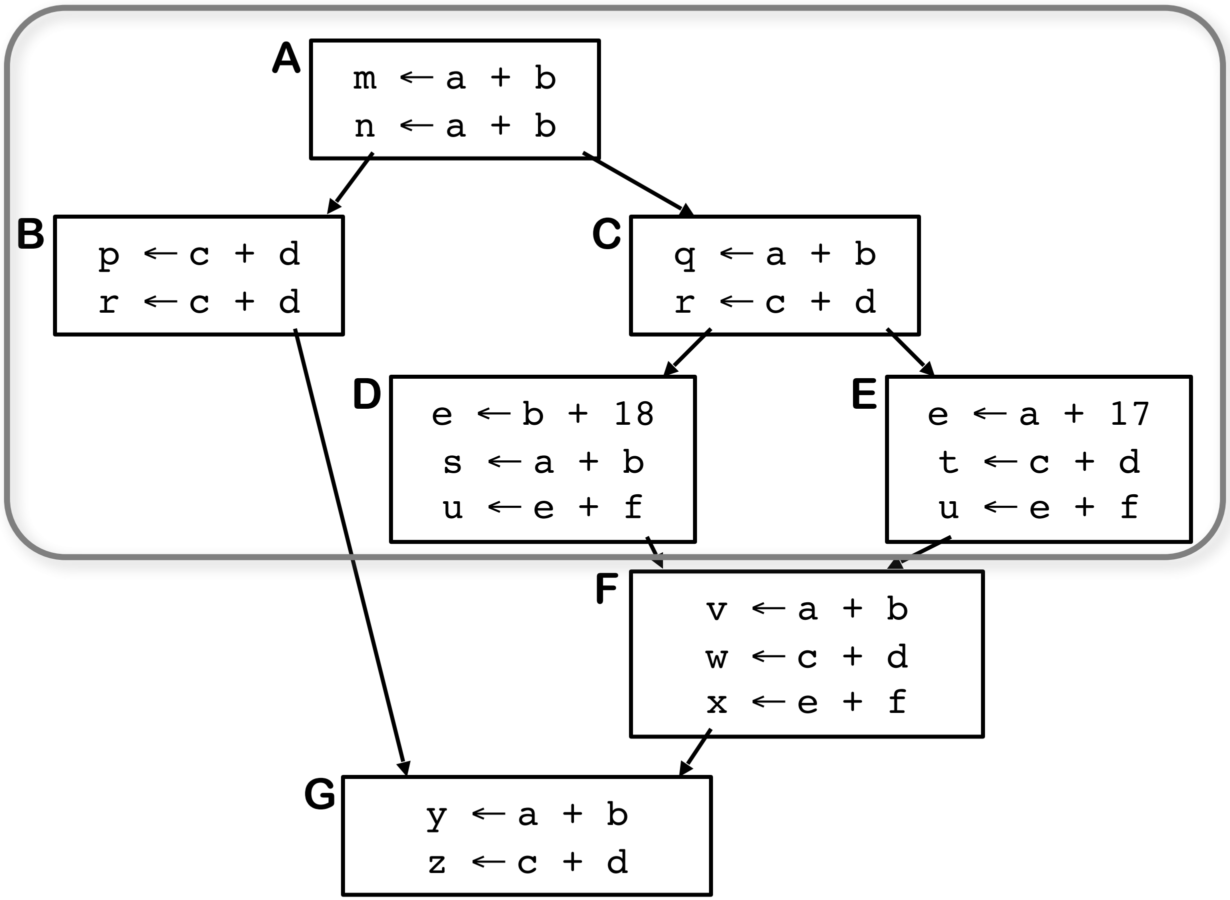
```
WorkList ← { entry block }                                      Blocks to process
Empty ← new table                                               Table for base case
while (WorkList is not empty)
      remove a block b from WorkList
      SVN(b, Empty)

SVN( Block, Table)
      t ← new table for Block, with Table linked as surrounding scope
      LVN( Block, t)                                            Use LVN for the work
      for each successor s of Block
        if s has just 1 predecessor                             In the same EBB
            then SVN( s, t )                                    Starts a new EBB
         else if s has not been processed
            then add s to WorkList
      deallocate t
```

