# Code Optimization

15CSE311 Compiler Design

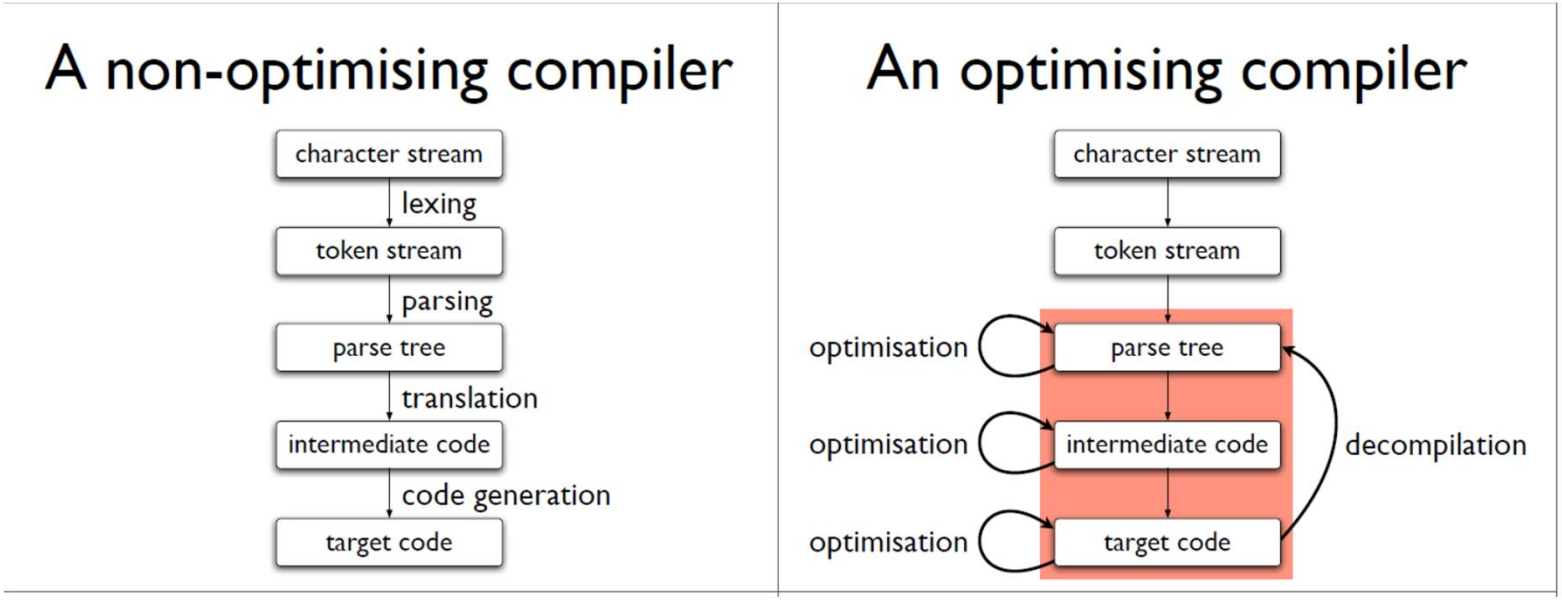
Dr.S.Padmavathi

CSE

# CODE OPTIMIZATION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations.
- Compilers that apply code-improving transformations are called **optimizing compilers**
- Optimizations are classified into two categories. They are
  - Machine independent optimizations:
  - Machine dependant optimizations:
- Machine independent optimizations:
  - Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine. Eg. Dead code elimination, common subexpression elimination, constant folding
- Machine dependant optimizations:
  - Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences. Eg: register allocation, instruction selection

# Debugging vs Optimizing compiler




**Debugging compiler**

- quick compilation at the expense of code quality.
- did not significantly rearrange the code,
- strong correspondence between the source code and the executable code.
- mapping a runtime error to a specific line of source code

**Optimizing compiler**

- improving the running time of the executable code at the expense of compile time.
- optimizer often moves operations around,
- the mapping from source code to executable code is less transparent, and debugging is harder

# Examples of transformation

```
while (i <= k*2) {
j = j * i;
i = i + 1;
}
```

**correct**

```
int t = k * 2;
while (i <= t) {
j = j * i;
i = i + 1;
}
```

```
while (i <= k*2) {
k = k - i;
i = i + 1;
}
```

**Incorrect**

```
int t = k * 2;
while (i <= t) {
k = k - i;
i = i + 1;
}
```

# Examples of Transformation

```
int main(void)
{
return 42;
}
int f(int x)
{
return x * 2;
}
```

**correct**

```
int main(void)
{
return 42;
}
```

```
int main(void)
{
return f(21);
}
int f(int x)
{
return x * 2;
}
```

**incorrect**

```
int main(void)
{
return f(21);
}
```

# Typical Transformations

- Discover & propagate a constant value – constant folding, copy propagation
- Move evaluation to a less-frequently executed place in the code – code motion
- Specialize code based on context- strength reduction
- Find & remove redundant code –common subexpression elimination
- Remove useless or unreachable code – dead code elimination
- Take advantage of a processor feature- instruction selection

# Dead vs. unreachable code

```
int f(int x, int y) {
int z = x * y; DEAD
return x + y;
}
```

Dead code computes unused values.

