

Code Optimization Using Graph Mining

Saranya Venkatachalam, N. Sairam and B. Srinivasan
School of Computing, SASTRA University, Thanjavur, India

Abstract: In compiler theory, optimization is the process of fine-tuning the output of a compiler to curtail or capitalize on some aspects of an executable computer program. Optimization is commonly implemented using a chain of optimizing transformations, algorithms which inputs a program and transform it to produce an equivalent output program that yields better performance. In this study, we propose a novel technique which involves the concept of dominators to optimize the code. This technique is employed to a graph, depicting the program flow (CFG) from which, the information is extracted and the loop invariants are relocated to the dominating program block there by reducing the redundant computation.

Keywords: Control Flow Graph (CFG), dominators, loop invariants

INTRODUCTION

In computer science, program or software optimization involves the application of rules and algorithms to program code to make it execute more rapidly with less memory storage or other resources, or draw less power. Optimization can be performed at multiple levels of program representation as listed below.

- Source code
- Intermediate code
- Target machine code

An intermediate representation is a compile time data structure with ease of generation and manipulation. Intermediate code is simple and explicit by decomposing the whole code generation problem into simpler pieces. The front end produces Intermediate Representation (IR) for the source program. The back end works with the code in IR form to produce native code for the target machine. The optimizer transforms the code in IR form into an equivalent program that may run more efficiently.

Flow analysis is the traditional optimizing compiler technique for determining useful information about a program at compile time. Essentially, there includes two form of analysis to optimize the code at a high level:

- Data flow analysis
- Control flow analysis

As the name indicates data flow analysis attempts to discover how information flows through a program.

Control flow analysis begins by constructing a Control Flow Graph (CFG). CFG models the transfer of control in the procedure. It is a triple $G = (V, A, s)$ where V, A is a (finite) directed graph with $|V| = n$ vertices and

$|A| = m$ arcs such that there is a path from initial node $s \in V$ to every node. Without loss of generality, any node unreachable from s can be deleted. An exit node of a flow graph has no successors.

Many programs spend most of their time on loops. Hence, it is worthwhile to consider optimization specific for loops. Classical loop optimization technique includes:

- Loop invariant code motion
- Induction variable recognition
- Strength reduction
- Linear test replacement
- Loop unrolling

Xu-Ze and Hong-Guang (2009) used an optimization scheme about code motion based on DAG code list. This scheme removes loop invariant out of the loop, thus saving clock cycles in the target machine. This results in improvement of the quality of the object code produced by a C compiler (16 bit embedded CPU system).

To optimize the code, Pohua *et al.* (1991) describes the intend and execution of an optimizing compiler that includes components, which incorporates probes into the input program, executes for several inputs and accumulates profile information. With this profile information, the code is optimized.

(Murat *et al.*, 2011) introduced a new approach for program optimization. The approach differs from standard way by applying transformation to subset of translation units resulting in the reduction of compilation time.

The objective of this work is to relocate the loop invariants in order to provide speed up and allowing the constants to be stored in registers thereby not necessitating in calculation of the address and access to the memory at each iteration.

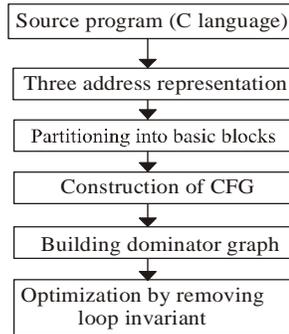


Fig. 1: Outline of the entire optimization technique

METHODOLOGY

Architecture: Figure 1 sketches the steps to be carried out for optimizing the input code. The main objective is to optimize the given C program. The input source program has to be transformed into a semantically equivalent output program by removing invariants out of the loop and placing it in such a way that redundant computations of the statements are exterminated.

