## More Optimizations

### Last Time

- Loop invariant code motion
- Loop induction variables

### Today

- Global Common Subexpression Elimination
- Value Numbering



## Common Subexpression Elimination (Example)

Given A[i][j] = A[i][j] + 1, and assuming

1. row-major order
2. each array element is 4 bytes, and
3. R rows in the array

will yield the following 3-address intermediate code:

```
t0 = A
t1 = R * 4
t2 = t1 * i
t3 = t0 + t2
t4 = j * 4
t5 = t3 + t4
t6 = [t5]

t7 = t6 + 1

t8 = A
t9 = R * 4
t10 = t8 * i
t11 = t8 + t10
t12 = j * 4
t13 = t11 + t12
[t13] = t7
```

What are the common sub-expressions?

## Global Common Subexpression Elimination

**The Goal**

to find common subexpressions whose range spans basic blocks, *and* eliminate unnecessary re-evaluations

**Safety**

- available expressions (AVAIL) proves that value is current
- transformation gives value the right name

**Profitability**

- don't add evaluations to any path
- add some `copy` instructions

  ⇒ *copy is as cheap as any operator*

  ⇒ *may shrink, may stretch live ranges*



## Available expressions

For a block $b$

- AVAIL($b$) the set of expressions available on entry to $b$.
- NKILL($b$) the set of expressions not killed in $b$.
- DEF($b$) the set of expressions defined in $b$ and not subsequently killed in $b$.

$$\text{AVAIL}(b) = \bigcap_{x \in pred(b)} \text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x))$$

IN(b) =

OUT(b) =

## AVAIL

What expressions are available at the end of this basic block?

```
t0 = A
t1 = R * 4
t2 = t1 * i
t3 = t0 + t2
t4 = j * 4
t5 = t3 + t4
t6 = [t5]
```

How do we compute AVAIL for a basic block?



## AVAIL

Returning to our example. How can we detect that t2 and r10 compute the same value?

```
t0 = A
t1 = R * 4
t2 = t1 * i
t3 = t0 + t2
t4 = j * 4
t5 = t3 + t4
t6 = [t5]

t7 = t6 + 1

t8 = A
t9 = R * 4
t10 = t8 * i
t11 = t8 + t10
t12 = j * 4
t13 = t11 + t12
[t13] = t7
```

## Value Numbering

Value numbering computes available expressions (AVAIL or DEF) for a basic block.

**Input**

basic block of tuples (*statements*)

symbol table

**Output**

improved basic block (*cse, constant folding*)

table of available expressions†

table of constant values

† An expression is *available* at point $p$ if it is defined along each path leading to $p$ and none of its constituent values has been subsequently redefined.



## Value Numbering

**Key Notions**

- each *variable*, each *expression*, and each *constant* is assigned a unique number, its *value number*
  - same number ⇒ same value
  - based solely on information from within the block
- if an expression's value is *available* (already computed), we should *not* recompute it
- constants denoted in both SYMBOLS and tuples

## Value numbering

Principle data structures

CODE

- array of tuples
- Fields: result, operator (*op*), operands (*lhs, rhs*)

SYMBOLS

- hash table keyed by variable name
- Fields: name, val, isConstant

AVAIL

- hash table keyed by (*lhs, op, rhs*)
- Fields: lhsVal, op, rhsVal, resultVal, tuple

CONSTANTS

- table to hold funky machine specific values
- important in cross-compilation
- Fields: val, bits



## Value numbering result = lhs op rhs

```
for i ← 1 to n
    r ← value number for rhs[i]
    l ← value number for lhs[i]
    if op[i] is a store then
        SYMBOLS[result[i]].val ← r
        if r is constant then
            SYMBOLS[lhs[i]].isConstant ← true
    else /* an expression */
        if l is constant then replace lhs[i] with constant
        if r is constant then replace rhs[i] with constant
        if l and r are both constant then
            create CONSTANTS(l,op[i],r)
            CONSTANTS(l,op[i],r).bits ← eval(l op[i] r)
            CONSTANTS(l,op[i],r).val ← new value number
            replace (lhs op[i] rhs) with " constant (l op[i] r)"
        else if (l,op[i],r) ∈ AVAIL then
            replace (lhs op[i] rhs) with "AVAIL(l,op[i],r).result[j]"
        else create AVAIL(l,op[i],r)
            AVAIL(l,op[i],r).val ← new value number
        endif
        SYMBOLS[result[i]].val ← value number of expression
    endif
    for all AVAIL(l,op[j],r)
        if result[i].val = l or r or result[j].val, (j < i)
            remove (l,op[j],r) from AVAIL
    endfor
endfor
```