(Waste of time.)

Deadness is a *data-flow* property

```
int f(int x, int y) {
return x + y;
int z = x * y; UNREACHABLE
}
```

Unreachable code cannot possibly be executed.

(Waste of space.)

Unreachability is a *control-flow* property

# Transformation: Reduction In Strength

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, $x^2$ is invariably cheaper to implement as x*x than as a call to an exponentiation routine.
- Fixed-point multiplication or division by a power of two is cheaper to implement as a shift.
- Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

# Array reference- Strength reduction

A

| 1,1 | 1,2 | 1,3 | 1,4 |
|---|---|---|---|
| 2,1 | 2,2 | 2,3 | 2,4 |

Row major

| 1,1 | 1,2 | 1,3 | 1,4 | 2,1 | 2,2 | 2,3 | 2,4 |
|---|---|---|---|---|---|---|---|

2000 2004 2008 2012 2016 2020 2024 2028

Column major

| 1,1 | 2,1 | 1,2 | 2,2 | 1,3 | 2,3 | 1,4 | 2,4 |
|---|---|---|---|---|---|---|---|

2000 2004 2008 2012 2016 2020 2024 2028

Computation of address of A[i,j]

$$@A + (i - low_1) \times len_2 \times w + (j - low_2) \times w$$

$$@m + (j - low_2(m)) \times (high_1(m) - low_1(m) + 1) \times w + (i - low_1(m)) \times w$$

$$@m + (j - 1) \times hw + (i - 1) \times w$$

**Strength reduction**

a transformation that rewrites a series of operations, for example

$$i \cdot c, (i+1) \cdot c, \ldots, (i+k) \cdot c$$

with an equivalent series

$$i'_1, i'_2, \ldots, i'_k,$$

where $i'_1 = i \cdot c$ and $i'_j = i'_{j-1} + c$

where $hw$ is $high_1(m) \times w$. If the reference occurs inside a loop where j runs from 1 to $k$, the compiler might use *operator strength reduction* to replace the term $(j - 1) \times hw$ with a sequence $j'_1, j'_2, j'_3, \ldots j'_k$, where $j'_1 = (1 - 1) \times hw = 0$ and $j'_i = j'_{i-1} + hw$. If i is also the induction variable of a loop running from 1 to $l$, then strength reduction can replace $(i - 1) \times w$ with the sequence $i'_1, i'_2, i'_3, \ldots i'_l$, where $i'_1 = 0$ and $i'_j = i'_{j-1} + w$. After these changes, the address calculation is just

$$@m + j' + i'$$

# Transformation: Loop unrolling

This replicates the loop body for distinct iterations and adjusts the index calculations to match.

```
      do 60 j = 1, n2
         do 50 i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
50       continue
60    continue
```

16 assignments merged into a single statement
eliminating 15 occurrences of y(i) = y(i) + ; eliminates 15 additions and most of the loads and stores of y(i)

**Unrolling** the loop eliminates some scalar operations.

```
        ...
      jmin = j+16
      do 60 j = jmin, n2, 16
          do 50 i = 1, n1
           y(i) = ((((((((((((((( (y(i))
     $          + x(j-15)*m(i,j-15)) + x(j-14)*m(i,j-14))
     $          + x(j-13)*m(i,j-13)) + x(j-12)*m(i,j-12))
     $          + x(j-11)*m(i,j-11)) + x(j-10)*m(i,j-10))
     $          + x(j- 9)*m(i,j- 9)) + x(j- 8)*m(i,j- 8))
     $          + x(j- 7)*m(i,j- 7)) + x(j- 6)*m(i,j- 6))
     $          + x(j- 5)*m(i,j- 5)) + x(j- 4)*m(i,j- 4))
     $          + x(j- 3)*m(i,j- 3)) + x(j- 2)*m(i,j- 2))
     $          + x(j- 1)*m(i,j- 1)) + x(j) *m(i,j)
50        continue
60    continue
        ...
      end
```

# Transformation continued

## Constant Folding

- at compile time if the value of an expression is a constant, using constant instead of expression is known as **constant folding**
- For example,
- a=3.14157/2 can be replaced by
- a=1.570 there by eliminating a division operation

