CENG513 Compiler Design and Construction
Code Optimizations

Note by Işıl ÖZ:

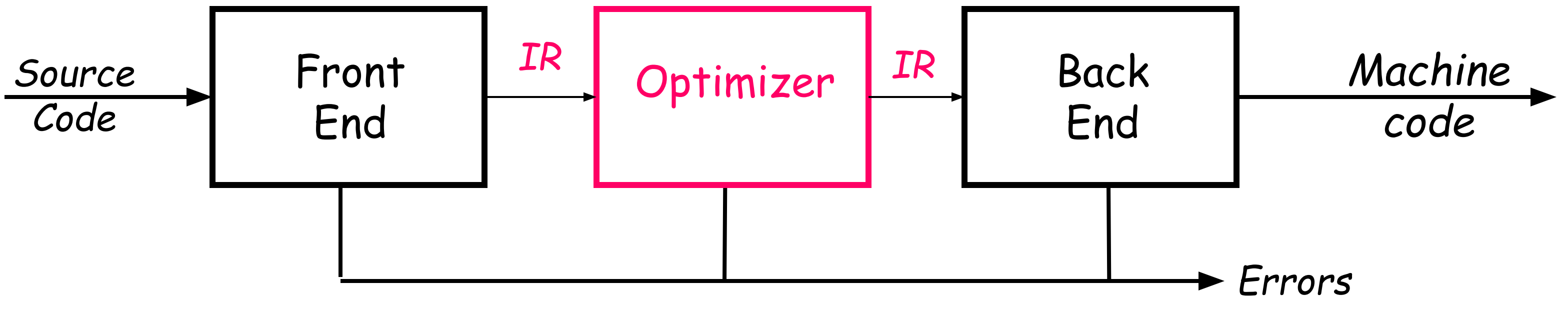
Our slides are adapted from Cooper and Torczon's slides that are prepared for COMP 412 at Rice.

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.

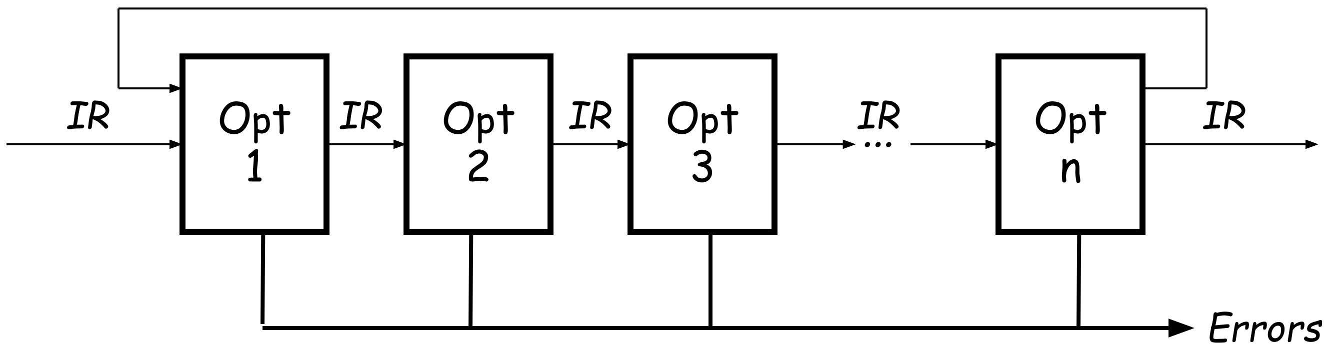
Traditional Three-Phase Compiler



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

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

The Limitations of the Optimizer

- Operate under fundamental constraint
  - Must not cause any change in program behavior
  - Often prevents it from making optimizations that would only affect behavior under pathological conditions
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
- Most analysis is based only on static information
- <u>When in doubt, the compiler must be conservative</u>

Memory Matters

Code updates b[i] on every iteration

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
    b[i] = 0;
    for (j = 0; j < n; j++)
        b[i] += a[i*n + j];
    }
}
```

Potential Optimization

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
    b[i] = 0;
    for (j = 0; j < n; j++)
        b[i] += a[i*n + j];
    }
}
```

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
    double val = 0;
    for (j = 0; j < n; j++)
        val += a[i*n + j];
         b[i] = val;
    }
}
```

Optimization Blocker: Memory Aliasing

Two different memory references specify single location

Compiler cannot determine whether pointers may be aliased, limits optimization

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
    b[i] = 0;
    for (j = 0; j < n; j++)
        b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of **B**:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

Get in habit of introducing local variables

Procedure Calls

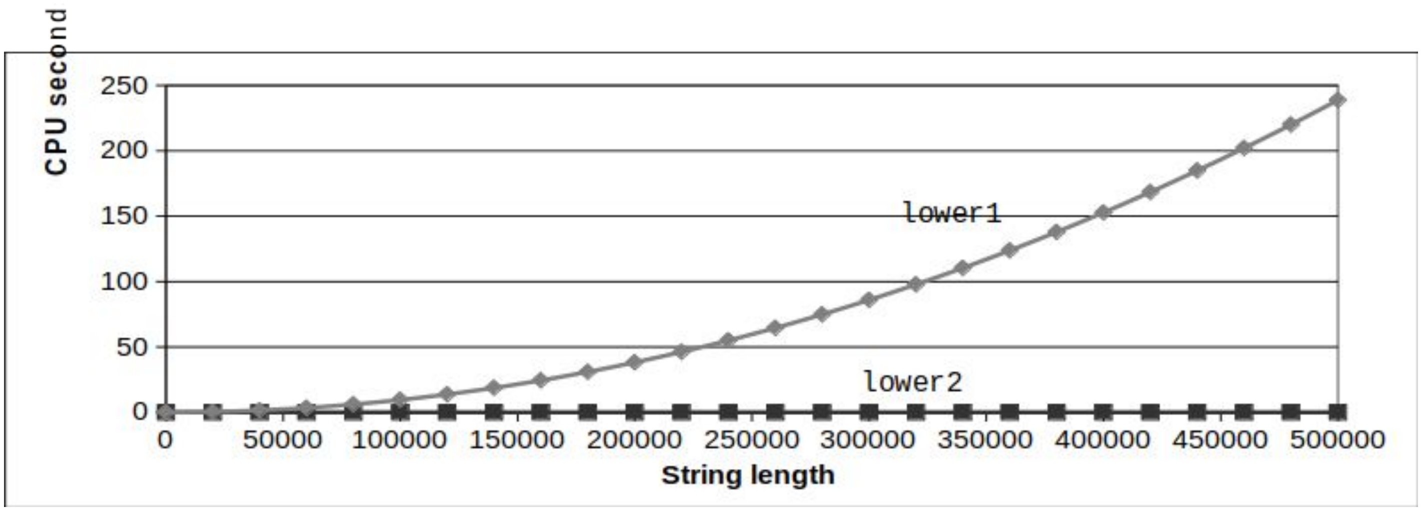
```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

