


3-1-1985

Code optimization with stack oriented intermediate code

Satischandra B. Reddy
Atlanta University

Follow this and additional works at: <http://digitalcommons.auctr.edu/dissertations>

 Part of the [Applied Mathematics Commons](#), [Computer Sciences Commons](#), and the [Mathematics Commons](#)

Recommended Citation

Reddy, Satischandra B., "Code optimization with stack oriented intermediate code" (1985). *ETD Collection for AUC Robert W. Woodruff Library*. Paper 2629.

This Thesis is brought to you for free and open access by DigitalCommons@Robert W. Woodruff Library, Atlanta University Center. It has been accepted for inclusion in ETD Collection for AUC Robert W. Woodruff Library by an authorized administrator of DigitalCommons@Robert W. Woodruff Library, Atlanta University Center. For more information, please contact cwiseman@auctr.edu.

CODE OPTIMIZATION WITH STACK ORIENTED INTERMEDIATE CODE

A THESIS

SUBMITTED TO THE FACULTY OF ATLANTA UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF MASTER OF SCIENCE

BY

SATISHCHANDRA B. REDDY

DEPARTMENT OF MATHEMATICAL SCIENCES

ATLANTA, GEORGIA

MARCH, 1985

$R=0$ $T=16$

ACKNOWLEDGMENT

I acknowledge with thanks Dr. Bennett Setzer, Associate Professor, Department of Mathematical Sciences for his advice in the preparation of this thesis. I also thank Dr. Benjamin Martin, Professor and Chair, Department of Mathematical Sciences for his help and advice throughout my student life at the Atlanta University. Thanks are also due to Dr. Nazir Warsi, Professor, Department of Mathematical Sciences for his valuable suggestions.

TABLE OF CONTENTS

CHAPTER I	1 - 3
INTRODUCTION	1
MACHINE MODEL	1
MACHINE INSTRUCTIONS	1
CHAPTER II	4 - 5
STRUCTURE OF A COMPILER	4
CHAPTER III	6 - 16
ALGORITHM I	7
ALGORITHM II	11
APPENDIX A	
REFERENCES	

CHAPTER I

INTRODUCTION:

The problem discussed in this paper is related to the optimization of straightline segments of computer code. The general approach is to convert the straightline code into directed acyclic graphs with labels and then convert them into straightline code.

Two algorithms aimed at optimization are presented. Algorithm-I converts the given code into a DAG. Algorithm - II utilizes a DAG similar to the one produced by Algorithm-I and produces efficient code. The DAG may have been produced from the first DAG by optimization techniques. DAG to DAG optimization techniques are not discussed in this paper.

MACHINE MODEL:

The machine has a random access memory and the memory cells are numbered 0,1,2,3,... The machine has a built-in stack of infinite depth and all computations take place in this stack. It is assumed that the stack and memory do not overlap.

MACHINE INSTRUCTIONS:

The machine instructions are PICK n, ROLL n, STORE, FETCH, LIT n, OP x and DROP, where n is a positive integer or zero

and x is a binary operator. These operations are discussed below.

1. PICK n : Place a copy of the nth value from the top of the stack onto the top of the stack. e.g. if a,b,c,d,... (a on the top) is the initial stack configuration then after PICK 3 in the final stack configuration will be c,a,b,c,d,... Top is incremented by 1.

2. ROLL n : Remove the nth value from the top of the stack and place it on the top of the stack. e.g. if a,b,c,d,... are on the stack (a on the top) then the stack status after ROLL 3 will be c,a,b,d,...

3. STORE : Pop the top two elements from the stack and place Stack(top-1) into a memory location whose address is Stack(top). e.g. if m and n are on the stack (m on the top) then after STORE the contents of memory location m will be n.

4. FETCH : Replace the Stack(top) with the contents of the memory location whose address is Stack(top).

5. LIT n : Place the value n on the top of the stack and increment the top by 1.

6. OP x : Pop the top two elements of the stack and perform $x(\text{Stack}(\text{top}), \text{Stack}(\text{top}-1))$ and push this value onto the stack. The top is decremented by 1.