## Example

| *Tuples* | *Source* | *Avail* |
|---|---|---|
| a ← C4 | a ← 4 | |
| T2 ← i × j | | |
| T3 ← T2 + C5 | | |
| k ← T3 | k ← i × j + 5 | |
| T5 ← C5 × a | | |
| T6 ← T5 × k | | |
| l ← T6 | l ← 5 × a × k | |
| m ← i | m ← i | |
| T9 ← m × j | | |
| 10 ← i × a | | |
| T11 ← T9 + T10 | | |
| b ← T11 | b ← m × j + i × a | |



## Example

| | **Original** | **SSA Form** |
|---|---|---|
| | $A \leftarrow X + Y$ | $A_0 \leftarrow X_0 + Y_0$ |
| | $B \leftarrow X + Y$ | $B_0 \leftarrow X_0 + Y_0$ |
| | $A \leftarrow 1$ | $A_1 \leftarrow 1$ |
| (1) | $C \leftarrow A + Y$ | $C_0 \leftarrow A_1 + Y_0$ |
| | $B \leftarrow A$ | $B_1 \leftarrow A_1$ |
| | $C \leftarrow 3$ | $C_1 \leftarrow 3$ |
| (2) | $D \leftarrow B + Y$ | $D_0 \leftarrow B_1 + Y_0$ |

## Global Common Subexpression Elimination

**Algorithm**:

1. $\forall$ block $b$, compute DEF($b$) and NKILL($b$)
2. $\forall$ block $b$, compute AVAIL($b$)
3. $\forall$ block $b$, value number the block using AVAIL
4. replace expressions in AVAIL($b$) with references

Computing DEF($b$) and NKILL($b$)

- use value numbering, *or*
- do it by inspection

Specifics

1. $\forall$ block $b$, compute DEF($b$) and KILL($b$)
2. assign each $e \in$ AVAIL($b$) a *name* (*small integer*)
3. $\forall$ variables $v$, initialize MAP($v$) to empty
4. $\forall$ expressions $e$, $\forall v \in e$, add *name* to MAP($v$)
5. $\forall$ block $b$, $\text{NKILL}(b) = \bigcup_{v \notin \text{KILL}(b)} \text{MAP}(v)$

*A bit-vector set implementation works fine for* AVAIL
$\Rightarrow$ *can use bit position as e's name*



## Global Common Subexpression Elimination

How do we handle naming?

*Would like to ensure that* $e \in$ AVAIL($b$) *has a unique name.*

1. as replacements are done
   (a) generate unique temporary name
   (b) add a copy at each DEF for that expression
2. map textual expressions into unique names
   (*hash, bv pos'ns*)
3. equivalent value numbers get same temporary name

**Strategies to be discussed**

(1) the classical method - it works
(2) limits replacement to textually identical expressions
(3) requires more analysis but yields more *cses*

*Are copies a concern? Ask the register allocator.*

## Global Common Subexpression Elimination

**Approach 1**:

generate a unique name for each replacement

In value numbering step

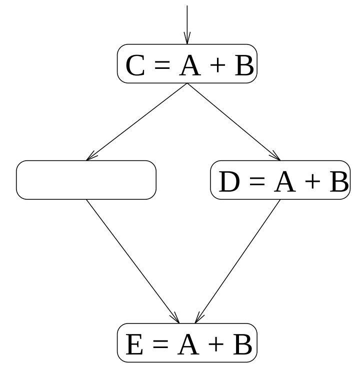
1. $\forall\ e \in$ AVAIL$(b)$, initialize AVAILTAB and mark it
2. if we use $e$ and $e \in$ AVAIL$(b)$,
   (a) allocate a name $n$,
   (b) search backwards from $b$ along each path in the *cfg* to find last DEF of $e$,
   (c) insert a copy to $n$ after DEF
   (d) replace $e$ with $n$

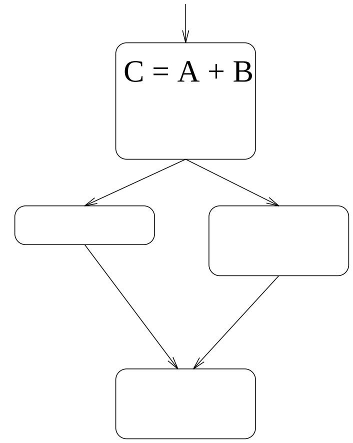
Problems:

- searching might take a fair amount of time
  - conceptually ugly notion
  - Scarborough suggests that this is not a major worry
- $|$names$| \propto |$USES$|$ $\qquad\qquad (> |$AVAIL$|)$
- single DEF followed by many copies