Potential Optimization

```
void lower1(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
void lower2(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```



Optimization Blocker: Procedure Calls

Why not compiler move strlen out of inner loop?

```
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++; length++;
    }
    lencnt += length;
    return length;
}
```

Procedure may have side effects

- Alters global state each time called
- Function may not return the same value for given arguments
- Depends on other parts of global state
- Procedure lower could interact with strlen

Compiler treats procedure call as a black box

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.

Basic Block

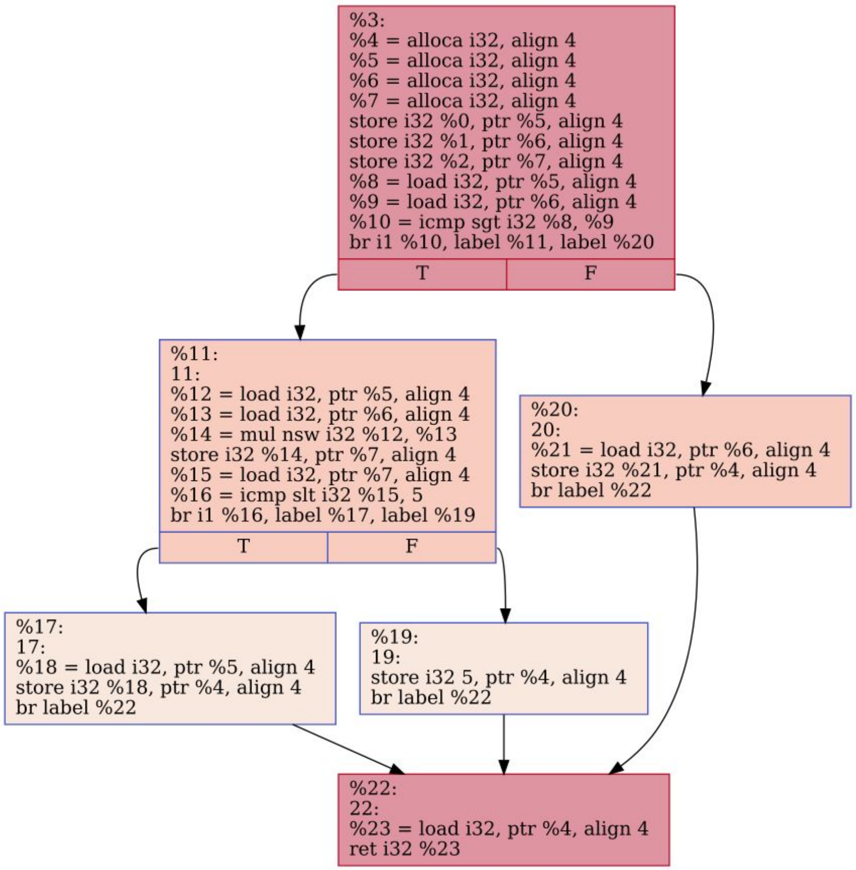
Sequence of operations that always execute together

```c
int BB(int a, int b, int x) {
   if (a > b) {
       x = a * b;
       if(x < 5)
           return a;
       else
           return 5;
   } else {
       return b;
   }
}
```

```llvm
define dso_local i32 @BB(i32 noundef %0, i32 noundef %1, i32 noundef %2) #0 {
 ...

 %10 = icmp sgt i32 %8, %9
 br i1 %10, label %11, label %20

11:                                               ; preds = %3
 ...
 %14 = mul nsw i32 %12, %13
 store i32 %14, ptr %7, align 4
 %15 = load i32, ptr %7, align 4
 %16 = icmp slt i32 %15, 5
 br i1 %16, label %17, label %19

17:                                               ; preds = %11
 %18 = load i32, ptr %5, align 4
 store i32 %18, ptr %4, align 4
 br label %22

19:                                               ; preds = %11
 store i32 5, ptr %4, align 4
 br label %22

20:                                               ; preds = %3
 %21 = load i32, ptr %6, align 4
 store i32 %21, ptr %4, align 4
 br label %22

22:                                               ; preds = %20, %19, %17
 %23 = load i32, ptr %4, align 4
 ret i32 %23
}
```

Basic Block

Sequence of operations that always execute together



CFG for 'BB' function

```
define dso_local i32 @BB(i32 noundef %0, i32 noundef %1, i32 noundef %2) #0 {
 ...