Optimization technique:

Intermediate code generation: The intermediate representation makes retargeting of the source code possible and allows certain optimizations to be carried out. The commonly used intermediate representations are:

- Post-fix notation
- Syntax tree
- Three address codes

A pseudo IR called three address codes which uses virtual registers is enlisted here to optimize the code. In Three Address Code (TAC) form of representation, each statement has at most one operation in its right hand side. In general, they allow statements of the form:

$$x \leftarrow y \text{ op } z$$

with single operator and at most three names. The temporary variables are introduced to have at most three names/address locations to be referred. The representation scheme allows in establishing:

- Temporary variables (if needed)
- Controlling structures to be broken down into branch and goto statements
- Pointers and address calculations to be made explicit

For example, the equivalent three address representation of the arithmetic expression $c = a + b$ is:

```
temp = a + b
C = temp
```

where, a, b and c are variable names. Note that temp variable is expended by the compiler to keep the count of operands down to three.

Construction of CFG: To optimize the code, the program flow is required which is well represented by the Control Flow Graph (CFG). To build control flow graph, the three address representation of the source program (in C language) is obtained. The three address code is divided into basic blocks, which forms the nodes of CFG and edges of CFG correspond to the flow of control (branches or fall-through execution).

(Aho *et al.*, 1986) explains that a basic block is a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning and leave at the end. The first statement of a basic block can be target of a branch and only the last statement of a basic block can be a branch statement.

A basic block begins in one of the following ways:

- The instruction immediately following a branch or return
- The target of a branch
- The entry point into a function

A basic block ends in one of the following ways:

- A jump statement
- A conditional or unconditional branch
- A return statement

The algorithm for identifying basic blocks (Aho *et al.*, 1983) is given below:

Algorithm 1: Identifying basic blocks

Input : A sequence of three address statements

Output : A list of basic blocks

Method :

- (1) Identify leader statement by using the following rules:
 - The first statement of the input sequence is a leader.
 - The statement that is the target of a branch statement is a leader (for most intermediate languages these are statement with an associated label)
 - Any statement that immediately follows a branch or return statement is a leader.
- (2) The basic block corresponding to a leader consists of the leader, plus all the statements up to but not including the next leader or up to the end of the three address statement.

A control flow graph is built using these basic blocks as nodes and nodes are connected if there exists a block that could follow another block (i.e. nodes) in the execution of the program. If a block cannot be reached from the starting block then the block can be removed. The built graph does not contain information about the data and so the edge in the constructed graph indicates that the execution of the program takes that path. Using the guidelines in algorithm: 2 CFG is constructed.

Algorithm 2: Guidelines for constructing CFG:

Input : A list of basic blocks
 Output : A directed multi graph
 Method :

There exists a directed edge from basic block (say B1) to another block (say B2) if the following condition holds:

- (i) if the last statement of B1 contains a branching statement with target as B2
- (ii) B2 immediately follows B1
- (iii) B1 does not end with conditional branch

Computation of dominator graph: The dominator tree is a subsidiary data structure depicting the dominator relationships. A block D dominates a block V, if every path from the initial node of the CFG to V goes through D (Hiralal, 1994). In other words, vertex D dominates vertex V if $D \neq V$ and D is on every path from entry node to V. i.e., D dominates V (denoted by $D \text{ DOM } V$) in a CFG iff all paths from entry node to V includes D.

$\text{DOM}(V)$ = the set of all vertices that dominates V

- All vertices dominate themselves $V \in \text{DOM}(V)$
- Entry dominates every node in the graph:

$$\forall v \in V: \text{Entry} \in \text{DOM}(V)$$

Most global optimization use dominance information. The number of edges in the dominator graph is less than or equal to the number of edges in CFG. For this reason, dominator graph is constructed using algorithm: 3.