## Copy Propagation

- Assignments of the form f : = g called copy statements.
- Copy propagation means use of one variable instead of another, there by possibly eliminating it.
- For example: copy propagation :
- x=Pi;
- …… ……
- A=x*r*r; A=Pi*r*r;

variable x is eliminated

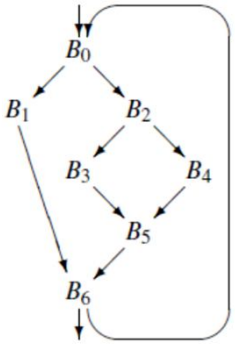
# Terminology

- Optimization- refer to a broad technique or strategy, such as code motion or dead code elimination
- Transformation -refer to algorithms & techniques that rewrite the code being compiled eg. Dead code elimination, common sub expression elimination
- Analysis- refer to algorithms & techniques that derive information about the code being compiled eg. Data flow analysis, live variable analysis, available expression analysis
- Optimization = Analysis + Transformation
- Eg available expression analysis+ common sub expression elimination

# 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.**
- **Different scopes**
  - **Local optimization**
    - A transformation of a program is called **local** if it can be performed by looking only at the statements in a basic block
  - **Regional optimization**
  - **Whole procedure optimization (*intraprocedural* or *global*)**
  - **Whole program optimization (*interprocedural* or *universal*)**

# Scope of Optimization




**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, Dominator

**Whole procedure optimization (*intraprocedural* or *global*)**

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

e.g. live variable analysis, available expressions

**Whole program optimization (*interprocedural* or *universal*)**

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

e.g. unreachable-procedure elimination

A **basic block** is a maximal length
sequence of straight-line code.

**Extended basic block**
a set of blocks β1, β2, . . ., β*n* where β1 has multiple CFG predecessors and each other *i* has just one, which is some *j* in the set
Eg. {*B*0, *B*1, *B*2, *B*3, *B*4},{*B*5}, and {*B*6}.

**Dominator**
In a CFG, *x dominates y* if and only if every path from the root to *y* includes *x*.

A **control-flow graph** (**CFG**) has a node for each basic block and an edge for each branch or jump.

A **call graph** has a node for each
procedure and an edge for each call

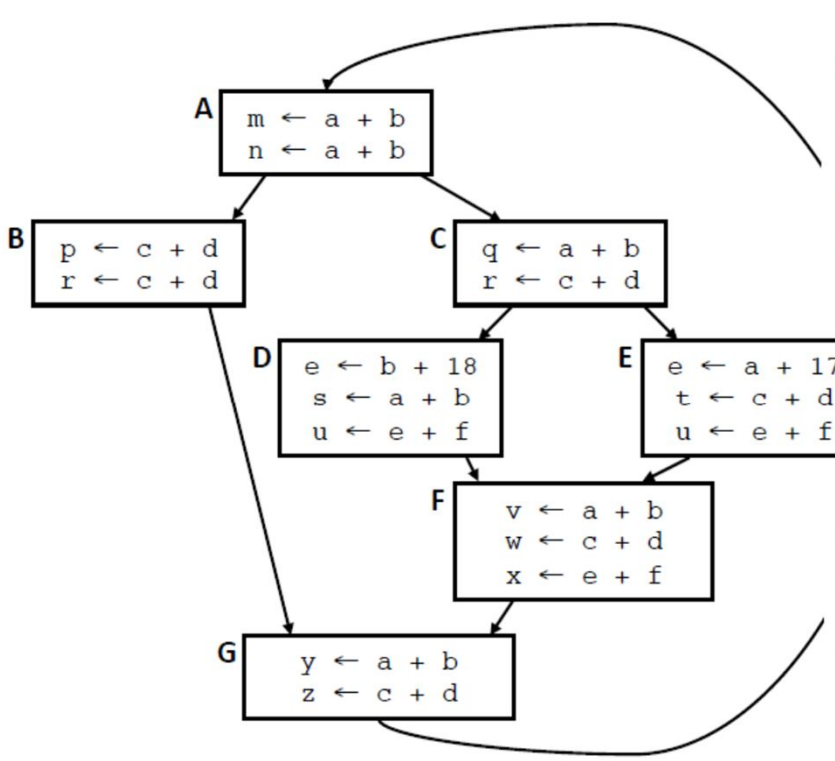
# Redundancy Elimination or Common Subexpression Elimination

- 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 local technique for accomplishing both is called *value numbering*

a ← **b + c**
b ← a - d
c ← b + c not redundant
d ← a – d redundant
Original Block

a ← b + c
b ← a - d
c ← b + c
d ← b
Rewritten Block

# CFG example




**Control-flow graph (CFG)**

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

**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(LVN)

- **Performing redundancy elimination in the local context works well**
- • Within a block, compiler understands the execution order
- – Blocks that execute before or after the current block are uncertain territory
- – Any value created inside the block is certain
- • Much of the available improvement can be caught locally
- – Redundancy in address expressions, constant folding
- – Algebraic simplification of expressions
- **What, then, is the role of larger scopes?**
- • Some opportunities need more context than a block
- – Code motion & placement are clearly non-local
- – Regional optimizations, such as improving a loop nest
- – Removing useless or unreachable code requires a larger scope
- • Discovering and using knowledge about the “uncertain territory”
- – Serious opportunities exist, but compiler should get local ones first

# Value Numbering

- **The Key Notion**
- • Assign an identifying number, VN(n), to each expression
- – VN(x+y) = VN(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 Þrefer to prior value 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**, a *directed acyclic graph*)

**Identities (on VNs)**
X←y, x+0, x-0, x*1, x÷1, x-x, x*0, x÷x, xV0, x ^ 0xFF...FF, max(x,MAXINT), min(x,MININT), max(x,x), min(y,y), and so on ...

