Local Optimizations

Contents

- What is code optimization and why is it needed?
- Types of optimizations
- Basic blocks and control flow graphs
- Local optimizations

Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Using variables instead of constants
  - Extra copies of variables,
  - Repeated evaluation of expressions, etc.

- Code optimization removes such inefficiencies and improves code (**time**, **space**,)

- Structure of the program may be changed (sometimes of beyond recognition)
  - eliminates some programmer-defined variables, unrolls loops, etc.
- Optimizations may be classified as local and global

Local and Global Optimizations

- **Local optimizations**: within basic blocks
  - Local common subexpression elimination
  - Dead code (instructions that compute a value that is never used) elimination
  - Reordering computations using algebraic laws

- **Global optimizations**: on whole procedures/programs
  - Global common sub-expression elimination
  - Constant propagation and constant folding
  - Loop invariant code motion
  - Partial redundancy elimination
  - Loop unrolling and function inlining

Basic Blocks and Control-Flow Graphs

- Basic blocks are sequences of intermediate code with a **single entry** and a **single exit**

- Control flow graphs show control flow among basic blocks

- Basic blocks are represented as **DAGs**, which are in turn represented using the **value-numbering method** applied on ***quadruples***

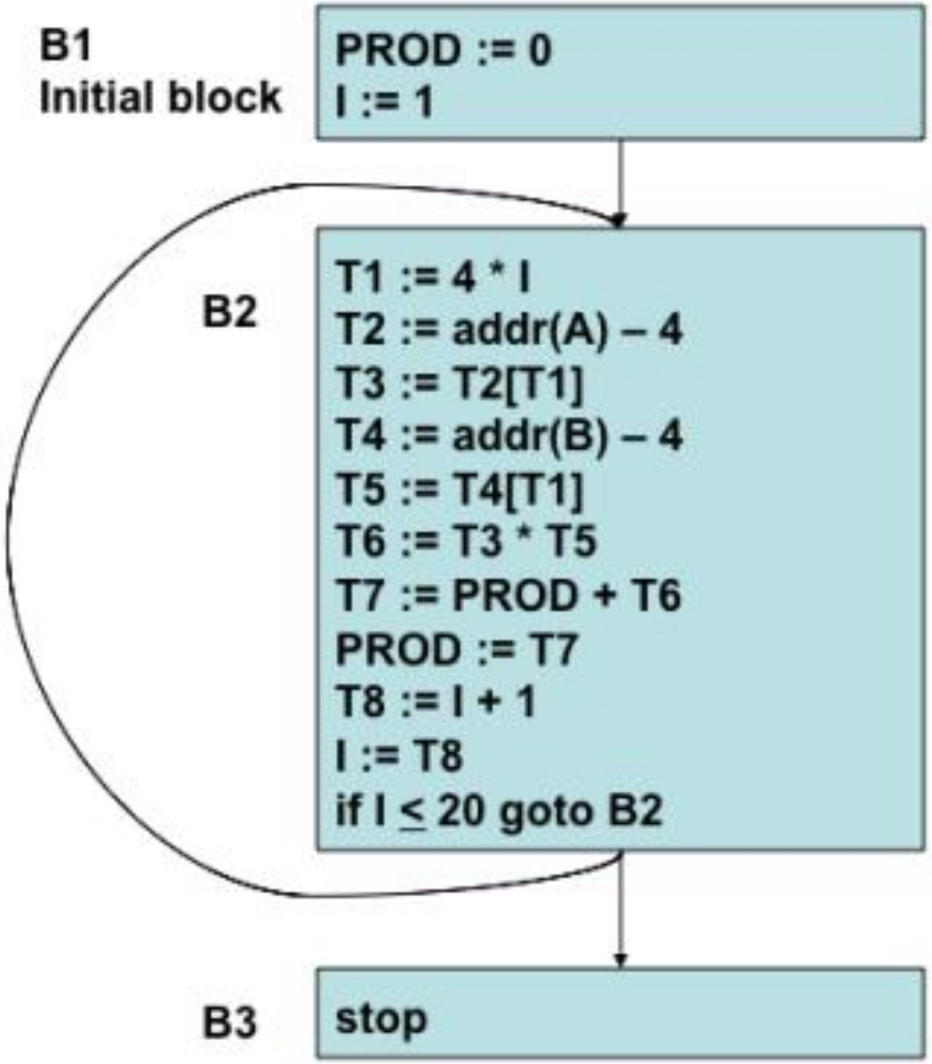
- Optimizations on basic blocks

Example of Basic Blocks and Control Flow Graph

High level language code:

```
{ PROD = 0;
  for ( I = 1; I <= 20; I++)
    PROD = PROD + A[I] * B[I];
}
```

```
PROD := 0
I := 1
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
stop
```



Partitioning into Basic Blocks

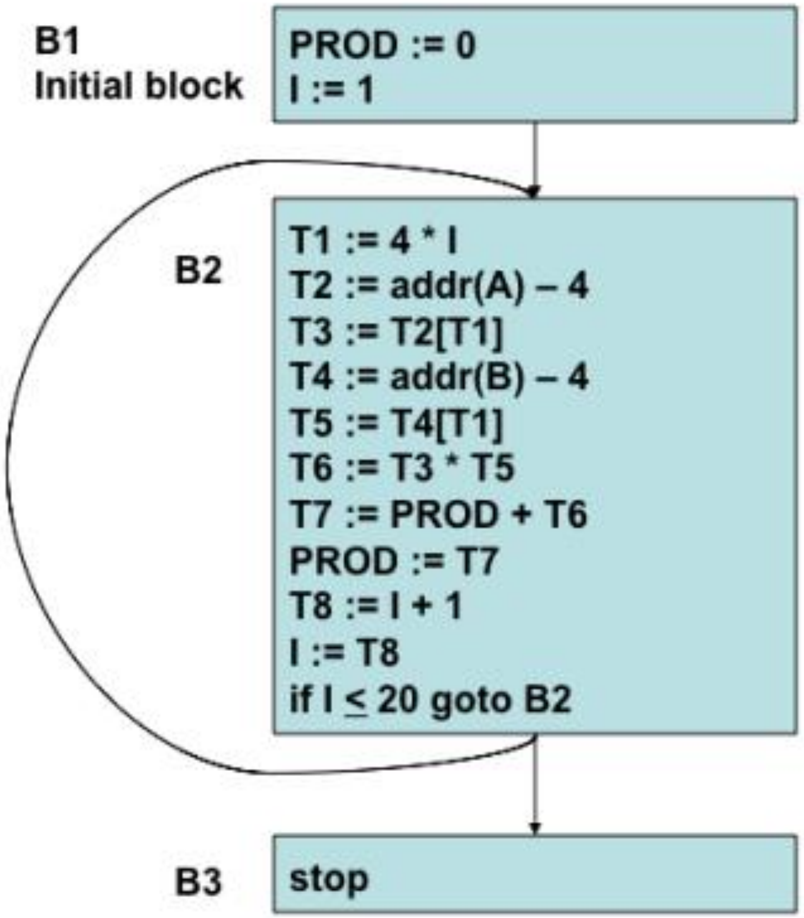
- Determine the set of **leaders** (*first statements of basic blocks)*
  - The **first statement** is a leader
  - Any statement which is the **target of a conditional or unconditional goto** is a leader
  - Any **statement which immediately follows a conditional goto** is a leader
- **Basic Block**: A leader and all statements which follow it upto (but not including) the next leader (or the end of the procedure), is the basic block corresponding to that leader
- Any statements, not placed in a block, can never be executed, and can be removed

Example of Basic Blocks and Control Flow Graph

High level language code:

```
{ PROD = 0;
  for ( I = 1; I <= 20; I++)
    PROD = PROD + A[I] * B[I];
}
```

```
PROD := 0
I := 1
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
stop
```



If goto is unconditional?