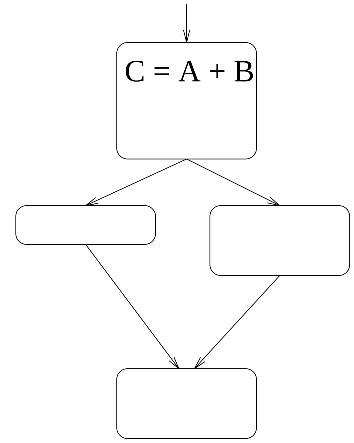


## Example




*Approach 1*




*What's the best we can do?*

## Global Common Subexpression Elimination

**Approach 2**:

textually identical expressions get the same name

Before any value numbering

1. initialize an array USED to *false*
   ($|\text{USED}| = |\text{AVAIL}|$)
2. $\forall\ e \in \text{AVAIL}(b)$, initialize AVAILTAB and mark it
3. if we use $e$ and $e \in \text{AVAIL}(b)$
   (a) if $e$ is unused (*i.e.*, it has not been assigned a *name*) allocate one, else use assigned *name*
   (b) set USED [*name*] to *true*
   (c) replace $e$ with *name*

After all value numbering

4. $\forall$ block $b$, if $e \in \text{DEF}(b)$ and $\text{USED}[e]$

   insert a copy to *name* after the DEF of $e$

Problems

- may insert extra copies
- adds one more pass over the code



## Global Common Subexpression Elimination

**Approach 3**:

textually identical expressions get the same name

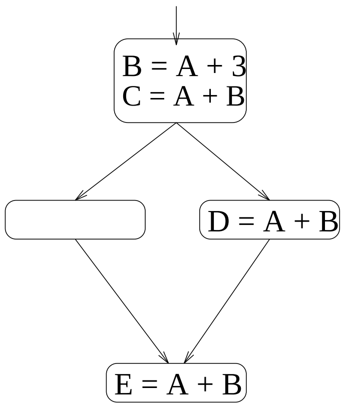
In value numbering step

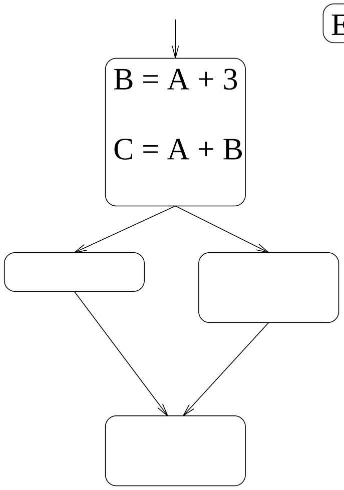
1. $\forall\ e \in \text{AVAIL}(b)$, initialize AVAILTAB
2. at an evaluation of $e$
   - insert a copy of $e$ to *name* (*hash 'em*)
3. at a use of $e$, if $e \in \text{AVAIL}(b)$
   - replace it with a reference to *name*

Problems:

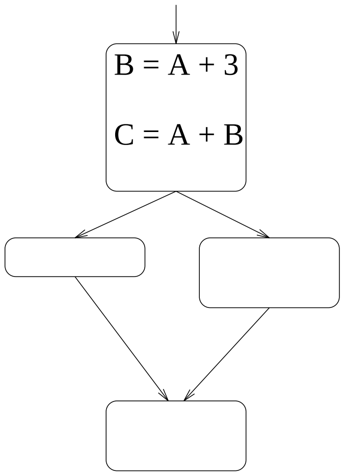
- inserts more extraneous copies than approach 2
- extra copies are *dead*

## Example




Approach 2




Approach 3



## Global Common Subexpression Elimination

What about all those extra copies?

- dead code elimination
- subsumption *or* coalescing

⇒ *rely on other optimizations to catch them!*

Common strategy (PL.8 compiler)

- insert copies that might be useful
- let dead code eliminator discover the useless ones
- simplifies compiler (*for a price*)

Dead code elimination

- must be able to recognize global usefulness
- must be able to eliminate useless stores
- must have strong notion of "dead"

## Global Common Subexpression Elimination

The iterative algorithm computes a *maximum fixed point* MFP solution to the set of equations. This need not be same as the MOP.

- if $\mathcal{F}$ is distributive, MOP = MFP
- if $\mathcal{F}$ is *not* distributive, MOP $\geq$ MFP

Is AVAIL$(b)$ distributive?

Is AVAIL$(b)$ *rapid*, *i.e.* does AVAIL$(b)$ converge after two iterations around a loop?



## Next Time

**Scheduling and Register in a single basic block**

Read: Proebsting & Fischer, "Linear-time, Optimal Code Scheduling for Delayed-Load Architectures," *ACM Conference on Programming Language Design and Implementation*, pp. 256-267, Toronto, Canada, June 1991.