# LVN algorithm

**The Algorithm**

For each operation ***o* = <operator, o1, o2>** in the block, in order

1. Get value numbers for operands from hash lookup

2. Hash **<operator, VN(o1), VN(o2)>** to get a value number for *o*

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

4. If o1 & o2 are constant, evaluate it & replace with a **loadI**

for i  0 to n - 1, where the block has n operations
"T*i*  L*i* Op*i* R*i*"

1. get the value numbers for L*i* and R*i*

2. construct a hash key from Op*i* and the value numbers for L*i* and R*i*

3. if the hash key is already present in the table then

replace operation i with a copy of the value into T*i* and

associate the value number with T*i*

else

insert a new value number into the table at the hash key location

record that new value number for T*i*

# Renaming available expressions

## A Simple 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 exprs with a *
- Coalesce results ?

***Options***

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

# Renaming with SSA number

**Example (continued)**

| 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
- Simple rewriting now works

# Simple extension

- **Constant Folding**
- • Add a field to the hash table that records when a value is constant
- • Evaluate constant values at compile-time
- • Replace with load immediate or immediate operand
- • No stronger local algorithm
- **Algebraic Identities**
- • Must check (many) special cases
- • Replace result with input VN
- • Build a decision tree on operation
- – No obvious way to hash

**Identities (on VNs)**

X←y, x+0, x-0, x*1, x÷1, x-x, x*0, x÷x, xV0, x ^ 0xFF…FF, max(x,MAXINT), min(x,MININT), max(x,x), min(y,y), and so on …

# Extended application

*for **i** ← 0 to n-1*

*1. get the value numbers V1 and V2 for Li and Ri*

*2. if Li and Ri are both constant then* — Constant folding

*evaluate Li Opi Ri, assign it to Ti, and mark Ti as a constant*

*3. if Li Opi Ri matches an identity then* — Algebraic identities

*replace operation **i** with a copy operation or an assignment*

*4. if Opi commutes and V1 > V2 then* — Commutativity

*swap V1 and V2*

*5. construct a hash key <V1,Opi,V2>*

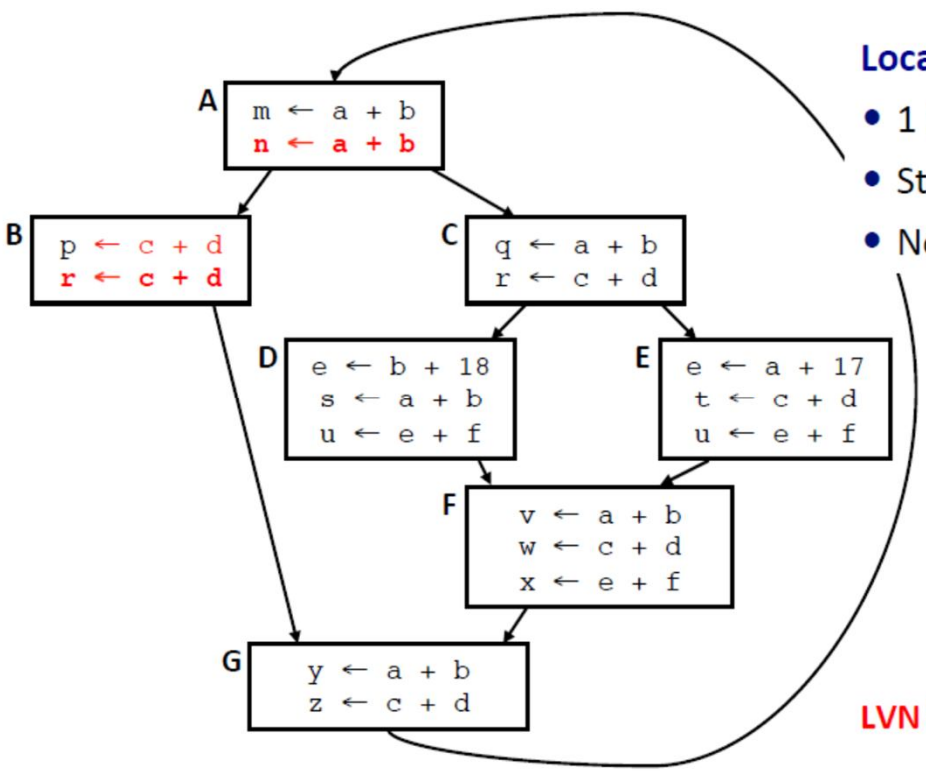
*6. if the hash key is already present in the table then*

