Code optimization

Introduction

- For each source program there are many possible object programs that implement the same computation
  - (same input->same output).
- Some may be better than others in terms of speed and size
- **Code optimization -** refers to the techniques a compiler can employ in an attempt to produce a better object language program for a given source program
- The quality of the object program is measured by its running time and size

- The optimizing compiler
  - makes only well judged attempts to improve the code it produces
  - without too much time at compilation
  - i.e running time we expect to save > the time spent by the compiler doing the optimization
- Trend
  - make several compilers for programming languages
  - or options within one compiler
  - (spends varying amount of time in improving code)

Criteria for optimizing

- Does the optimization capture most of the potential improvement without an unreasonable amount of effort
- Does the optimization preserve the meaning of the source program
- Does the optimization at least on the average reduce the time or space

Principal sources of optimization

- Code optimization is done after syntax analysis and before or during code generation
- Analyze the source program code and detect certain patterns that can be replaced by more efficient but equivalent ones
- The patterns
  - may be local or global
  - may be m/c dependent| m/c independent
- Intertwined with code optimization is code generation

Some sources of optimization

- Efficient use of registers and instruction sets
- Loop Optimization
- Procedure call optimization
- Array indexing optimization
- Identification of common sub-expressions
- Constant folding i.e replacing a name by a value if value is constant.

- Inner loops
  - 90-10 rule
  - The inner loops are obvious target for optimization
- Constant folding
  - Substitution of values for names whose values are constant
- Subexpression
  - E.g., A[i+1]=B[i+1] and A[i,j]=A[i,j]+1

Algorithm optimization

- The most important source of improvement in the running program often lies beyond the reach of the compiler
  - The algorithm that is use within the source program

Loop Optimization

- Consider the example where a dot product of two vectors A and B of length 20 is computed

```
prod = 0;
i=1;
do
{
        prod=prod+A[i]*B[i];
        i=i+1;
} while (i<=20);
```

Three address code computing dot product

(1) Prod=0
(2) i=1
(3) T1=4*I
(4) T2=addr(A)-4
(5) T3=T2[T1]
(6) T4=addr(B)-4
(7) T5=T4[T1]
(8) T6=T3*T5
(9) Prod=Prod+T6
(10) i=i+1
(11) If i<=20 goto (3)

Basic Blocks

- A basic block is a sequence of three-address statements that can be entered only at the beginning, and control ends after the execution of the last statement, without a halt or any possibility of branching, except at the end

Algorithm for partitioning a sequence of three-address statements into basic blocks

- **Algorithm**
  - *Input :* A sequence of three-address statements
  - *Output:* A list of basic blocks with *each* three-address statement in exactly one block

1. Determine the set of *leaders*
   i. The first statement is a leader statement.
   ii. Any statement which is the target of a conditional or unconditional goto is a leader.
   iii. Any statement that immediately follows a conditional goto is a leader
2. Identify the leader statements in the three-address code and then include all the statements, starting from a leader, and up to, but not including, the next leader.

Using example

- Statement (1) is a leader, being the first statement.
- Statement (3) is a leader, being the target of a goto.
- Statement following (11) is a leader, being the next statement after a goto.

Basic Block 1	Prod=0
	i=1
Basic Block 2	T1=4*I
	T2=addr(A)-4
	T3=T2[T1]
	T4=addr(B)-4
	T5=T4[T1]
	T6=T3*T5
	Prod=Prod+T6
	i=i+1
	If i<=20 goto (3)