7. DROP : Remove the top element from the stack and decrement the top.

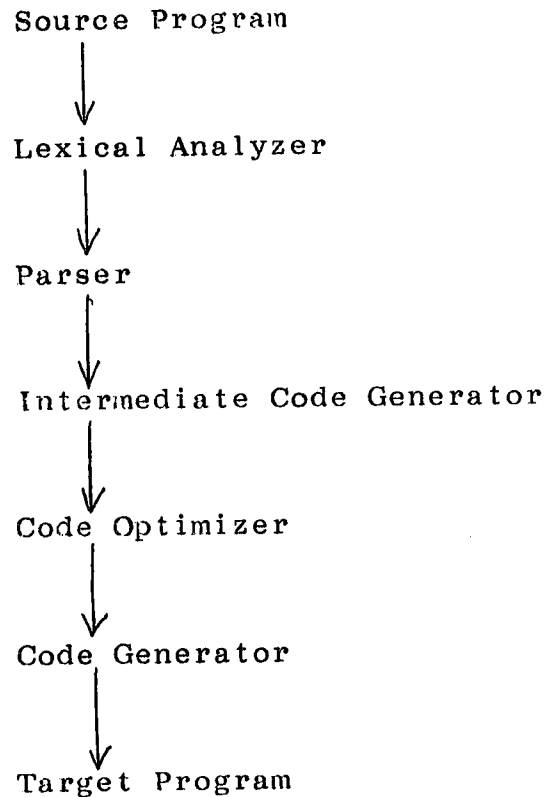
It may be noted that the operations 1 and 5 increase the size of the stack by 1; 2 and 4 neither increase nor decrease the size; 6 and 7 decrease the stack by 1 and the operation 3 decrease the stack by 2.

The study of programs for a stack oriented machine has practical importance in view of the existence of such machines (e.g. Burroughs 5000 family). It may be noted that the machine instructions discussed above are implemented in the Forth language. Also, some compilers produce intermediate code of this form.

CHAPTER II

STRUCTURE OF A COMPILER:

The following diagram describes a compiler.



The lexical analyzer takes the source program as input and identifies the basic lexical units of the program and reports any errors it discovers. The parser accepts the output of the lexical analyzer as input and verifies whether the source program satisfies the specifications of the language compiled and groups together the tokens into syntactic structures.

The intermediate code generator takes the output of the parser as input and breaks it into a sequence of straightline code segments called blocks and then produces a stream of instructions in the language described earlier. Such a block has two important properties which are described below.

1. The control in such a block enters only at the beginning and leaves only at the end.

2. Within a block the flow of control is sequential.

The code optimizer accepts the straightline code and generates a DAG. An algorithm for this conversion is given in the next chapter. A DAG is a directed acyclic graph with no cycles and with labels on nodes. Each node has a label which holds the instruction, the number of parents of the node and a number which indicates the order in which the node was created. Finally, the code generator takes the DAG with labels and emits code. An algorithm to this effect is presented in the next section.

CHAPTER III

The reported work on code optimization can be divided into two classes. The first consist of proven optimal methods and the second consist of code improvement methods. The former class is much smaller than the latter one. Many of the proven techniques involve optimizing arithmetic expressions which have no common subexpressions.

The optimization techniques are also classified as machine dependent or machine independent. The former class exploit the special features of the target machines which may have the following capabilities.

1. The machine has n accumulators or n registers.
2. The machine can execute several instructions in parallel.
3. The machine can execute instructions (both arithmetic and logical) upon multiple data streams.

The latter class, also called architecture independent techniques, use the nature of the data, operators in hand and are not dependent on the peculiarities of the machine. Some of the techniques are described below.

1. Constant folding: Performing the operations whose operands are known at the compilation time.

2. Identifying and removing the null operations. e.g. adding a 0 or multiplying by 1.

3. Making use of the algebraic properties such as commutativity, associativity and distributivity.

4. Code motion: Moving the loop-invariant computation out of the loop.

5. Eliminating redundant operations by identifying the common subexpressions.

6. Eliminating the dead or useless variables.

7. Loop jamming etc.

The problem of generating optimal code from DAGs has already been studied. Bruno and Sethi have shown that the problem is NP-complete for a single register machine. Aho, Johnson and Ullman have shown that the problem remains NP-complete even when the DAGs are almost trees (i.e., all nodes with more than one parent are interior nodes whose children are leaves).