 %10 = icmp sgt i32 %8, %9
 br i1 %10, label %11, label %20

11:                                               ; preds = %3
 ...
 %14 = mul nsw i32 %12, %13
 store i32 %14, ptr %7, align 4
 %15 = load i32, ptr %7, align 4
 %16 = icmp slt i32 %15, 5
 br i1 %16, label %17, label %19

17:                                               ; preds = %11
 %18 = load i32, ptr %5, align 4
 store i32 %18, ptr %4, align 4
 br label %22

19:                                               ; preds = %11
 store i32 5, ptr %4, align 4
 br label %22

20:                                               ; preds = %3
 %21 = load i32, ptr %6, align 4
 store i32 %21, ptr %4, align 4
 br label %22

22:                                               ; preds = %20, %19, %17
 %23 = load i32, ptr %4, align 4
 ret i32 %23
}
```

```c
int BB(int a, int b, int x) {
    if (a > b) {
        x = a * b;
        if(x < 5)
            return a;
        else
            return 5;
    } else {
        return b;
    }
}
```

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

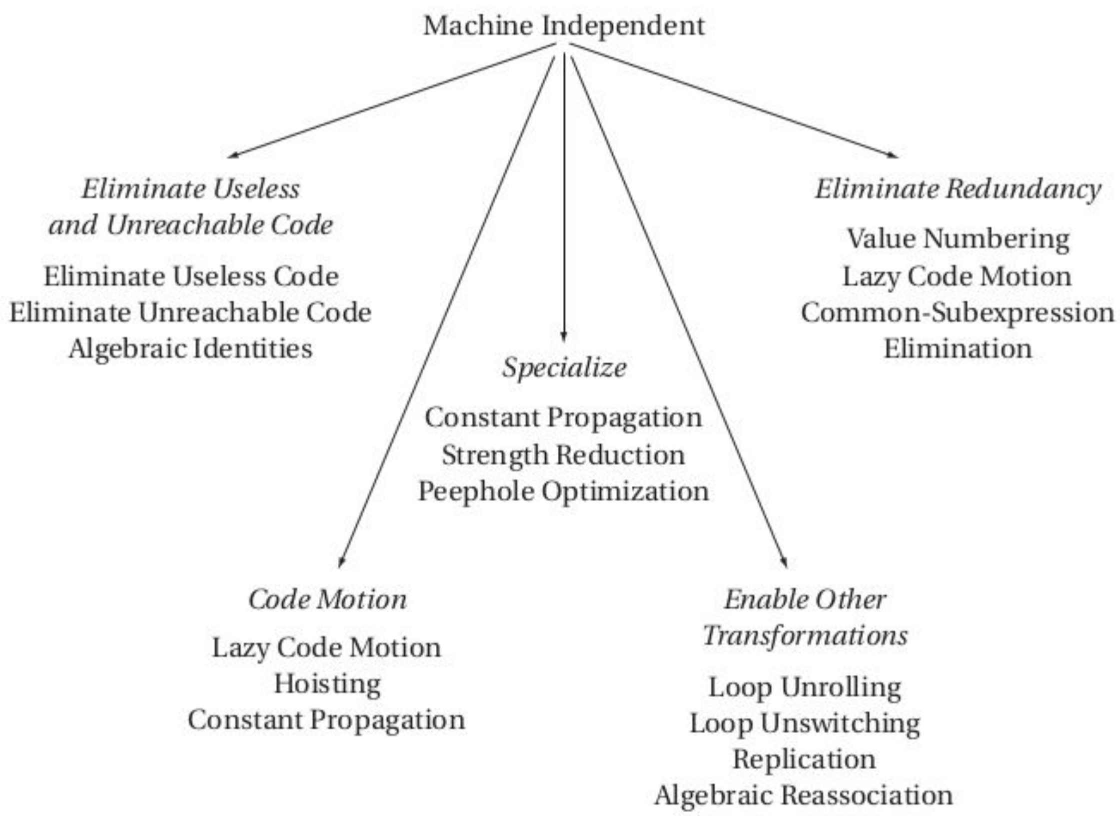
Whole procedure optimization (*intraprocedural*)

- 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

Machine-Independent Transformations



Machine-Independent Transformations

Eliminate useless and unreachable code

- If an operation is either useless or unreachable

Code motion

- Move an operation to a place where it executes less frequently

Specialize a computation

- If the compiler can understand the specific context in which an operation will execute

Enable other transformations

- If the compiler can rearrange the code in a way that exposes more opportunities for other transformations

Eliminate redundancy

- If the compiler can prove that a computation is redundant

Dead Code Elimination - DCE

```
//Code
int x = a+23;              //the variable x is never used
                           //in the program. Thus it is a dead code.
z = a + y;
printf("%d %d",z,y);
```

```
//After Optimization
z = a + y;
printf("%d %d",z,y);
```

Unreachable Code

```
//Code
int foo(void)
{
     int a = 24;
     int b = 25;   //Assignment to dead variable
     int c;
     c = a * 4;
     return c;
     b = 24;
     return 0;     //Unreachable code
 }
```

```
//After optimization
int foo(void)
{
     int a = 24;
     int c;
     c = a * 4;
     return c;
}
```

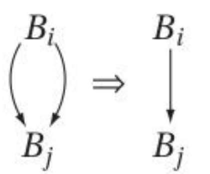
Algebraic Identities

```
//Code
b = a * 1;
c = d + 0;
```

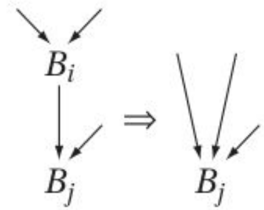
```
//After optimization
b = a;
c = d;
```

$a + 0 = a$	$a - 0 = a$	$a - a = 0$	$2 \times a = a + a$
$a \times 1 = a$	$a \times 0 = 0$	$a \div 1 = a$	$a \div a = 1, a \neq 0$
$a^1 = a$	$a^2 = a \times a$	$a \gg 0 = a$	$a \ll 0 = a$
$a \text{ AND } a = a$	$a \text{ OR } a = a$	$\text{MAX}(a,a) = a$	$\text{MIN}(a,a) = a$

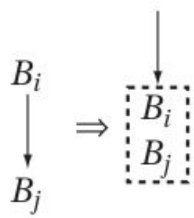
Eliminating Useless Control Flow - Fig 10.4



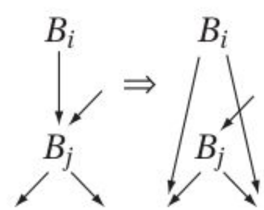
(1) Folding a Redundant Branch



(2) Removing an Empty Block



(3) Combining Blocks



(4) Hoisting a Branch

Loop Invariant Code Motion