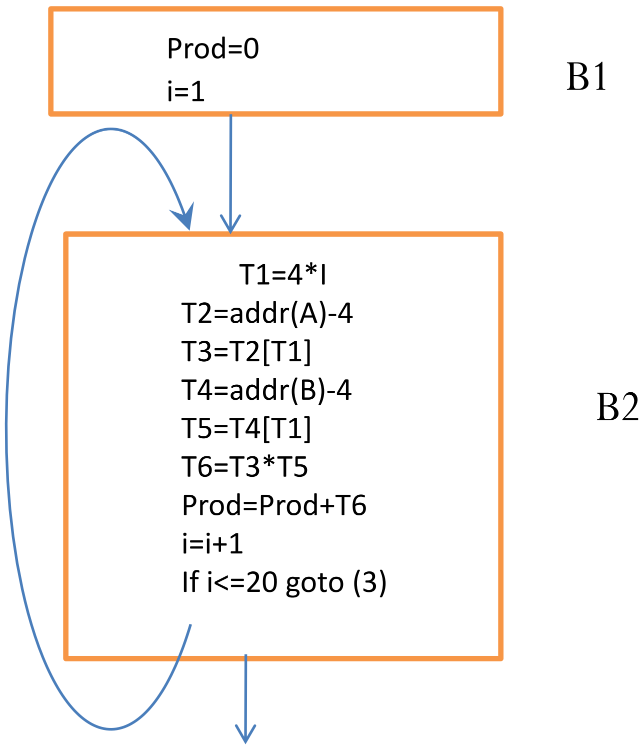
Flow Graphs

- Portraying the basic blocks and their successor relationship
- It is a directed graph
- The **nodes** are the basic block
- One node is distinguished as the **initial** it is the block whose leader is the first statement
- shows how the control is flowing in the program and how the control is being used.
- To obtain this graph, we must partition the intermediate code into basic blocks.

- Each node is a basic block.
- One node is the ***initial***; the block with leader as the first statement.
- B1 is a ***predecessor*** of B2 and B2 is a ***successor*** of B1.
- For adding edges to the graph
  - if B1 and B2 are the two blocks, then add an edge from B1 to B2 in the program flow graph, if the block B2 follows B1 in an execution sequence.
  - The block B2 follows B1 in an execution sequence if and only if:
    - The first statement of block B2 immediately follows the last statement of block B1 in the three-address code, and the last statement of block B1 is not an unconditional goto statement.
    - The last statement of block B1 is either a conditional or unconditional goto statement, and the first statement of block B2 is the target of the last statement of block B1.

Flow graph

B1 is a predecessor of B2 and B2 is a successor of B1



To block beginning with statement following (11)

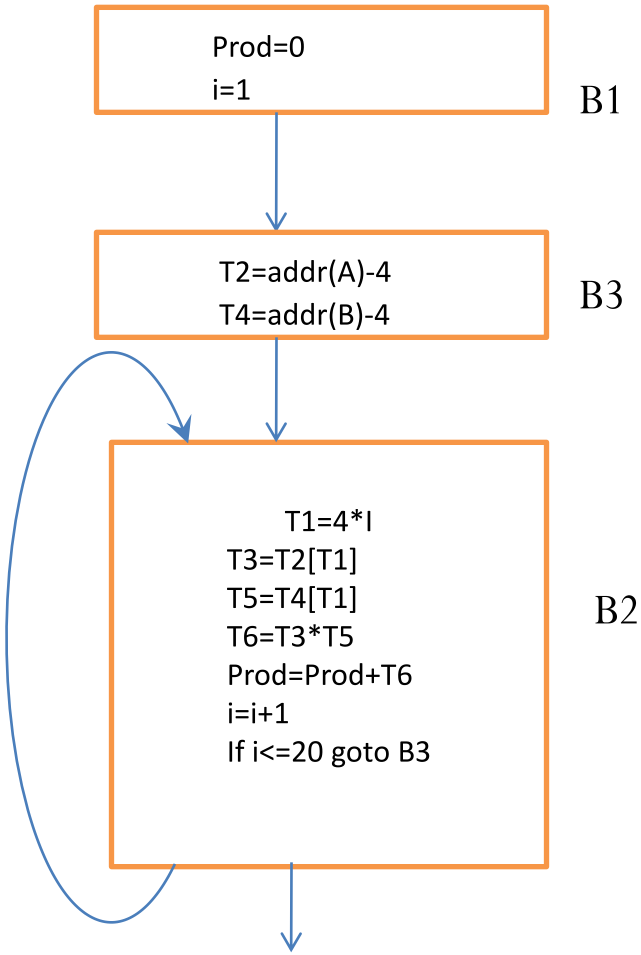
Loops

- A loop is a cycle in the flow graph that satisfies two properties:
  - Strongly connected
    - From any node in the loop to any other, there is a path of length one or more, wholly within the loop
  - A unique entry
    - A node in the loop should have a single entry to reach a node of the loop form a node outside the loop.

Code motion

- The running time of a program can be improved by decreasing the length of one of its loops, especially an inner loop
- Based on the assumption that the loop is executed at least once.
- **Code motion** takes a computation that yields the same result independent of the no. of times the loop executes (loop invariant) and places it before the loop.