*replace operation **i** with a copy into Ti and mark Ti with the VN*

*else*

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

# Performance of LVN

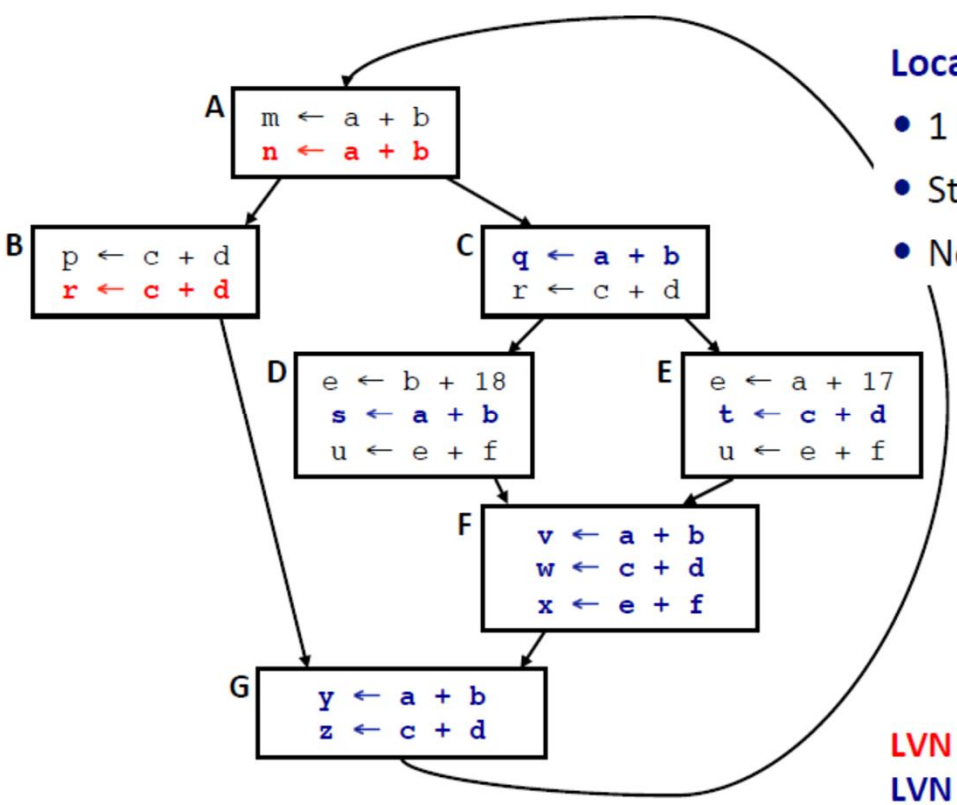



**Local Value Numbering (LVN)**

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

**LVN finds redundant ops in red**

# Performance continued

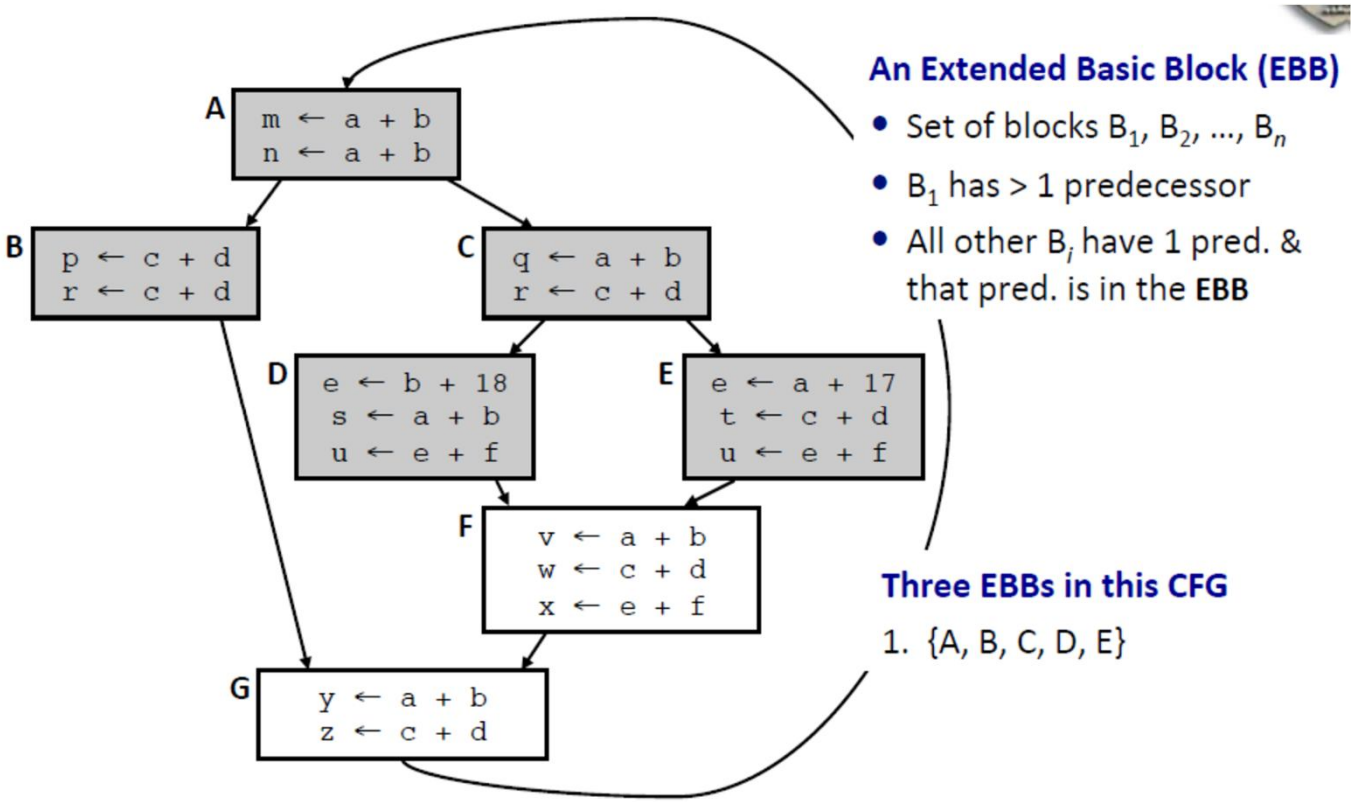