```
//Code
t = 10;
for(i = 0; i < 10; i++){
     a = t * 10;          //Loop invariant code
     b = 20 * i;
     y += a * b;
}
```

```
//After optimization
t = 10;
a = t * 10;
for(i = 0; i < 10; i++){
     b = 20 * i;
     y += a * b;
}
```

Constant Propagation - Constant Folding

```
//Code
x = 3;
y = x + 4;
```

```
//After constant propagation
y = 3 + 4;

//After constant folding
y = 7;
```

Common Subexpression Elimination

```
//Code
a = 10;
b = a + 1 * 2;
c = a + 1 * 2;      //'c' has common expression as 'b'
d = c + a;
```

```
//After transformation
a = 10;
b = a + 1 * 2;
d = b + a;
```

Strength Reduction

```
//Code
c = 4;
for (i = 0; i < N; i++)
{
    y[i] = c * i;
}
```

```
//Replace by cumulative addition
c = 4;
k = 0;
for (i = 0; i < N; i++)
{
    y[i] = k;
    k = k + c;
}
```

```
//Replace by shift left
c = 4;
for (i = 0; i < N; i++)
{
    y[i] = i << 2;
}
```

Loop Interchange

```
//Code
for (j = 0; j < M; j++)
{
     for (i = 0; i < N; i++)
     {
          a[i][j] = i + j;
     }
}
```

```
//After transformation
for (i = 0; i < N; i++)
{
     for (j = 0; j < M; j++)
     {
          a[i][j] = i + j;
     }
}
```

Loop Unrolling

```
//Code
for (i = 0; i < 10000; i++)
{
     a[i] = i + 17;
}
```

```
//After transformation
for (i = 0; i < 10000; i+=4)
{
     a[i] = i + 17;
     a[i+1] = (i+1) + 17;
     a[i+2] = (i+2) + 17;
     a[i+3] = (i+3) + 17;
}
```

Loop Fusion

```
//Code
for (i = 0; i < 300; i++)
  a[i] = a[i] + 3;

for (i = 0; i < 300; i++)
  b[i] = b[i] + 4;
```

```
//After transformation
for (i = 0; i < 300; i++)
{
    a[i] = a[i] + 3;
    b[i] = b[i] + 4;
}
```

Loop Unswitching

```
//Code
for (i = 0; i < n; i++)
{
  if(x > y)              //Loop invariant control-flow operation
     a[i] = b[i] * x;
  else
     a[i] = b[i] * y;
}
```

```
//After transformation
if(x > y)
{
     for (i = 0; i < n; i++)
          a[i] = b[i] * x;
}else
     for (i = 0; i < n; i++)
          a[i] = b[i] * y;
}
```

Function Inlining

```
//Code
int pred(int x)
{
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y)
{
    return pred(y) + pred(0) + pred(y+1);
}
```