Code motion example



- Statements T2=addr(A)-4 and T4=addr(B)-4 will not change throughout the loop as the arrays are static.
  - Loop-invariant computation
- Remove these statements from the loop and add it to a new block B3.
- Add edges where ever necessary.
- Blocks B1 and B3 can also be combined.

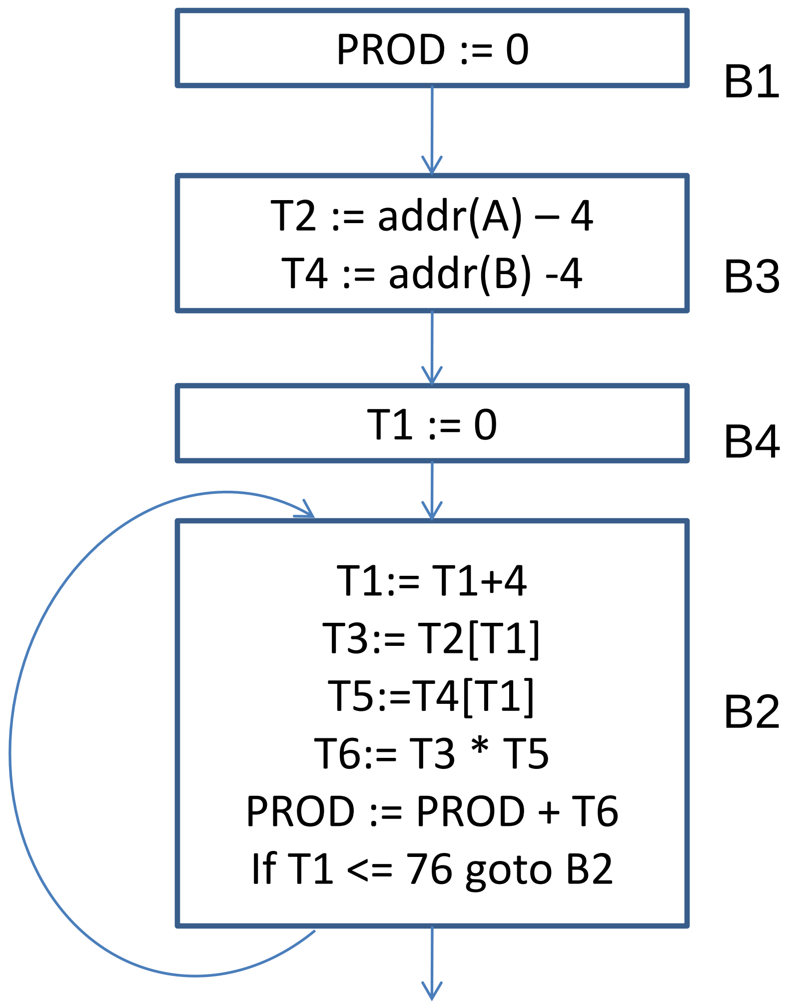
Induction Variables

- Another optimization which may be applied to the flow graph which can decrease the total number of instructions and increase the speed.
- **Induction variables** are variables that form arithmetic progression in lock step.
- Example T1 and I

Eliminating Induction Variables

- When there are two or more induction variables in a loop, we can remove all but one.
- In the example, I is a basic induction variable and T1 is an induction variable dependent on I.
- Assuming that I is not needed outside the loop, we can eliminate I in the following manner -
  - Replace I in blocks B1, B2 and B3 by T1
  - Replace I<=20 by T1<=76, since the values of T1 follows an arithmetic progression with a difference of 4.
  - Replace the statement T1=4*I by T1=T1+4
  - Since T1 does not have an initial value , so the statement T1=0 can be placed outside block B2.

Flow Graph after elimination induction variable I



Reduction in strength

- Replacement of an expensive operation by a cheaper one.
- Example
  - Replacing T1= 4 * I by T1 = T1 + 4
    - Multiplication is more expensive than addition.
  - Replacing L = Length(S1|| S2) by L = Length(S1) + Length(S2)

Directed Acyclic Graphs…

- A useful data structure to analyze basic blocks.
- A DAG is a directed graph with no cycles.
- It gives an idea of how the value computed by each statement in a basic block is used in subsequent statements in the block
- Constructing a DAG from three address statements provides the following advantage
  - Determine common sub-expressions within a block
  - Determine which names are used inside the block but evaluated outside the block
  - Determine which statements of the block have their value used outside the block.