Assumes LVN has been parameterized around block and table

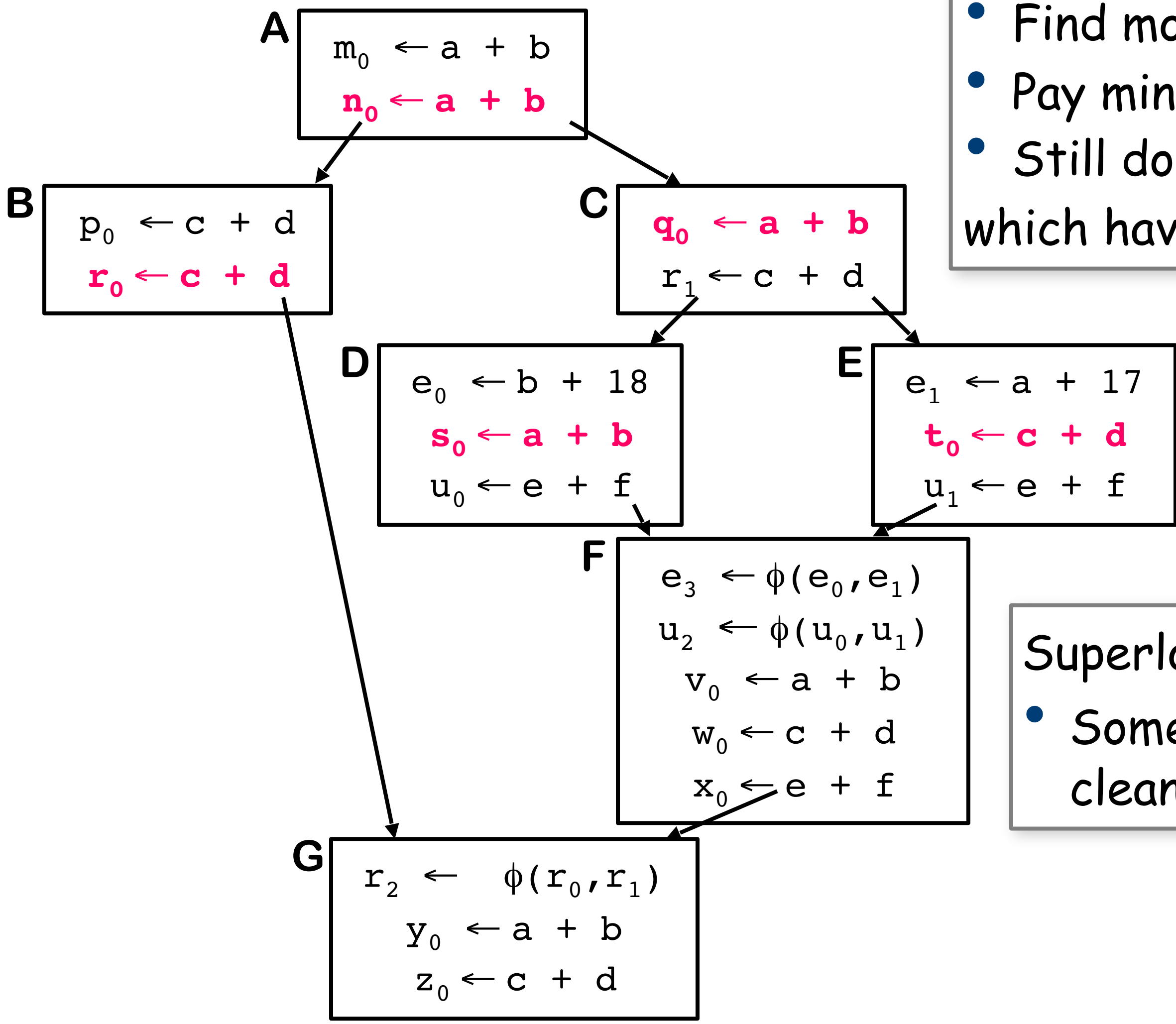
A Regional Technique

Superlocal Value Numbering



1. *Create scope for* B_0
2. *Apply* LVN *to* B_0
3. *Create scope for* B_1
4. *Apply* LVN *to* B_1
5. *Add* B_6 *to* `WorkList`
6. *Delete* B_1*'s scope*
7. *Create scope for* B_2
8. *Apply* LVN *to* B_2
9. *Create scope for* B_3
10. *Apply* LVN *to* B_3
11. *Add* B_5 *to* `WorkList`
12. *Delete* B_3*'s scope*
13. *Create scope for* B_4
14. *Apply* LVN *to* B_4
15. *Delete* B_4*'s scope*
16. *Delete* B_2*'s scope*
17. *Delete* B_0*'s scope*
18. *Create scope for* B_5
19. *Apply* LVN *to* B_5
20. *Delete* B_5*'s scope*
21. *Create scope for* B_6
22. *Apply* LVN *to* B_6
23. *Delete* B_6*'s scope*

Superlocal Value Numbering



With all we saw SVN
- Find more redundancy
- Pay minimal extra cost
- Still does nothing for F & G

which have some opportunities….

Superlocal techniques
- Some local methods extend cleanly to superlocal scopes

A Regional Technique

Loop Unrolling

Applications spend a lot of time in loops

- We can reduce loop overhead by unrolling the loop

```
do i = 1 to 100 by 1
    a(i) ← b(i) * c(i)
    end
```

Complete unrolling

```
a(1)   ← b(1) * c(1)
a(2)   ← b(2) * c(2)
a(3)   ← b(3) * c(3)
  ...
a(100) ← b(100) * c(100)
```

- Eliminated additions, tests and branches: reduce the number of operations Can subject resulting code to strong local optimization!
- Only works with fixed loop bounds & few iterations
- The principle, however, is sound
- Unrolling is always safe, as long as we get the bounds right

Loop Unrolling

Unrolling by smaller factors can achieve much of the benefit

Example: unroll by 4 (8, 16, 32? depends on # of registers)

```
do i = 1 to 100 by 1
        a(i) ← b(i) * c(i)
    end
```

Unroll by 4

```
do i = 1 to 100 by 4
