

# Code Optimization

Note by Baris Aktemur:

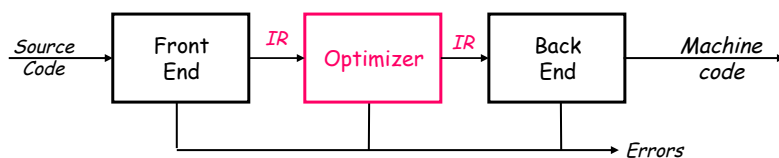
Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

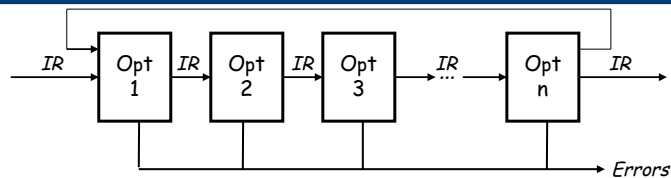
## Traditional Three-Phase Compiler



### 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, ...
- Must preserve “meaning” of the code
  - Measured by values of named variables
  - A course (or two) unto itself

## 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
- Encode an idiom in some particularly efficient form

2

## 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

3

## 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*

Assume a low-level, linear IR such as ILOC 4

## Local Value Numbering

An example

### Original Code

$a \leftarrow x + y$   
\*  $b \leftarrow x + y$   
 $a \leftarrow 17$   
\*  $c \leftarrow x + y$

### With VNs

$a^3 \leftarrow x^1 + y^2$   
\*  $b^3 \leftarrow x^1 + y^2$   
 $a^4 \leftarrow 17$   
\*  $c^3 \leftarrow x^1 + y^2$

### Rewritten

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

### *Two redundancies*

- Eliminate stmts with a \*
- Coalesce results ?

### *Options*

- Use  $c^3 \leftarrow b^3$
- Save  $a^3$  in  $t^3$
- Rename around it

5

## Local Value Numbering

Example (continued):

### Original Code

$a_0 \leftarrow x_0 + y_0$   
\*  $b_0 \leftarrow x_0 + y_0$   
 $a_1 \leftarrow 17$   
\*  $c_0 \leftarrow x_0 + y_0$

### With VNs

$a_0^3 \leftarrow x_0^1 + y_0^2$   
\*  $b_0^3 \leftarrow x_0^1 + y_0^2$   
 $a_1^4 \leftarrow 17$   
\*  $c_0^3 \leftarrow x_0^1 + y_0^2$

### Rewritten

$a_0^3 \leftarrow x_0^1 + y_0^2$   
\*  $b_0^3 \leftarrow a_0^3$   
 $a_1^4 \leftarrow 17$   
\*  $c_0^3 \leftarrow a_0^3$

#### 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

6

## Value Numbering

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

### The key notion

- Assign an identifying number,  $V(n)$ , to each expression
  - $V(x+y) = V(j)$  iff  $x+y$  and  $j$  always have the same value
  - Use hashing over the value numbers to make it efficient
- Use these numbers to improve the code

### Improving the code

- Replace redundant expressions
  - Same VN  $\Rightarrow$  refer rather than recompute
- Simplify algebraic identities
- Discover constant-valued expressions, fold & propagate them
- Technique designed for low-level, linear IRs, similar methods exist for trees (e.g., build a DAG)

Within a basic block; definition becomes more complex across blocks

7

## Local Value Numbering

### The Algorithm

For each operation  $o = \langle \text{operator}, o_1, o_2 \rangle$  in the block, in order

- 1 Get value numbers for operands from hash lookup
- 2 Hash  $\langle \text{operator}, \text{VN}(o_1), \text{VN}(o_2) \rangle$  to get a value number for  $o$
- 3 If  $o$  already had a value number, replace  $o$  with a reference
- 4 If  $o_1$  &  $o_2$  are constant, evaluate it & replace with a loadI

If hashing behaves, the algorithm runs in linear time

- If not, use multi-set discrimination<sup>†</sup> or acyclic DFAs<sup>††</sup>

Handling algebraic identities

- Case statement on operator type
- Handle special cases within each operator

<sup>†</sup>see p. 251 in *EaC*

<sup>††</sup>DFAs for REs without closure can be built online to provide a “perfect hash”