```
//After transformation
int func(int y)
{
    int tmp;
    if (y   == 0) tmp  = 0; else tmp  = y - 1;       /* (1) */
    if (0   == 0) tmp += 0; else tmp += 0 - 1;       /* (2) */
    if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1; /* (3) */
    return tmp;
}
```

Useless Code Elimination as an Example

Mark-sweep collectors

The first walk:

- marks critical operations if it
  - sets return values for the procedure
  - is an input/output statement
  - affects the value in a storage location that may be accessible from outside the procedure

Examples of critical operations include code in the procedure's entry and exit blocks and calls to other procedures

- traces the operands of critical operations back to their definitions and marks those operations as useful

The second pass walks the code and removes any operation not marked as useful

Useless Code Elimination as an Example

When is a branch useful?

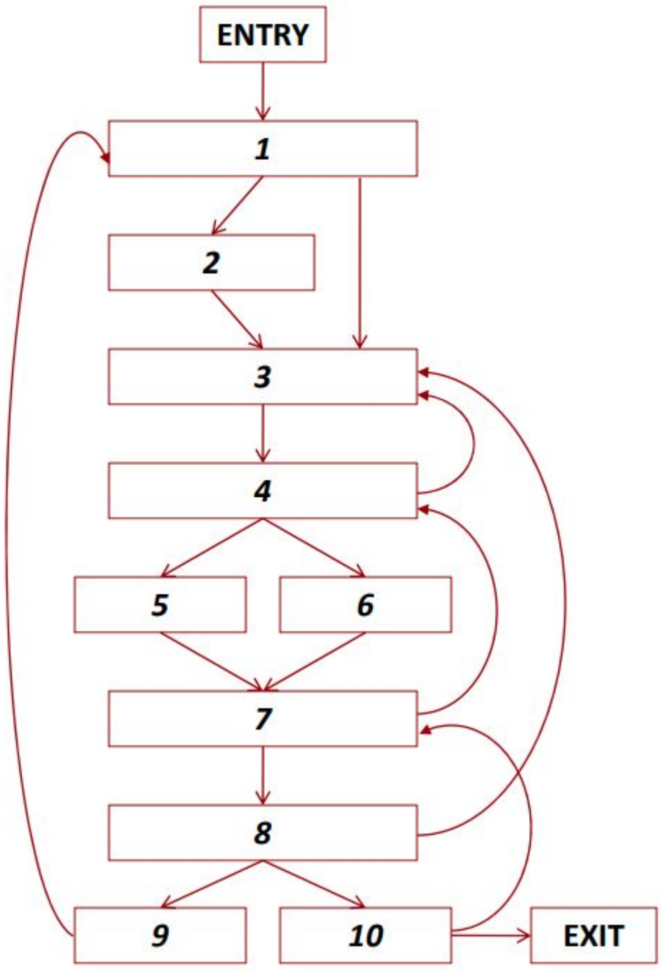
> In the CFG, *j* is <u>control dependent</u> on *i* if
>
> 1. ∃ a non-null path *p* from *i* to *j* ∋ *j* post-dominates every node on *p* after *i*
> 2. *j* does not strictly post-dominate *i*

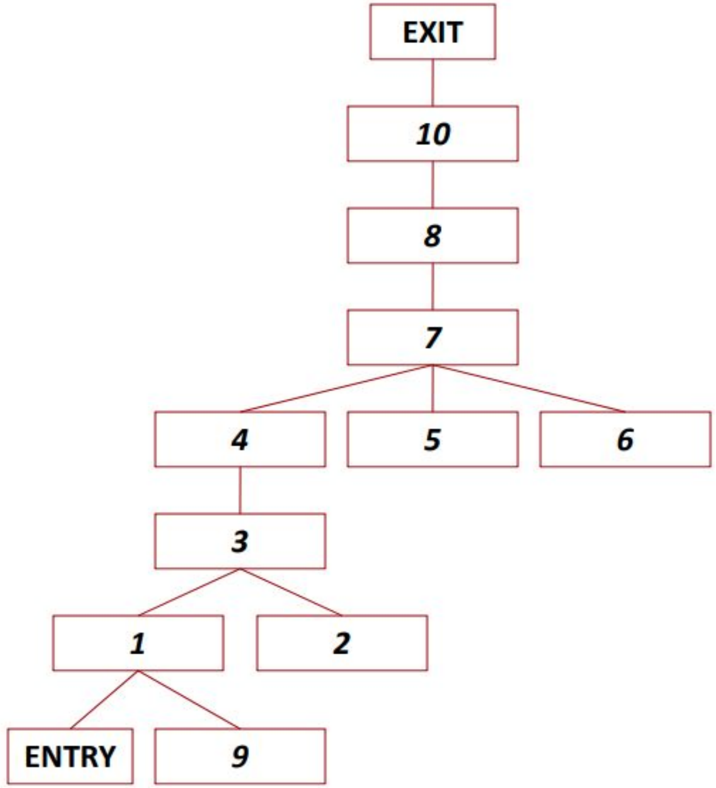
- j control dependent on i one path from i leads to j, one doesn't
- This is the reverse dominance frontier of j (RDF(j))

node j <u>postdominates</u> node i if every path from i to the cfg 's exit node passes through j

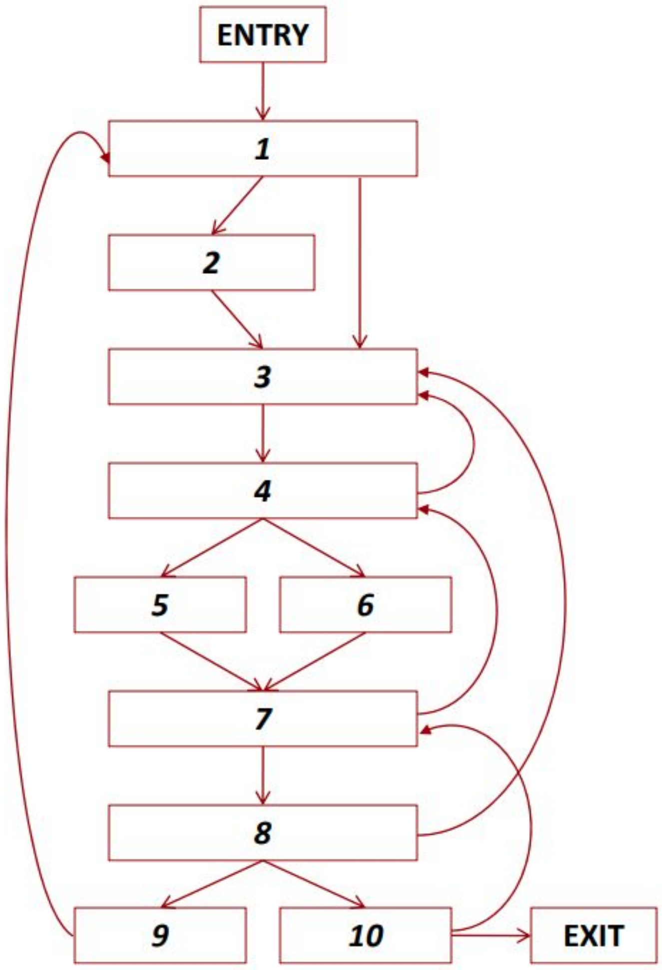
node d <u>dominates</u> node n if every path from ENTRY to n goes through d

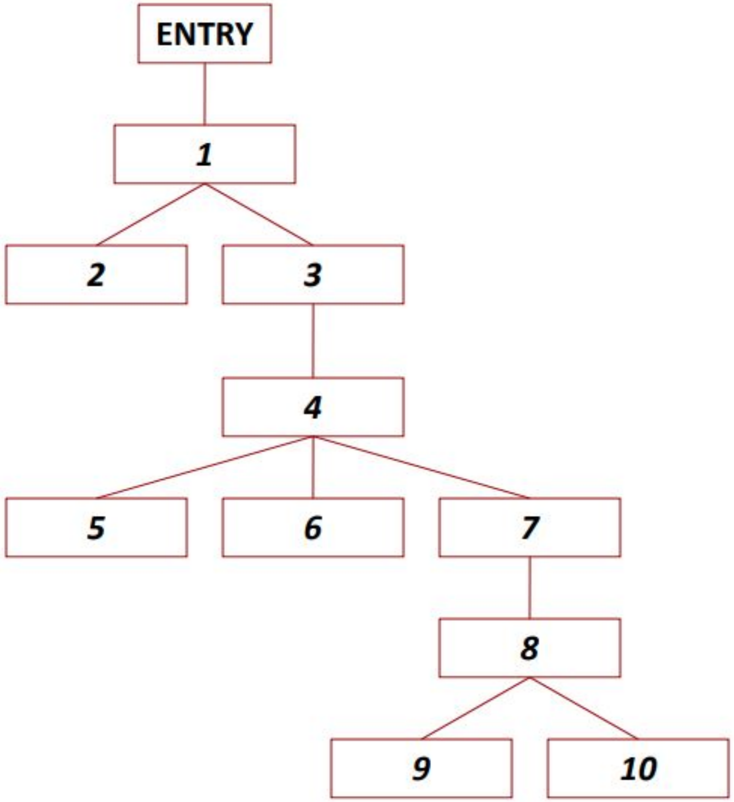
Post Dominator Tree





Dominator Tree





Useless Code Elimination as an Example