   a(i)   ← b(i) * c(i)
   a(i+1) ← b(i+1) * c(i+1)
   a(i+2) ← b(i+2) * c(i+2)
   a(i+3) ← b(i+3) * c(i+3)
   end
```

Achieves much of the savings with lower code growth

- Reduces tests & branches by 25%
- LVN will eliminate duplicate adds and redundant expressions
- Less overhead per useful operation

But, it relied on knowledge of the loop bounds…

Loop Unrolling

Unrolling with unknown bounds

Need to generate guard loops

```
do i = 1 to n by 1
      a(i) ← b(i) * c(i)
      end
```

Unroll by 4

```
i ← 1
do while (i+3 < n )
     a(i)     ← b(i) * c(i)
     a(i+1) ← b(i+1) * c(i+1)
     a(i+2) ← b(i+2) * c(i+2)
     a(i+3) ← b(i+3) * c(i+3)
     i ←i + 4
     end
do while (i < n)
     a(i)     ← b(i) * c(i)
     i ← i + 1
     end
```

Achieves most of the savings

- Reduces tests & branches by 25%
- LVN still works on loop body
- Guard loop takes some space

Can generalize to arbitrary upper & lower bounds, unroll factors

Loop Unrolling

i=1,…100 : a(i)=a(i)+b(i)+b(i-1)

One other unrolling trick

Eliminate copies at the end of a loop

```
t1 ← b(0)
do i = 1 to 100 by 1
        t2 ← b(i)
        a(i) ← a(i) + t1 + t2
    t1 ← t2
    end
```

Unroll and rename

```
t1 ← b(0)
do i = 1 to 100 by 2
        t2     ← b(i)
       a(i)    ← a(i) + t1 + t2
        t1     ← b(i+1)
     a(i+1) ← a(i+1) + t2 + t1
     end
```

Unroll

- Eliminates the copies, which were a naming artifact
- Achieves some of the benefits of unrolling
  - Lower overhead, longer blocks for local optimization
- Situation occurs in more cases than you might suspect

Sources of Degradation

- It increases the size of the code
- The unrolled loop may have more demand for registers
- If the demand for registers forces additional register spills (store and reloads) then the resulting memory traffic may overwhelm the potential benefits of unrolling