## Local Value Numbering

### The Algorithm

For each operation  $o = \langle \text{operator}, o_1, o_2 \rangle$  in the block, in order

- 1 Get value numbers for operands from hash lookup
- 2 Hash  $\langle \text{operator}, \text{VN}(o_1), \text{VN}(o_2) \rangle$  to get a value number for  $o$
- 3 If  $o$  already had a value number, replace  $o$  with a reference
- 4 If  $o_1$  &  $o_2$  are constant, evaluate it & replace with a loadI

asymptotic

constants

Complexity & Speed Issues

- “Get value numbers” — linear search versus hash
- “Hash  $\langle \text{op}, \text{VN}(o_1), \text{VN}(o_2) \rangle$ ” — linear search versus hash
- Copy folding — set value number of result
- Commutative ops — double hash versus sorting the operands

## Simple Extensions to Value Numbering

### Constant folding

- Add a bit that records when a value is constant
- Evaluate constant values at compile-time
- Replace with load immediate or immediate operand
- No stronger local algorithm

### Identities (on VNs)

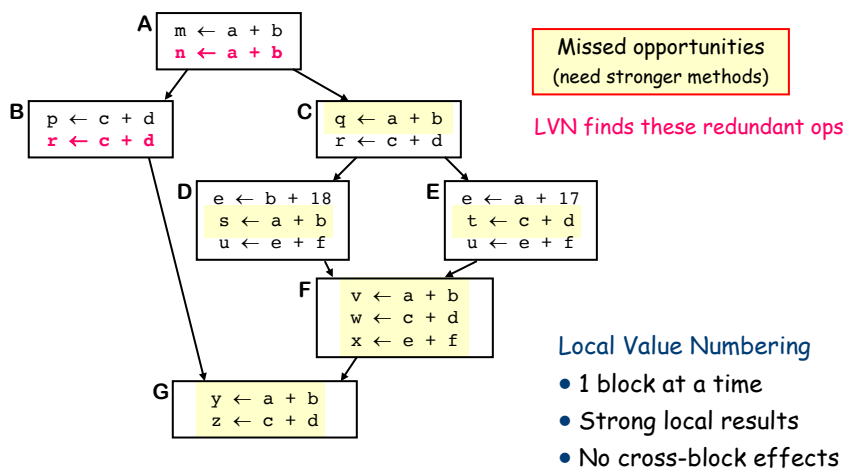
$x \leftarrow -y$ ,  $x \leftarrow 0$ ,  $x \leftarrow -0$ ,  $x \leftarrow x * 1$ ,  $x \leftarrow 1 * x$ ,  $x \leftarrow x - x$ ,  $x \leftarrow x * 0$ ,  
 $x \leftarrow x + x$ ,  $x \leftarrow 0$ ,  $x \leftarrow x \wedge 0x\text{FF}\dots\text{FF}$ ,  
 $\max(x, \text{MAXINT})$ ,  $\min(x, \text{MININT})$ ,  
 $\max(x, x)$ ,  $\min(y, y)$ , and so on ...

### Algebraic identities

- Must check (many) special cases
- Replace result with input VN
- Build a decision tree on operation

10

## Value Numbering



11

## Scope of Optimization

### Local optimization

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

A basic block is a maximal length sequence of straightline code.

### Regional optimization

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

### Whole procedure optimization *(intraprocedural)*

- Operate on entire CFG for a procedure
- Presence of cyclic paths forces analysis then transformation