**Local Value Numbering (LVN)**

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

**LVN finds redundant ops in red**
**LVN misses redundant ops in blue**

# Regional optimization- EBB



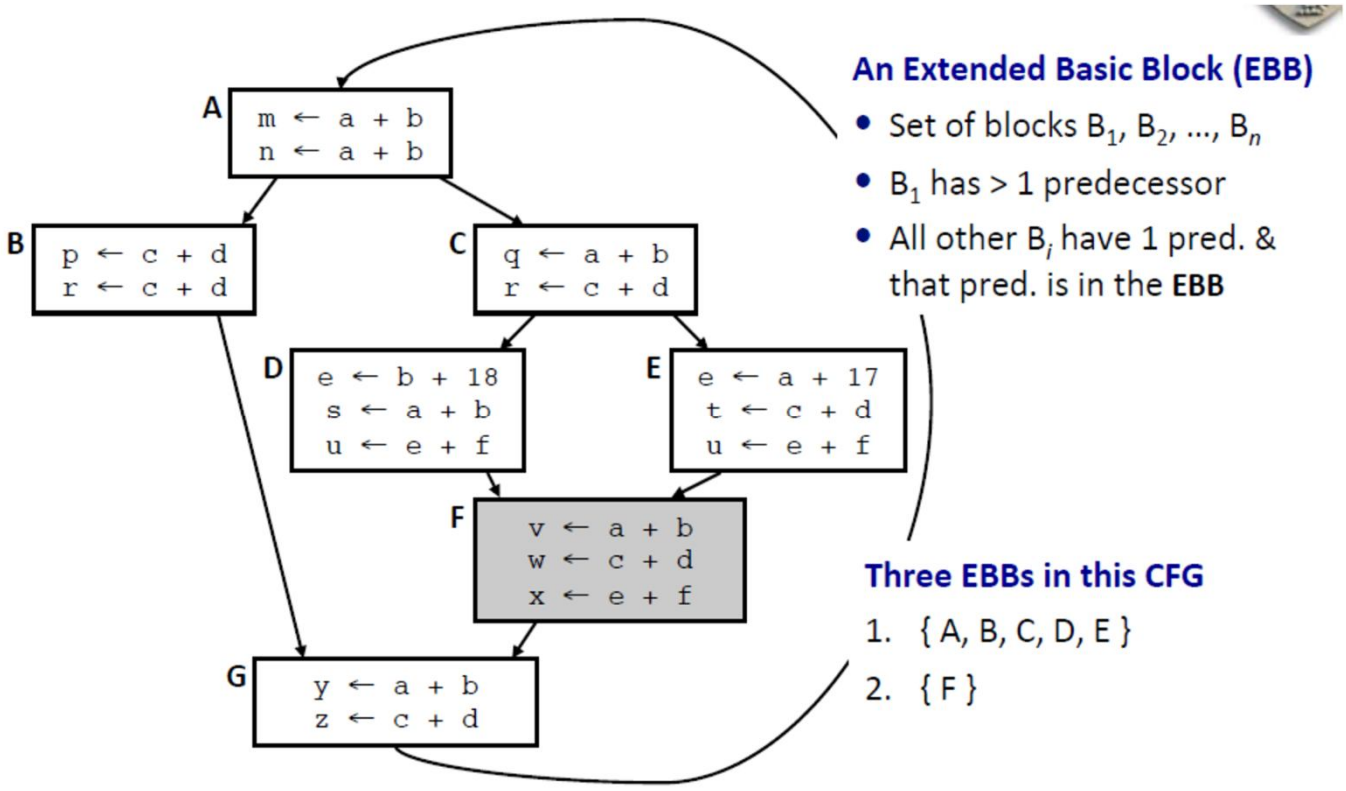

**An Extended Basic Block (EBB)**

- Set of blocks $B_1$, $B_2$, ..., $B_n$
- $B_1$ has > 1 predecessor
- All other $B_i$ have 1 pred. & that pred. is in the **EBB**

