Local Optimizations - Part 2

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

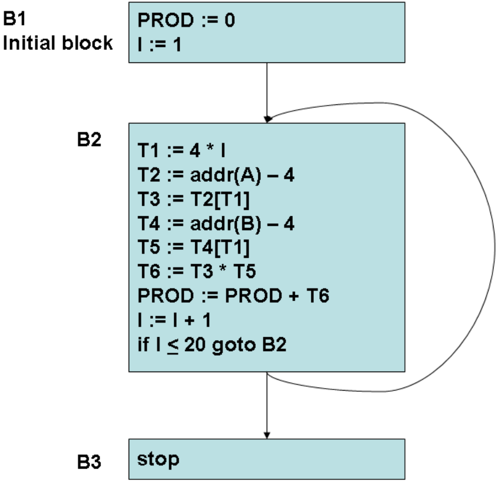
NPTEL Course on Compiler Design

Outline of the Lecture

1. Basic blocks and control flow graphs
2. Local optimizations
3. Building a control flow graph
4. Directed acyclic graphs and value numbering

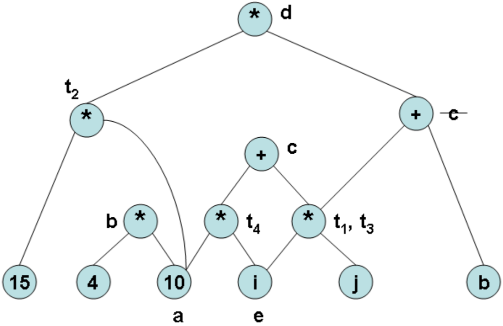
Topics 1,2, and 3 were covered in part 1 of the lecture.

Example of a Control Flow Graph



Example of a Directed Acyclic Graph (DAG)

1. $a = 10$
2. $b = 4 * a$
3. $t1 = i * j$
4. $c = t1 + b$
5. $t2 = 15 * a$
6. $d = t2 * c$
7. $e = i$
8. $t3 = e * j$
9. $t4 = i * a$
10. $c = t3 + t4$



Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*
- While searching DAGs represented using pointers etc., is inefficient, *value-numbering* uses hash tables and hence is very efficient
- Central idea is to assign numbers (called value numbers) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs
- We assume quadruples with binary or unary operators
- The algorithm uses three tables indexed by appropriate hash values:
  HashTable, ValnumTable, and *NameTable*
- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one

Data Structures for Value Numbering

In the field *Namelist*, first name is the defining occurrence and replaces all other names with the same value number with itself (or its constant value)

HashTable entry
(indexed by expression hash value)

Expression	Value number

ValnumTable entry
(indexed by name hash value)

Name	Value number

NameTable entry
(indexed by value number)

Name list	Constant value	Constflag

Example of Value Numbering

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$	1. $a = 10$	1. $a = 10$
$b = 4 * a$	2. $b = 4 * a$	2. $b = 40$
$c = i * j + b$	3. $t1 = i * j$	3. $t1 = i * j$
$d = 15 * a * c$	4. $c = t1 + b$	4. $c = t1 + 40$
$e = i$	5. $t2 = 15 * a$	5. $t2 = 150$
$c = e * j + i * a$	6. $d = t2 * c$	6. $d = 150 * c$
	7. $e = i$	7. $e = i$
	8. $t3 = e * j$	8. $t3 = i * j$
	9. $t4 = i * a$	9. $t4 = i * 10$
	10. $c = t3 + t4$	10. $c = t1 + t4$
		(Instructions 5 and 8 can be deleted)

Running the algorithm through the example (1)

1. $a = 10$:
   - a is entered into *ValnumTable* (with a *vn* of 1, say) and into *NameTable* (with a constant value of 10)
2. $b = 4 * a$:
   - a is found in *ValnumTable*, its constant value of 10 in *NameTable*
     - We have performed *constant propagation*
     - $4 * a$ is evaluated to 40, and the quad is rewritten
     - We have now performed *constant folding*
     - b is entered into *ValnumTable* (with a *vn* of 2) and into *NameTable* (with a constant value of 40)
3. $t1 = i * j$:
   - i and j are entered into the two tables with new *vn* (as above), but with no constant value
   - $i * j$ is entered into *HashTable* with a new *vn*
   - $t1$ is entered into *ValnumTable* with the same *vn*

Running the algorithm through the example (2)

4. Similar actions continue till $e = i$
   - e gets the same *vn* as i
5. $t3 = e * j$:
   - e and i have the same *vn*
   - hence, $e + j$ is detected to be the same as $i + j$
   - since $i + j$ is already in the HashTable, we have found a *common subexpression*
   - from now on, all uses of $t3$ can be replaced by $t1$
   - quad $t3 = e * j$ can be deleted
6. $c = t3 + t4$:
   - $t3$ and $t4$ already exist and have *vn*
   - $t3 + t4$ is entered into *HashTable* with a new *vn*
   - this is a reassignment to c
   - c gets a different *vn*, same as that of $t3 + t4$
7. Quads are renumbered after deletions

Example: *HashTable* and *ValNumTable*

HashTable

Expression	Value-Number
$i*j$	5
$t1+40$	6
$150*c$	8
$i*10$	9
$t1+t4$	11

ValNumTable

Name	Value-Number
a	1
b	2
i	3
j	4
$t1$	5
c	6,11
$t2$	7
d	8
e	3
$t3$	5
$t4$	10

Example: *NameTable*

NameTable

Name	Constant Value	Constant Flag
a	10	T
b	40	T
i, e		
j		
$t1, t3$		
$t2$	150	T
d		
c		

Handling Commutativity etc.