```
Dead( )
  MarkPass( )
  SweepPass( )

SweepPass( )
  for each operation i
    if i is unmarked then
      if i is a branch then
        rewrite i with a jump
          to i's nearest marked
          postdominator
      if i is not a jump then
        delete i
```

```
MarkPass( )
  WorkList ← ∅
  for each operation i
    clear i's mark
    if i is critical then
      mark operation i
      WorkList ← WorkList ∪ {i}
  while (WorkList ≠ ∅)
    remove i from WorkList
      (assume i is x ← y op z)
    if def(y) is not marked then
      mark def(y)
      WorkList ← WorkList ∪ {def(y)}
    if def(z) is not marked then
      mark def(z)
      WorkList ← WorkList ∪ {def(z)}
    for each block b ∈ RDF(block(i))
      let j be the branch that ends b
      if j is unmarked then
        mark j
        WorkList ← WorkList ∪ {j}
```

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 *avail*
- Rewriting the code to eliminate the redundant evaluation

$$\begin{array}{l} m \leftarrow 2 \times y \times z \\ n \leftarrow 3 \times y \times z \\ o \leftarrow 2 \times y - z \end{array}$$

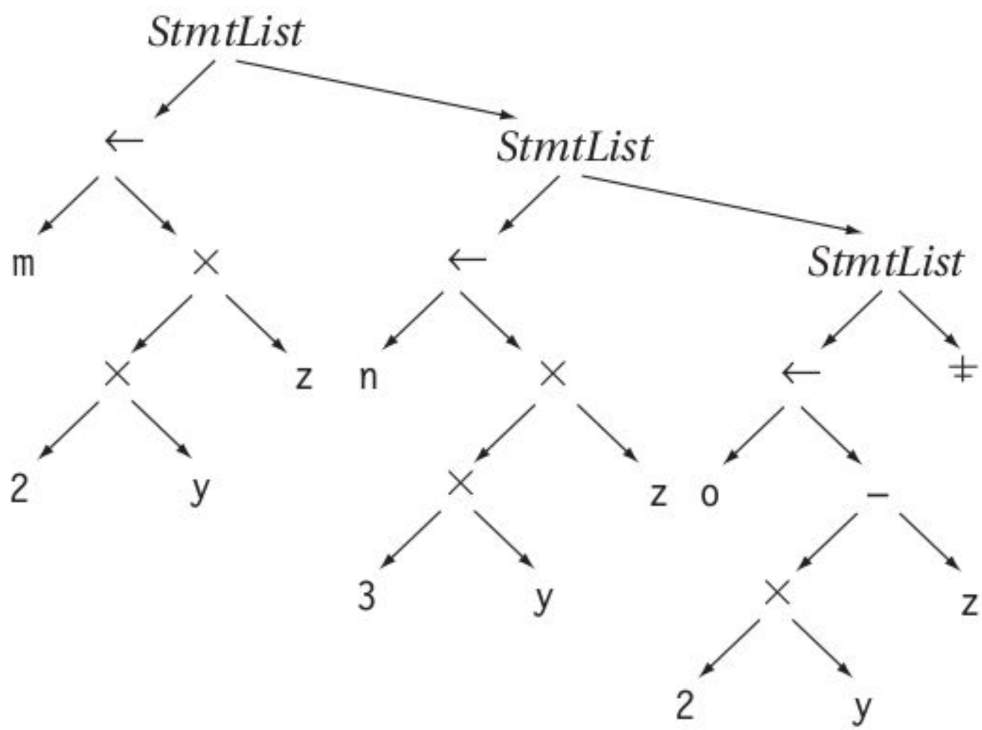
$$\begin{array}{l} t_0 \leftarrow 2 \times y \\ m \leftarrow t_0 \times z \\ n \leftarrow 3 \times y \times z \\ o \leftarrow t_0 - z \end{array}$$

Techniques for accomplishing both: *DAG building* and *value numbering*

$$m \leftarrow 2 \times y \times z$$
$$n \leftarrow 3 \times y \times z$$
$$o \leftarrow 2 \times y - z$$

Building a Directed Acyclic Graph

