Intermediate Code Optimization

Syntax directed translation scheme combined with run-time enviornment lead to generation of intermediate code that is semantically equivalent to the original source program. However, as we have seen in the earlier analysis, the quality of intermediate code is not very satisfactory.

The intermediate code optimization phase of a compiler takes the intermediate code produced by the front end as an input and performs several anlayses over the intermediate code to improve the quality of code for more efficient execution at run time. Two distinct analyses, known as control flow analysis and data flow analysis form the key component of this phase. In this module, both the analysis are discussed and illustrated with suitable examples. For more details, text books and other references should be read.

The motivation of intermediate code optimization (also referred to as code optimization to improve the quality of intermediate code because this in turn leads to generation of high quality target code. The constraints for the optimizations are, i) semantic equivalence with the source, and ii) the algorithms in the source remain unchanged.

1. Introduction

The motivation of intermediate code optimization (also referred to as code optimization) is to improve the quality of intermediate code. This in turn leads to generation of better quality target code. The constraints for the optimizations are, i) semantic equivalence with the source, and ii) the algorithms in the source remain unchanged. There are two diffferent levels at which optimizations are performed ina compiler.

- Machine dependent optimization : such optimizations are performed for a better choice of target machine instructions, or addressing modes and machime registers. These optimizations, if performed, are a part of code generation or a seprarte pass post code generation and are not considered in this module.
- Machine independent optimization : These optimizations are performed on the intermediate code before the generation of machine code. All the optimizations discussed here belong to this category.

The machine independent optimizations are applied at two different scopes.

In local optimization the scope is restricted o straight line parts of intermediate code called as basic blocks. Code in a basic block is sequentially executed and hence these are relatively simpler to analyse and implement.

Global optimization has a wider scope ranging from a single procedure or a function to that of the entire program. In this course, we shall restrict the scope of global optimization to that of a single procedure, usually called as intraprocedural global data flow analysis. Interprocedural data flow analysis opens up more opportunities for optimization but is outside the purview of this course.

2. Construction of Basic Blocks

Algorithm : Partition intermediate code into basic blocks and construct a control flow graph

i/p: A sequence of three-address statements after semantic analysis

o/p : Control flow graph

Method outline :

1. Identify the set of headers, which are the first statements of basic blocks. Headers of basic blocks are identified as follows.

- The first statement is a header.
- Any statement that is the target of a conditional or unconditional goto is a header.
- Any statement that immediately follows a goto or conditional goto statement is a header.

2. Extract the body of each basic block. For each header, its basic block consists of the header and all statements up to but not including the next header or the end of the program.

3. Insert the edges between the basic blocks. Insert edges from the basic block to its successor blocks.

Illustrative Example :

```
void quicksort(int m, int n)
{ int i, j, v, x;
  if (n <= m) return;
  i = m-1; j = n; v = a[n]; /* v is the pivot */
  while(1) /* Move values smaller */
  {  do i = i + 1; while (a[i] < v); /* than v to the left of */
     do j = j - 1; while (a[j] > v); /* the split point (sp) */
     if (i >= j) break; /* and other values */
     x = a[i]; a[i] = a[j]; a[j] = x; /* to the right of sp */
   } /* of the split point */
   x = a[i]; a[i] = a[n]; a[n] = x; /* Move the pivot to sp */
   quicksort(m,i); quicksort(i+1,n); /* sort the partitions to */
 } /* the left of sp and to the right of sp independently */
```

3. Control Flow Analysis

The control flow graph of a program is a directed graph, represented using the notation G = (N, E, n_o), where

- N is the set of basic blocks or nodes

- E is the set of edges between the nodes, and
- n_o is the unique entry node of G which has no predecessors.

The notation, succ(i) and pred(i) denote the the set of successors and predecessors of node i.

Dominators of a node : The dominator of a node i in the cfg is a node that lies on every path from the entry node to i.

DOM (i) = {i} ∪ {∩j∈ Pred (i) (DOM(j)

Initializations : DOM(Entry) = Empty; DOM(i) = Universal set; for all i in N

4. LOCAL DATA FLOW ANALYSIS

To be completed later.

5. FOUR GLOBAL DATA FLOW PROBLEMS :

Available Expressions Problem

AVIN (i)=∩ j ∈Pred (i) {GEN (j)∪(AVIN (j)−KILL(j))}

AVOUT (i)=GEN (i)∪(AVIN (i)−KILL(i))

Initializations : AVIN(Entry) = Empty; AVIN(i) = Universal set; for all i in N

Reaching Definitions Problem

RDIN (i)=∪ j ∈Pred (i) {GEN (j)∪(RDIN (j)−KILL(j)) }

RDOUT (i)=GEN (i)∪(RDIN (i)−KILL(i))

Initializations : RDIN(i) = Empty; for all i in N

Live Variable Problem

LIVEIN (i)=GEN (i)∪{ ∪j∈ Succ (i) (LIVEIN (j)−KILL(i)) }

LIVEOUT (i)=∪ j∈ Succ (i) LIVEIN (j)

Initializations : LIVEIN(i) = Empty; for all i in N

Very Busy Variable Problem

VBIN (i)=GEN (i)∪{ ∩j∈ Succ (i) (VBIN (j)−KILL(i))}

VBOUT (i)=∩ j∈ Succ (i) VBIN (j)

Initializations : VBIN(Entry) = Empty; VBIN(i) = Universal set; for all i in N

Summary of Data Flow Problems

Problem	Direction	Generated Information	Killed Information	Merge Operation	Initialization
Available Expressions	Forward	Downwards exposed expression	a = ... kills expressions involving a	Intersection ∩	Universal Set of expressions
Reaching Definitions	Forward	Downwards exposed definition 10 : v :=	30 : v := kills all other definitions of v	Union ∪	Empty set
Live Variables	Backward	Upwards exposed reference of x	20 : x : = kills variable x	Union ∪	Empty set
Very Busy Expressions	Backward	Upwards exposed occurance of expression	a = ... kills expressions involving a	Intersection ∩	Universal Set of expressions
Dominators	Forward			Union ∪	Universal Set of basic blocks

Generic Algorithm for iterative solution of data flow problems.

Algorithm Iterative Solver

Input : Data flow equations for XIN(i) and XOUT(i)

Control flow graph G = (N, E, n_o)

Output : XIN(i) and XOUT(i)

Method :

1. Initialize XIN(i) as required for the data flow problem
2. bool change = true;

```
while (change) do
{  change = false;
   for all i in N – {boundary node} do
   {  compute newXIN(i) using the recurrence relation;
      if (newXIN(i) != XIN(i) )
      then { XIN(i) = newXIN(i); change = true; }
   } // end of for
} // end of while
```

3. For all i in N, compute XOUT(i) using the converged values of XIN(i)