Algorithm 3: Pseudo-code to compute dominators

Input:

L : Adjacency list representation of CFG
 C : Count of number of basic blocks in the given C program

Output:

D : Dominator of each node of CFG

Method:

- (1) index ← 1
- (2) For each c in C do
 visit[c] ← 0
 order[c] ← 0
 end for
- (3) Enqueue the entry node (i.e., Block 1) to queue (say q) order[index] ← Block1, visit [index] = 1 increment index
- (4) while (queue is not empty)
 - (i) element = dequeue()//pops out the first element from the queue
 - (ii) if(element exists and it is not visited)
 - if \exists a single adjacent node to element
 enqueue adjacent node1 to q//add adjacent node to q
 if (visit[adjacent node 1] == 0)
 order[index] = adjacent node1
 increment index
 end if
 - end if
 - if \exists another adjacent node to element
 enqueue adjacent node2 to q//add adjacent node to q
 if (visit[adjacent node2] == 0)
 order[index] = adjacent node2
 increment index
 end if
 - end if
 - visit[element] = 1
 - end if
- (5) Visit the nodes as per the sequence in “order” list
- (6) Compute the dominators of each node using the following:
 - (i) every node dominates itself
 - (ii) all nodes are dominated by the entry node
 - (iii) If a node (say N) is reachable from multiple ways, then include all those nodes along the path that is mandatory to reach N
- (7) Return dominators of each node

Relocation of loop invariants: Most of the computation time is spent on loops. To minimize the computation time, it is necessary to identify the loop invariants (Xu-Ze and Hong-Guang, 2009) and move them to the dominating program block. An expression is invariant in a loop iff:

- It is a pure computation, all of whose arguments are loop invariants and
- It is a variable use whose single reaching definition and the RHS of the definition is loop invariant.