Control Flow Graph

- The nodes of the **CFG** are basic blocks
- One node is distinguished as the initial node
- There is a directed edge **B1 –→ B2**, if **B2** can immediately follow **B1** in some execution sequence; i.e.,
  - There is a *conditional* or *unconditional* jump from the last statement of **B1** to the first statement of **B2**, or
  - **B2** immediately follows **B1** in the order of the program, and **B1** does not end in an unconditional jump

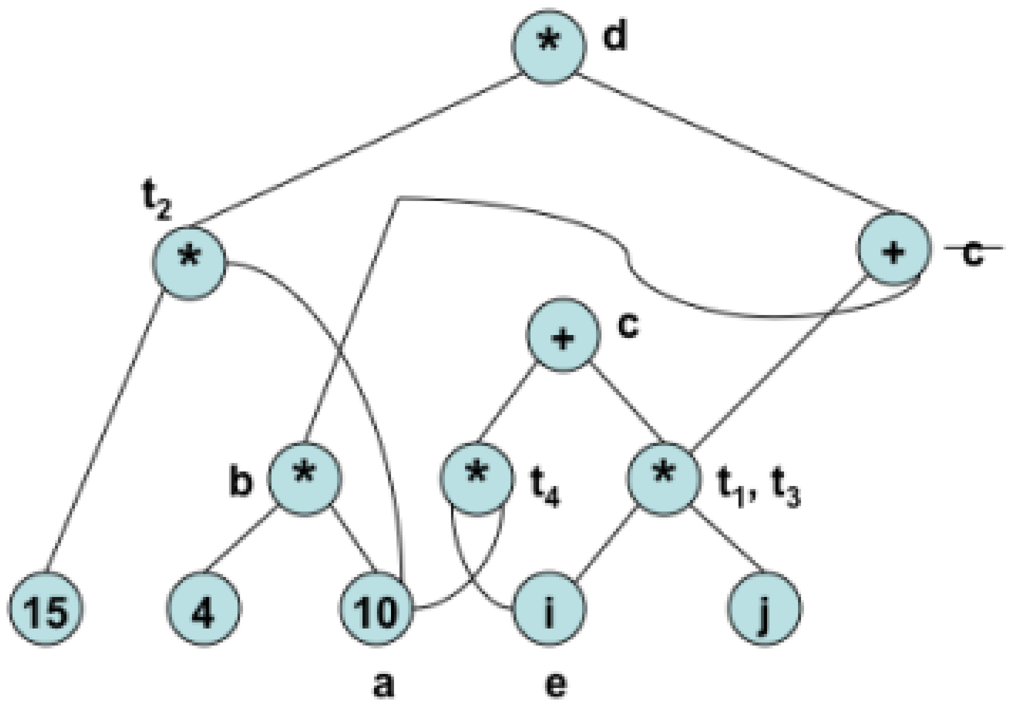
- A basic block is represented as a record consisting of
  - a count of the number of quadruples in the block
  - a pointer to the leader of the block
  - pointers to the predecessors of the block
  - pointers to the successors of the block

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. $c = t3 + t4$

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. $c = t3 + t4$



DAG representation of a basic block:

- Common sub-expression elimination
- Constant propagation and constant folding

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:*** assign numbers (called **value numbers**) to expressions
    - **two expressions receive the same number if they are equal for all possible program inputs**

- 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

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 is 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* as $i * j$

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

Quadruples before Value-Numbering
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. $c = t3 + t4$

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

HashTable

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

Example: HashTable and ValNumTable

Quadruples before Value-Numbering
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. $c = t3 + t4$

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

HashTable

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

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**
- **Procedure calls**

Extended Basic Blocks

- A sequence of basic blocks $\mathbf{B_1, B_2, \ldots, B_k}$, such that $\mathbf{B_i}$ is the unique predecessor of $\mathbf{B_{i+1}}(1 \leq i < k)$, and $\mathbf{B_1}$ 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 to perform value numbering

Extended Basic Blocks and their Trees

Extended basic blocks

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