**Three EBBs in this CFG**

1. {A, B, C, D, E}

# EBB example



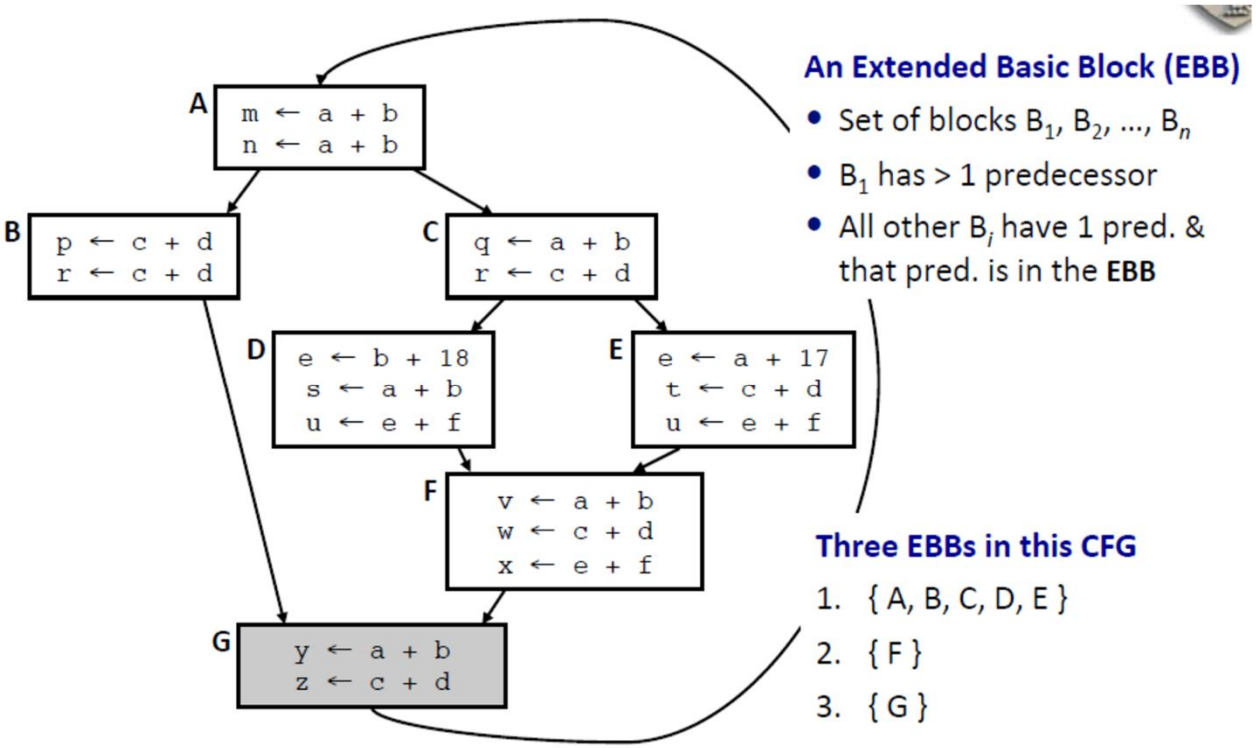

**An Extended Basic Block (EBB)**

- Set of blocks $B_1$, $B_2$, ..., $B_n$
- $B_1$ has > 1 predecessor
- All other $B_i$ have 1 pred. & that pred. is in the **EBB**