We now present two algorithms. Algorithm-I constructs a DAG from a sequence of intermediate code block and Algorithm-II generates the code.

ALGORITHM-1 Constructing_a_DAG

INPUT A basic block.

OUTPUT : A DAG with the following information. Each node has three label fields-one holds a number which tells the order which the node was created; the second holds the instruction; and the third holds the number of parents the node has.

DATA STRUCTURES: It is assumed that suitable data structures are available to create linked lists of labeled nodes.

METHOD: The DAG construction process is to do the following steps 0 through 1 for each statement of the block. Initially the DAG is empty. A newly created node has the number of parents equal to 0 and the rest of the fields empty. Link fields are assumed to be null. We also assume the existence of a stack S.

STEP-0 : Number all the instructions in a sequence starting from 1 except the instructions PICK, ROLL and DROP.

STEP-1 : Read each instruction. Case(Instruction) of

LIT n : Create a node.

Set

Number_label = Instruction_number

Instruction_label = LIT n

Push a pointer to the node onto the stack S.

OP X : Create a node.

Set

Number_label = Instruction_number

Instruction_label = OP x

This node will have as its left and right children the the last two nodes (pointed to by the last two pointers on the stack) of the stack S. The parent field of each of these nodes is incremented by 1 and the top two nodes of the stack are popped and a pointer to the current node is pushed onto S.

PICK n : A copy of the contents of the nth node of the stack S is pushed onto S. The parent field of the child node is incremented.

ROLL n : The nth node in the stack is removed and its contents are placed on S.

STORE : Create a node.

Set

Number_label = Instruction_number

Instruction_label = STORE

This node will have as its left and right children the nodes pointed to by the top two nodes of the stack S.

The parent field of each of these nodes is incremented and then these two nodes are deleted from the stack.

FETCH : Create a node.

Set

Number_label = Instruction_number

Instruction_label = FETCH

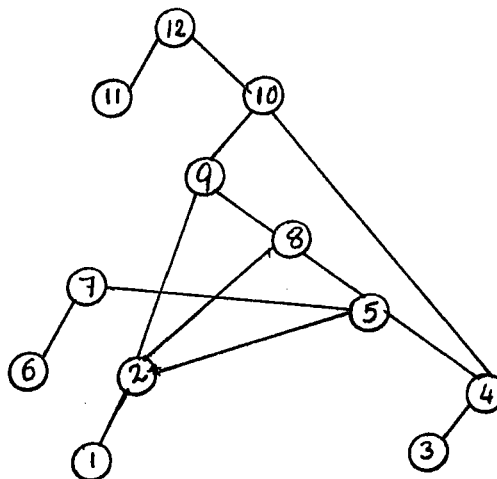
Replace the top of the stack by a pointer to the instruction number. Increment the parent field of the child node.

DROP : Delete the top node of S.

END. (Algorithm-I)

Tracing the algorithm with the following code produces the DAG shown below. (The code is shown on the left and the DAG on the right. The step by step derivation of the DAG from the code is given in the Appendix-A).

1. LIT 1
2. FETCH
3. LIT 2
4. FETCH
- PICK 2
- PICK 2
- PICK 2



5. OP +
 PICK 1
6. LIT 3
7. STORE
 PICK 2
8. OP +
 ROLL 2
9. OP +
10. OP -
11. LIT 4
12. STORE
 DROP

From the DAG it is clear that all the parent node numbers are higher than those of their children. Parents use the data computed by the children. It may be noted here that the parents are created after their children. i.e., the DAG is constructed bottom-up.

We shall now present an algorithm which takes the DAG as input and outputs the code.

ALGORITHM - II Producing_the_Code_From_the_DAG

INPUT : The root nodes of the DAG.

OUTPUT : The straight line code.