Directed Acyclic Graphs cont…

- A DAG has the following labels on nodes
  - **Leaves** are labeled by unique identifiers either *variable names* or *constants* like addr(A) to denote l-value while others denote r-value.
  - The leaves represent *initial* values of names and are subscripted 0 to differentiate them with labels denoting *current* values
  - Interior nodes are labeled by an operator symbol
  - **Nodes** are optionally given an extra *set of identifiers* for labels
    - **Interior nodes** represent computed values and identifiers labeling a node are deemed to have that value

Example

Basic Block 2

```
T1=4*I
T2=addr(A)-4
T3=T2[T1]
T4=addr(B)-4
T5=T4[T1]
T6=T3*T5
Prod=Prod+T6
i=i+1
If i<=20 goto (3)
```

Example

(1) S1=4*I

(2) S2=addr(A)-4

(3) S3=S2[S1]

(4) S4=4*I

(5) S5=addr(B)-4

(6) S6=S5[S4]

(7) S7=S3*S6

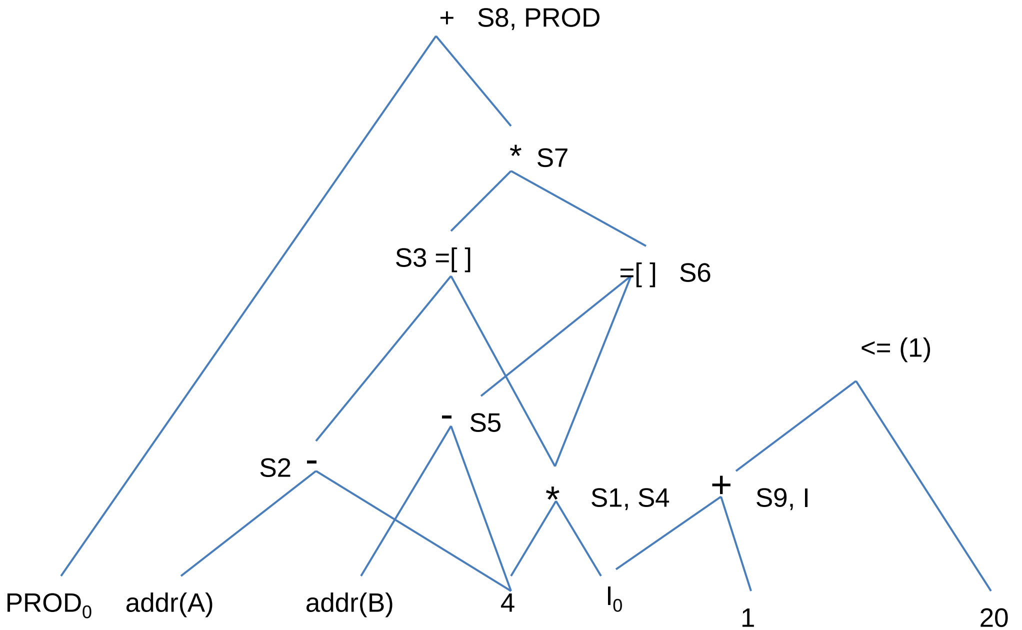
(8) S8=PROD+S7

(9) Prod=S8

(10) S9=I+1

(11) I=S9

(12) If I<=20 goto (1)



DAG construction

- A:=B+C
- Look for the nodes that represent the "current" values of B and C
- Could be leaves or interior nodes (evaluated by previous statements of the block)
  - *Create* a node labeled + and give it two children and then again label this node A
  - However if there is a node already denoting the same B+C, we *do not create* a node but give it an additional label A

DAG construction cont…

- If A(not A0) had previously labeled some node, remove it since the current value of A is the node just created
- A:=B
  - We do not create a new node but append the label A to the “current” value of B

DAG construction cont…

- Don’t do the following
  - Assignment to arrays
  - Indirect assignment through pointers
  - One location having two or more names

Algorithm

- **Input**: Basic Block
- **Output**: A DAG with the following information
  - A label for each node.
    - For **leaves** the label is an identifier/constants
    - For **interior nodes** an operator symbol
  - For each node a list of attached identifiers (constants not permitted here)

Algorithm cont…

- **Method**:
  - **Assumption:**
  - data structures to create nodes with two children i.e *left* and *right*
  - A place for a *label* for each node and the facility to create a *linked list* of attached identifiers for each node
  - **Maintenance**
  - A set of identifiers including constants for which there is a node associated either by a leaf / interior node