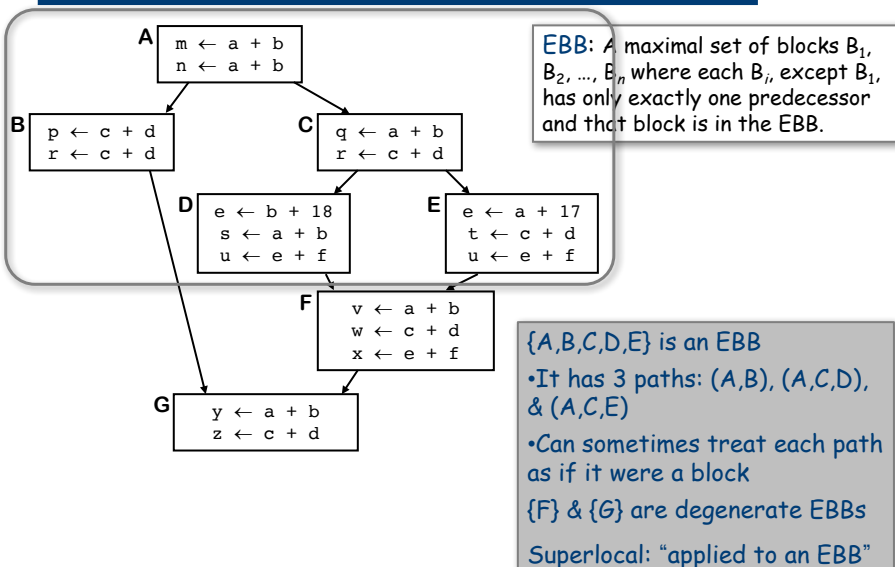
### Whole program optimization *(interprocedural)*

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

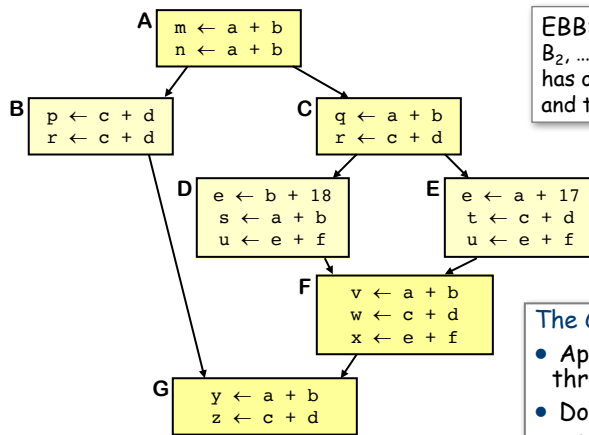
12

### A Regional Technique

## Superlocal Value Numbering



## Superlocal Value Numbering



EBB: A maximal set of blocks  $B_1, B_2, \dots, B_n$  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

\* 14

## SSA Name Space

(locally)

Example (from earlier):

### Original Code

$a_0 \leftarrow x_0 + y_0$   
 \*  $b_0 \leftarrow x_0 + y_0$   
 $a_1 \leftarrow 17$   
 \*  $c_0 \leftarrow x_0 + y_0$

### With VNs

$a_0^3 \leftarrow x_0^1 + y_0^2$   
 \*  $b_0^3 \leftarrow x_0^1 + y_0^2$   
 $a_1^4 \leftarrow 17$   
 \*  $c_0^3 \leftarrow x_0^1 + y_0^2$

### Rewritten

$a_0^3 \leftarrow x_0^1 + y_0^2$   
 \*  $b_0^3 \leftarrow a_0^3$   
 $a_1^4 \leftarrow 17$   
 \*  $c_0^3 \leftarrow a_0^3$

### 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

15



## SSA Name Space

(in general)

Two principles

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

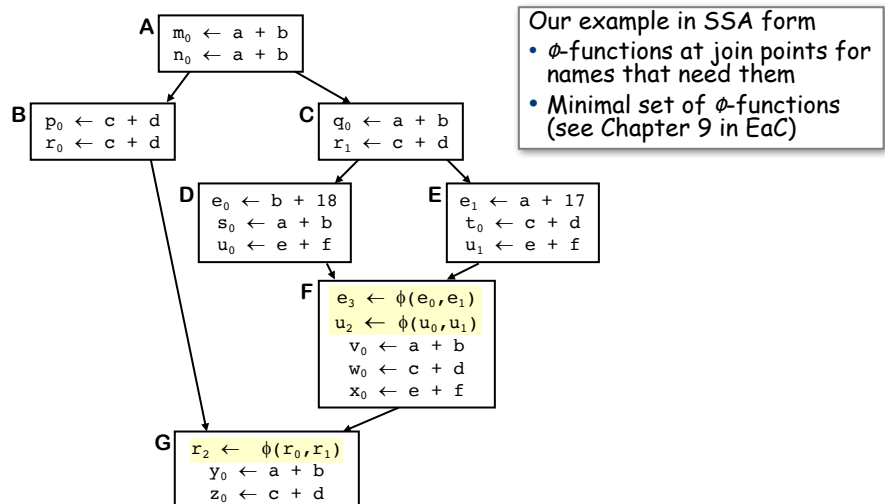
To reconcile these principles with real code