**Three EBBs in this CFG**

1. { A, B, C, D, E }
2. { F }

# EBB example



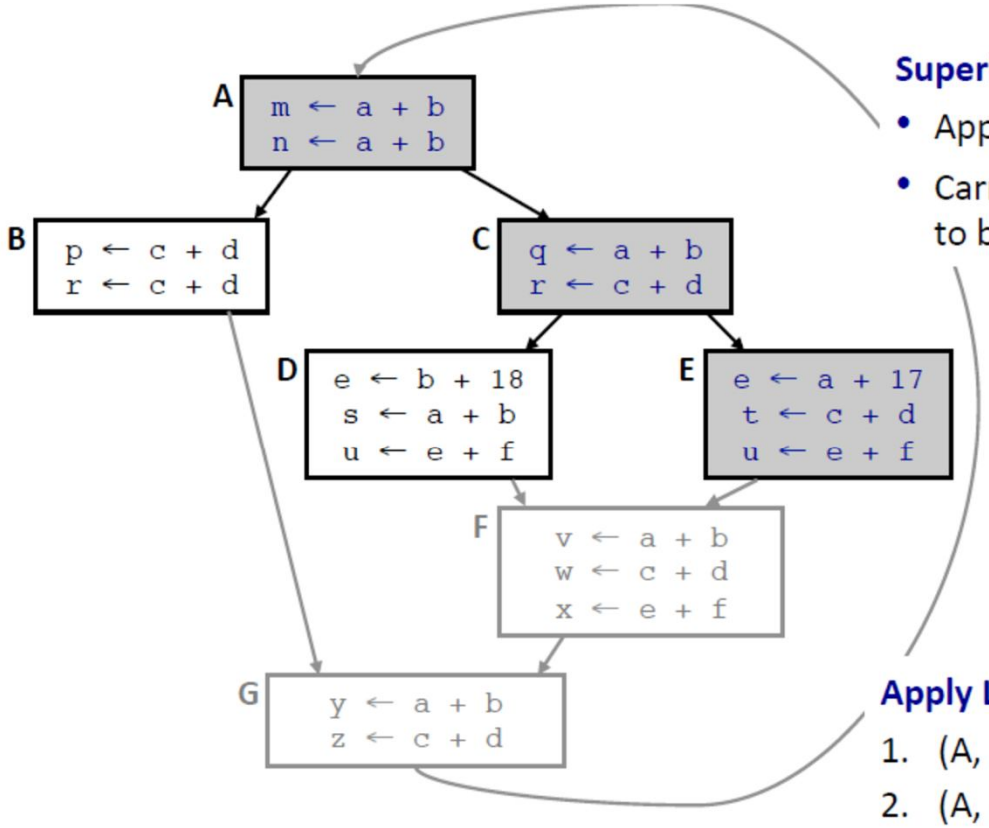

**An Extended Basic Block (EBB)**

- Set of blocks $B_1$, $B_2$, …, $B_n$
- $B_1$ has > 1 predecessor
- All other $B_i$ have 1 pred. & that pred. is in the **EBB**

**Three EBBs in this CFG**

1. { A, B, C, D, E }
2. { F }
3. { G }

# Regional optimization SVN



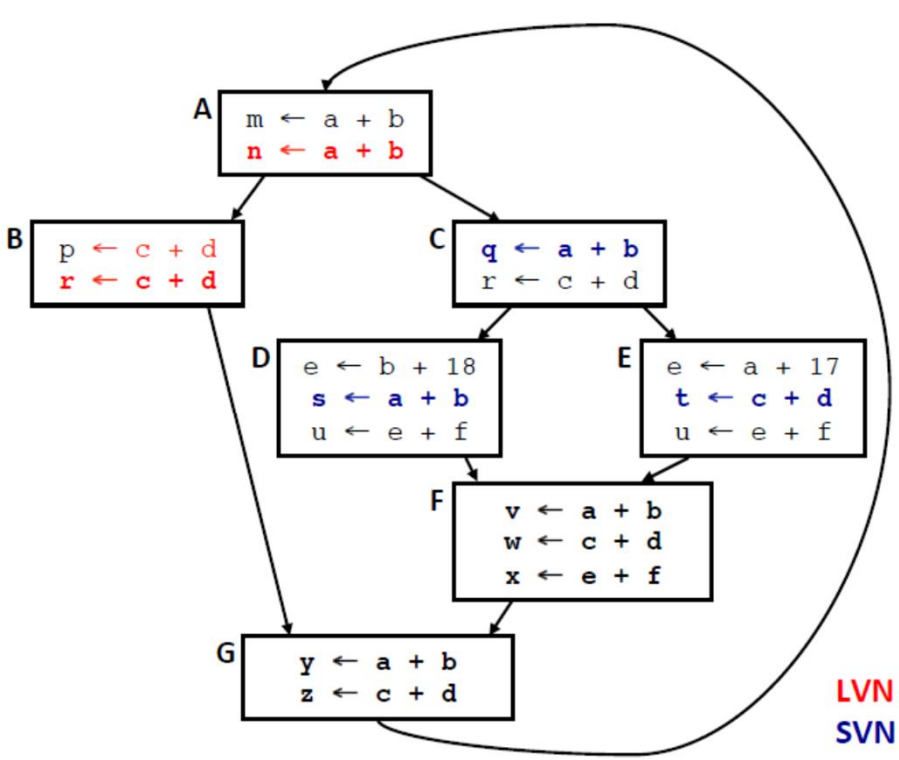

**Superlocal VN**

- Apply **LVN** to each path in EBB
- Carry hash table forward, block to block

**Apply LVN to each path in EBB**

1. (A, B)
2. (A, C, D)
3. (A, C, E)

# Performance of SVN




LVN finds redundant ops in red

SVN finds redundant ops in blue

***Both miss redundancies in F & G***