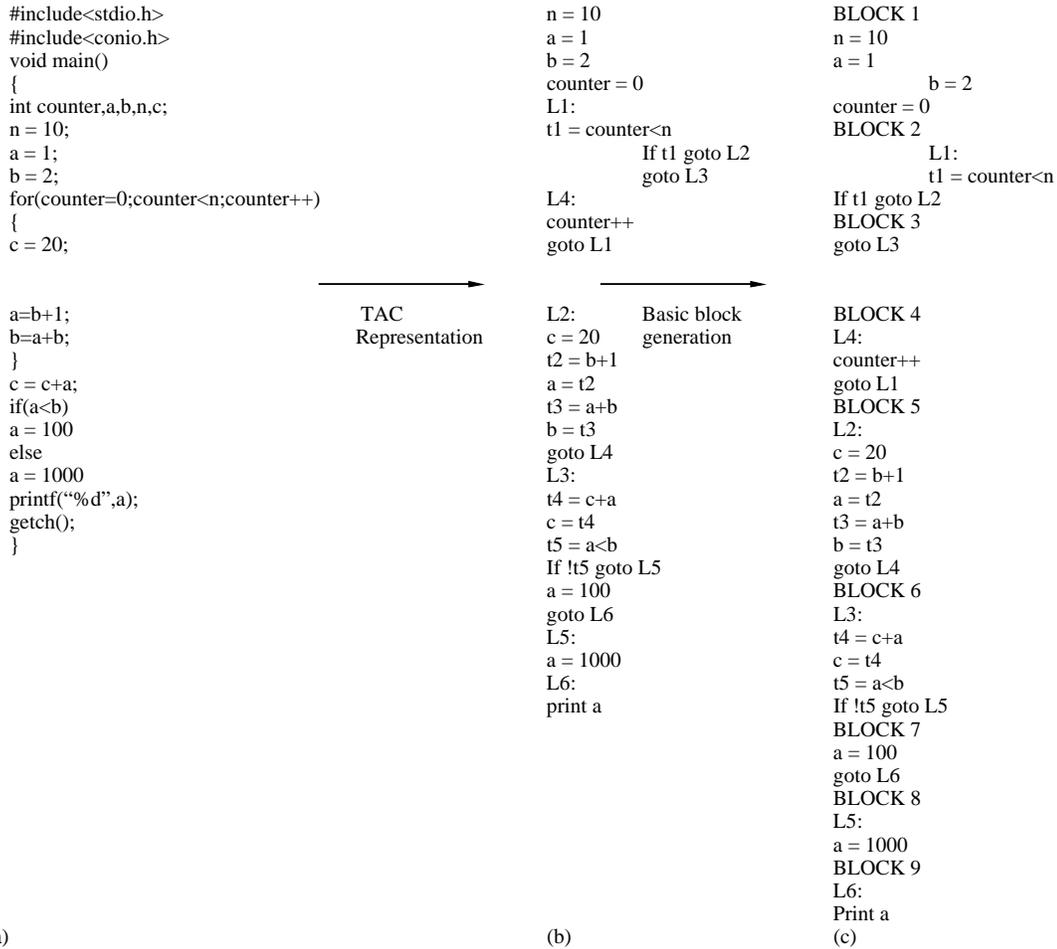


Fig. 2: a) Input C program b) TAC c) Partitioning of TAC into blocks

Table 1: Dominators of each node in CFG

Vertex v	DOM (v)
B1	{B1}
B2	{B1, B2}
B3	{B1, B2, B3}
B4	{B1, B2, B5, B4}
B5	{B1, B2, B5}
B6	{B1, B2, B3, B6}
B7	{B1, B2, B3, B6, B7}
B8	{B1, B2, B3, B6, B8}
B9	{B1, B2, B3, B6, B9}

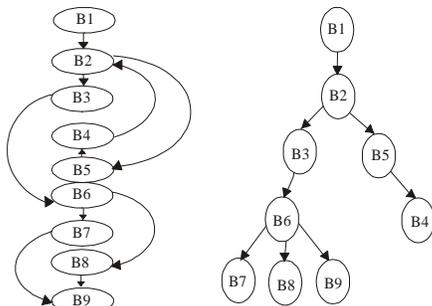


Fig. 3: CFG and its dominator graph

For example, the loop invariant statement for the input in Fig. 2 is $c = 20$. Loop invariant code which is hoisted out of a loop to a dominating program block is executed less often, thus providing a speed up.

EXPERIMENTAL RESULTS

Figure 2a shows the C program that is to be optimized. The equivalent three address code representation of the given program is obtained for the purpose of optimizing the input code at intermediate level. Following the guidelines in algorithm: 2 CFG is constructed. With pseudo code given in algorithm: 3, the dominating node/nodes of each vertex of CFG is computed and tabulated (Table 1). The dominator graph thus built can be used to relocate the loop invariant to the corresponding block. It is observed experimentally that the number of edges in the dominator graph is always less than or equal to the number of edges in CFG (Fig. 3).

The work was analyzed by taking several input C programs that is rich in loop invariant statements. The

proposed technique can be applied in any environment which demands code optimization. The C program that is taken for evaluation is based on the following specification:

- Assignment statements-allowed
- Unary + and-allowed
- Pre and post increment-allowed
- Arithmetic instructions with binary operands only
- Control structures:
 - if
 - if-else
 - multiple if
 - for
 - while
 - do while

CONCLUSION

The proposed optimization technique can be employed to any C program to retarget the loop invariants. The information for this retargeting of loop invariants is extracted from the dominator graph. The construction of dominator graph from CFG provides a refined view of the program flow. This process of retargeting, results in better performance by eliminating

redundant computation of the statements. The future enhancement is to identify the infeasible path within the program by making use of the obtained dominator graph.

REFERENCES

- Aho, A.V., J.E. Hopcroft and J.D. Ullman, 1983. Data Structures and Algorithms, Addison-Wesley.
- Aho, A.V., R. Sethi and J.D. Ullman, 1986. Compilers: Principles, Techniques and Tools Addison-Wesley.
- Hiralal, A., 1994. Dominators Super Blocks and Program Coverage, ACM.
- Murat, B., E.T. Kirk Kelsey, L. Xiaoming and R.G. Guang, 2011. Source Code Partitioning in Program Optimization IEEE 17th International Conference on Parallel and Distributed Systems, pp: 56-63.
- Pohua, P.C., A.M. Scott and W.H. Wen-Mei, 1991. Using Profile Information to Assist Classic Code Optimizations Center for Reliable and High-performance Computing, University of Illinois, Urbana-Champaign.
- Xu-Ze, L. and Z. Hong-Guang, 2009. Design and Implementation of Code Motion. Based on DAG Code List, IEEE, pp: 263-265