5. OP +
 PICK 1
6. LIT 3
7. STORE
 PICK 2
8. OP +
 ROLL 2
9. OP +
10. OP -
11. LIT 4
12. STORE
 DROP

From the DAG it is clear that all the parent node numbers are higher than those of their children. Parents use the data computed by the children. It may be noted here that the parents are created after their children. i.e., the DAG is constructed bottom-up.

We shall now present an algorithm which takes the DAG as input and outputs the code.

ALGORITHM - II Producing_the_Code_From_the_DAG

INPUT : The root nodes of the DAG.

OUTPUT : The straight line code.

METHOD : Follow each root node and traverse the tree in any one of the orders viz., preorder, postorder or inorder. As the tree pertaining to the root is traversed put all the nodes in a list (let us call this list as the old_list) if the node visited is not already in the list. Repeat this procedure until all the root nodes are exhausted. Now sort this list by the node number.

For every node in the old_list do the following :

(If) The node is a leaf (Then)

(If) the node has one parent (Then)

1. Generate the instruction.
2. Attach the instruction number to the new_list

(Else) (i.e. the node has more than one parent)

1. Generate the instruction.
2. Attach the instruction_number to the new_list.
3. Reduce the number of parents of the node by 1.

(Else) (i.e. the interior node or a root)

(If) The node has one child (Then)

(If) The node has one parent (Then)

(If) The node is at the end of the new_list (Then)

1. Generate the instruction.
2. Replace the previous instruction number by the present instruction number in the new_list.

(Else) (i.e. the node is not at the end of the

new_list)

1. Print 'ROLL k' where k is the position of the (child) node in the new_list.
2. Remove the kth node from the list.

(Else) (i.e. the node has two or more parents)

1. Print 'PICK k' where k is the position of the node in the new_list.
2. Attach a copy of the kth node in the new_list to the new_list.

(Else) (i.e. the node has two children : The instruction contained must be either OP x or STORE.)

A. (If) The right child has one more parent other than the present (Then)

1. Print 'PICK k' where k is the position of the node in the new_list.
2. Reduce the number of parents of this node by 1.
3. Attach a copy of the kth node in the new_list to the new_list.

(Else)

Print 'ROLL k' where k is the position of the node in the new_list, if the child is not at the end of the new_list. Remove the kth node from the new_list and add it to the end of the new_list if 'ROLL k' is performed.

B. Repeat Step - A (above) for the left child.

Now

1. Generate the instruction.
2. Delete the last two elements from the new_list and add the current instruction number (old_list) to the new_list if the instruction is OP x and has no parents after deleting the last two elements from the list. Otherwise just delete the last two elements from the list.

END. (Algorithm-II)

Algorithm-II applied to the DAG of page 10 produces the following code. The numbers that appear on the right denote the status of the stack at any given stage.

1. LIT 1	1
2. FETCH	2
3. LIT 2	2,3
4. FETCH	2,4
PICK 1	2,4,4
PICK 3	2,4,4,2
5. OP +	2,4,5
6. LIT 3	2,4,5,6
PICK 2	2,4,5,6,5
ROLL 2	2,4,5,5,6
7. STORE	2,4,5
PICK 3	2,4,5,2

8. OP +	2,4,8
ROLL 3	4,8,2
9. OP +	4,9
10. OP -	10
11. LIT 4	10,11
12. STORE	

While the code was produced by the Algorithm-II, all the nodes that were in the old_list was kept in the new_list and whenever a node was processed from the new_list, the number of parents of each node was reduced. So if a node had n parents, then that node remained in the list as long as it was not processed n times or until the number of parents of that node was greater than 0.

If the node was a leaf node, then the instruction was printed as it was and if the node was not a leaf then the algorithm searched the list and got the right and left children of the node and produced either PICK n or ROLL n depending upon the number of parents of that child node, n being the position of the node in the new_list. Also, it may be noted that the nodes of the list encountered was the same as that of the original code. By induction the two codes (both the original as well as the one produced by the Algorithm -II) produce the same result.