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



16

## Superlocal Value Numbering



17

## Superlocal Value Numbering

### The SVN Algorithm

```

WorkList ← { entry block }
Empty ← new table
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)
    for each successor s of Block
        if s has just 1 predecessor
            then SVN( s, t )
        else if s has not been processed
            then add s to WorkList
    deallocate t
    
```

Blocks to process

Table for base case

Use LVN for the work

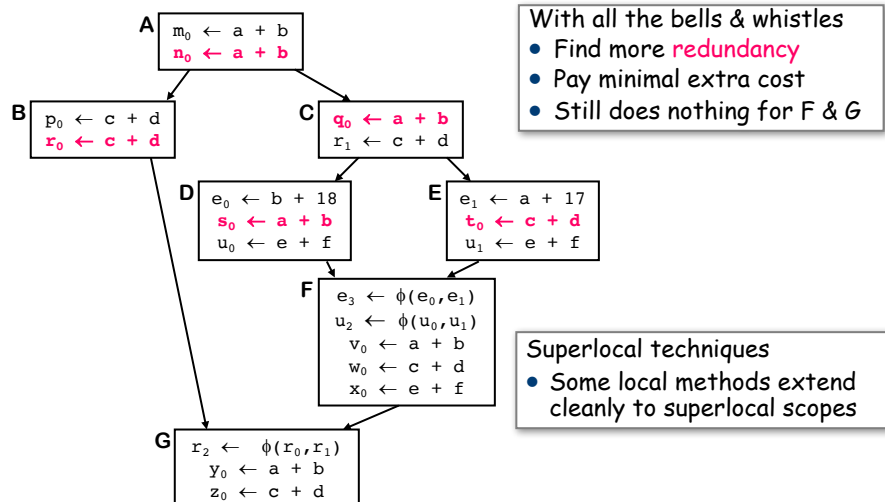
In the same EBB

Starts a new EBB

Assumes LVN has been parameterized around block and table

18

## Superlocal Value Numbering



19


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

- Eliminated additions, tests, and branches
  - 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


20

### Loop Unrolling

Unrolling by smaller factors can achieve much of the benefit

Example: unroll by 4

```
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...

21

## 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
```



```
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

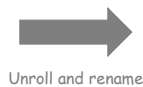
22

## Loop Unrolling

### 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
```



```
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
```

This result has been rediscovered many times. [Kennedy's thesis] 23

## The Story So Far ...

---

### Introduced scope of optimization

- Local — a single basic block
- Regional — a subset of the blocks in a procedure
- Global — an entire procedure Intraprocedural
- Whole Program — multiple procedures Interprocedural

### Some example optimizations

- Local Value Numbering
- Superlocal Value Numbering
- Loop Unrolling
- Data-analysis & an application
- Procedure Placement

24

### A Global Technique

## Finding Uninitialized Variables

---

We can use global data-flow analysis to find variables that might be used before they are ever defined

*A variable  $v$  is live at point  $p$  iff  $\exists$  a path in the CFG from  $p$  to a use of  $v$  along which  $v$  is not redefined.*

Any variable that is live in the entry block of a procedure may be used before it is defined

- Represents a potential logic error in the code
- Compiler should discover and report this condition

We are looking at this issue because it gives us an opportunity to introduce the computation of *liveness* - a data-flow analysis.

- **Data-flow analysis** is a form of compile-time reasoning about the runtime flow of values.

25