Algorithm cont…

- For constructing a basic block DAG, we make use of the function **node(id)**, which returns the most recently created node associated with id.
  - Intuitively **node(id)** is the node of the DAG which represents the value which **id** has at the current point
  - An entry in the symbol table record for **id** is the value of **node(id)**

Algorithm cont…

- Do Steps (1) through (3) for each statement of the block, in turn
- Initially assume there are no nodes
- **node( )** is undefined for all arguments
- Suppose the "current" three address statements is either
  1. A:=B **op** C
  2. A:=**op** B
  3. A:=B
- Refer to these as cases i, ii, iii
- Treat **if** I<=20 **goto** as case i with A undefined

Algorithm for DAG construction

Step 1 :

1. If node(B) is undefined, create a leaf labeled B, and let node(B) be this node.
2. In case i, if node(C) is undefined, create a leaf labeled C, and let that leaf be node(C).

Step 2 :

1. In case i, determine if there is a node labeled **op** whose left child is node(B) and whose right child is node(C) (to catch the common subexpressions). If not, create such a node. In either event let *n* be the node *found* or *created*.
2. In case ii ,determine if a node exists that is labeled **op** whose only child is node(B). If not create such a node and let *n* be the node *found* or *created*
3. In case iii let *n* be the node(B)

Step 3

1. Append A to the list of identifiers for the node *n* found in step 2.
2. Delete A from the list of attached identifiers for node(A), and set node(A) to *n*.

Example

(1) S1=4*I

(2) S2=addr(A)-4

(3) S3=S2[S1]

(4) S4=4*I

(5) S5=addr(B)-4

(6) S6=S5[S4]

(7) S7=S3*S6

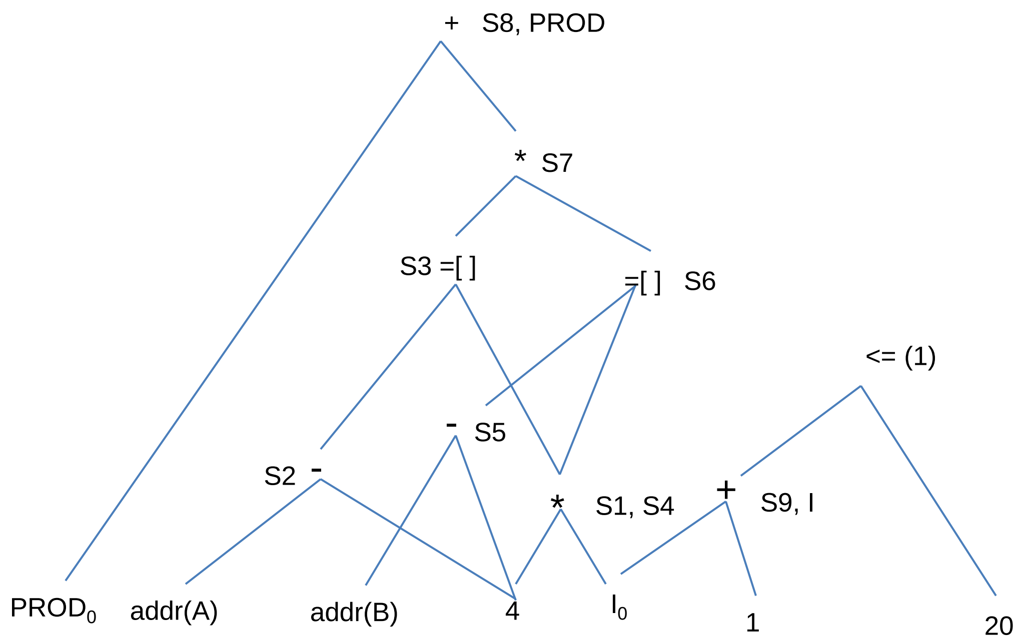
(8) S8=PROD+S7

(9) Prod=S8

(10) S9=I+1

(11) I=S9

(12) If I<=20 goto (1)



Applications of DAG

- Automatically detects common sub-expressions.
- Determine which identifiers have their values used in the block;
  - The ones for which a leaf node is created in step 1
- Determine which statements creates values which can be used outside the block;
  - The statements S whose node n constructed or found in step 2 still has NODE(A)=n at the end of the DAG construction, where A is the identifier assigned by statement S.
  - In the e.g all interior nodes can have their values used outside the block.