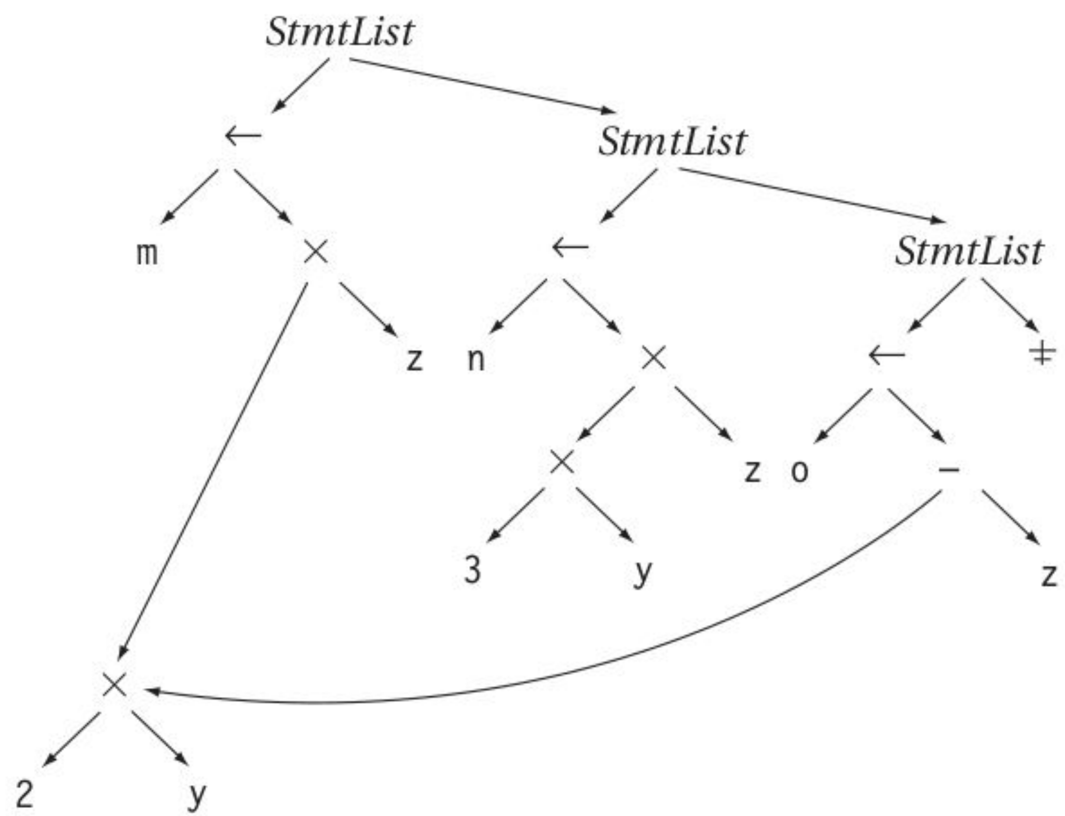
AST for the expression



$m \leftarrow 2 \times y \times z$
$n \leftarrow 3 \times y \times z$
$o \leftarrow 2 \times y - z$

Building a Directed Acyclic Graph

DAG for the expression enables explicit redundancy representation



Value Numbering

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

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
4. If o_1 & o_2 are constant, evaluate it & replace with a loadI

Handling algebraic identities

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

Local Value Numbering

An example

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

Local Value Numbering

An example

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

Rewrite code in a way that gives each assignment a distinct name (SSA form)

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

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)
>
> $x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x \div 1$, $x-x$, $x*0$, $x \div x$, $x \vee 0$, $x \wedge$ 0xFF…FF, max(x,MAXINT), min(x,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

Scope of Optimization

Local Methods

- operations that all occur in the same block

Superlocal Methods

- operate over extended basic blocks (EBBs)

Regional Methods

- scopes larger than a single ebb but smaller than a full procedure

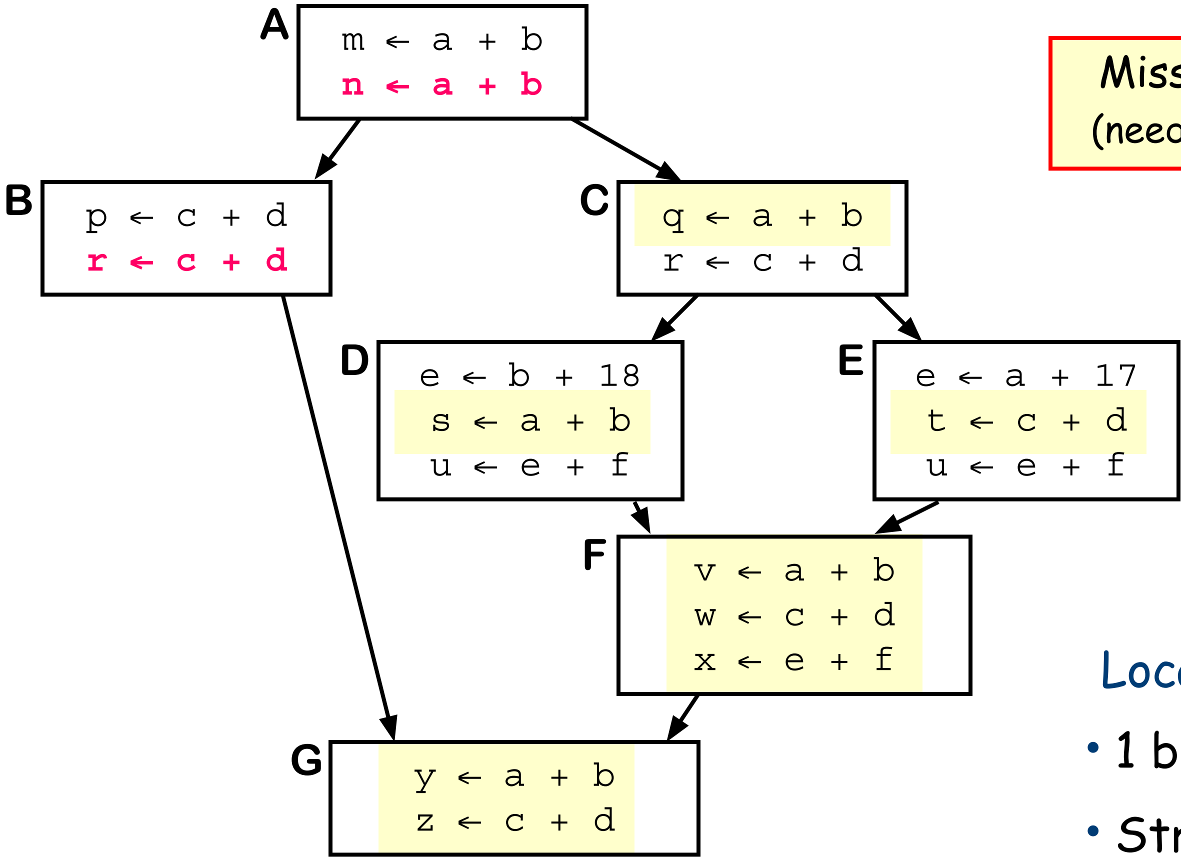
Global Methods

- intraprocedural methods, examine an entire procedure

Whole-Program Methods