- When a search for an expression $i + j$ in *HashTable* fails, try for $j + i$
- If there is a quad $x = i + 0$, replace it with $x = i$
- Any quad of the type, $y = j * 1$ can be replaced with $y = j$
- After the above two types of replacements, value numbers of x and y become the same as those of i and j, respectively
- Quads whose LHS variables are used later can be marked as *useful*
- All unmarked quads can be deleted at the end

Handling Array References

Consider the sequence of quads:

1. $X = A[i]$
2. $A[j] = Y$: i and j could be the same
3. $Z = A[i]$: in which case, $A[i]$ is not a common subexpression here

- The above sequence cannot be replaced by: $X = A[i]$; $A[j] = Y$; $Z = X$
- When $A[j] = Y$ is processed during value numbering, ALL references to array A so far are searched in the tables and are marked KILLED - this kills quad 1 above
- When processing $Z = A[i]$, killed quads not used for CSE
- Fresh table entries are made for $Z = A[i]$
- However, if we know apriori that $i \neq j$, then $A[i]$ can be used for CSE

Handling Pointer References

Consider the sequence of quads:

1. $X = *p$
2. $*q = Y$: p and q could be pointing to the same object
3. $Z = *p$: in which case, $*p$ is not a common subexpression here

- The above sequence cannot be replaced by: $X = *p; \ *q = Y; \ Z = X$
- Suppose no pointer analysis has been carried out
  - p and q can point to *any* object in the basic block
  - hence, When $*q = Y$ is processed during value numbering, ALL table entries created so far are marked KILLED - this kills quad 1 above as well
  - When processing $Z = *p$, killed quads not used for CSE
  - Fresh table entries are made for $Z = *p$

Handling Pointer References and Procedure Calls

- However, if we know apriori which objects *p* and *q* point to, then table entries corresponding to only those objects need to killed
- Procedure calls are similar
- With no dataflow analysis, we need to assume that a procedure call can modify any object in the basic block
- Hence, while processing a procedure call, ALL table entries created so far are marked KILLED
- Sometimes, this problem is avoided by making a procedure call a separate basic block

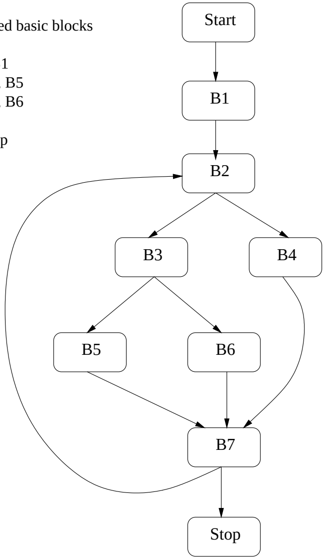
Extended Basic Blocks

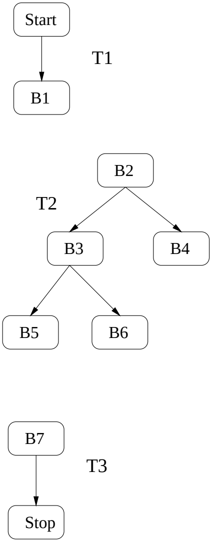
- A sequence of basic blocks $B_1, B_2, ..., B_k$, such that B_i is the unique predecessor of $B_{i+1} (i \leq i < k)$, and B_1 is either the start block or has no unique predecessor
- Extended basic blocks with shared blocks can be represented as a tree
- Shared blocks in extended basic blocks require scoped versions of tables
- The new entries must be purged and changed entries must be replaced by old entries
- Preorder traversal of extended basic block trees is used

Extended Basic Blocks and their Trees

Extended basic blocks

Start, B1
B2, B3, B5
B2, B3, B6
B2, B4
B7, Stop





Value Numbering with Extended Basic Blocks

```
function visit-ebb-tree(e) // e is a node in the tree
begin
  // From now on, the new names will be entered with a new scope into the tables.
  // When searching the tables, we always search beginning with the current scope
  // and move to enclosing scopes. This is similar to the processing involved with
  // symbol tables for lexically scoped languages
  value-number(e.B);
  // Process the block e.B using the basic block version of the algorithm
  if (e.left ≠ null) then visit-ebb-tree(e.left);
  if (e.right ≠ null) then visit-ebb-tree(e.right);
  remove entries for the new scope from all the tables
  and undo the changes in the tables of enclosing scopes;
end

begin // main calling loop
  for each tree t do visit-ebb-tree(t);
  // t is a tree representing an extended basic block
end
```