- interprocedural methods, consider the entire program as their scope (any transformation that involves more than one procedure as an interprocedural transformation)

Value Numbering

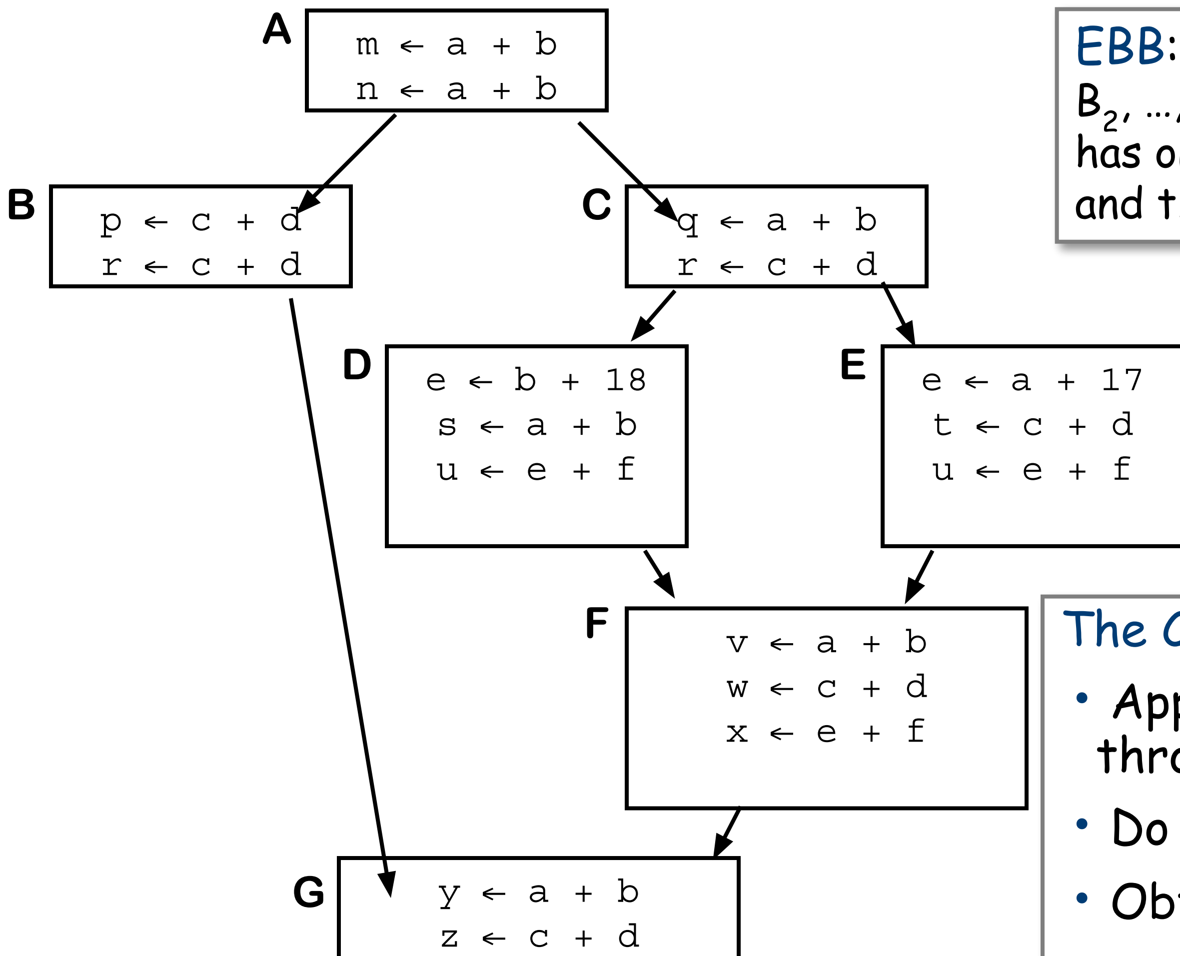


Missed opportunities
(need stronger methods)

Local Value Numbering

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

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

Profile-Guided Optimizations

The compiler accesses profile data

- from a sample run of the program
- across a representative input set

The results indicate which areas of the program are executed more frequently, and which areas are executed less frequently

The sample of data fed to the program during the profiling stage must be statistically representative of the typical usage scenarios

Profile information

- Branch mispredictions
- Cache misses
- Execution counts

Types of profiling

- Instrumentation
- Sampling

References

Chapter sections from the book:

- 8.3, 8.4, 10.2, 10.3

- Randal E. Bryant, David R. O'Hallaron: Computer Systems: A Programmer's Perspective, Pearson, 2011.

Selected videos from compiler course from California State University:

- https://www.youtube.com/watch?v=WrtT3Xwbt4s&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=36
- https://www.youtube.com/watch?v=zTey9_rPYSI&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=37
- https://www.youtube.com/watch?v=himxl1YiB6c&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=38

LLVM Transform Passes

- https://llvm.org/docs/Passes.html#transform-passes