CONCLUSION:

The problem of producing stack oriented code from an intermediate code was discussed in this paper. The approach was to translate the intermediate code into DAGs and then to convert the DAGs back into code. We have presented two algorithms to this effect and have shown intuitively that they produce equivalent code. DAG to DAG optimization was not considered. Much light has to be shed on this aspect. The author feels that further studies should be done in this area. A transformed, possibly an optimized DAG, fed as input to Algorithm-II may produce more efficient code.

APPENDIX - A

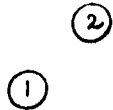
Step by step derivation of the DAG for the example on page 10.

1. LIT 1



1

2. FETCH



2

3. LIT 2



2, 3



4. FETCH



2, 4



PICK 2

Same DAG

2, 4, 2

PICK 2

Same DAG

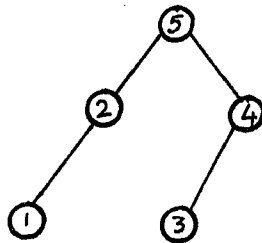
2, 4, 2, 4

PICK 2

Same DAG

2, 4, 2, 4, 2

5. OP +



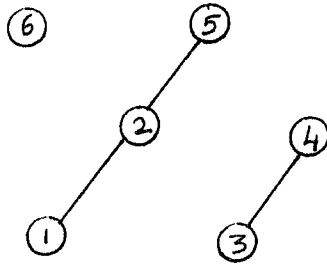
2, 4, 2, 5

PICK 1

Same DAG

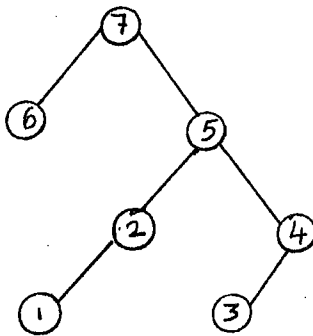
2, 4, 2, 5, 5

6. LIT 3



2, 4, 2, 5, 5, 6

7. STORE



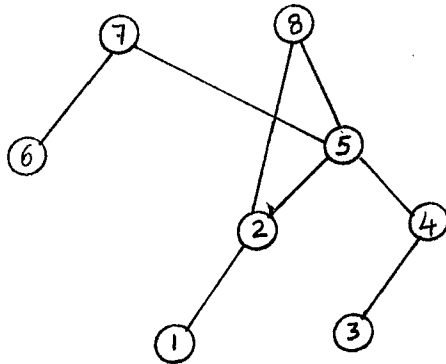
2, 4, 2, 5

PICK 2

Same DAG

2, 4, 2, 5, 2

8. OP +



2, 4, 2, 8

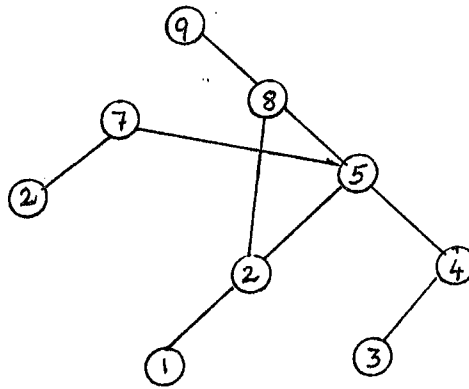
ROLL 2

Same DAG

2, 4, 8, 2

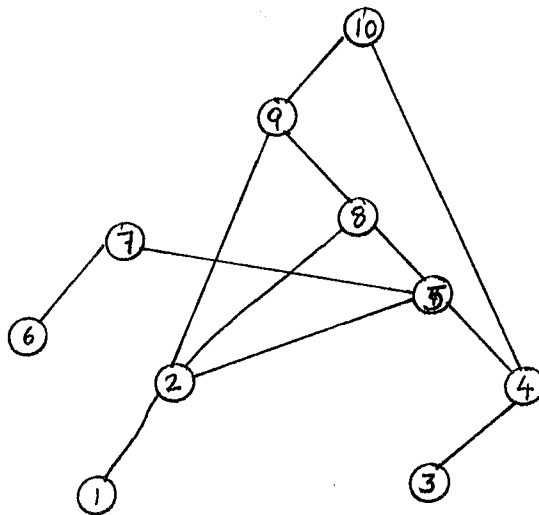
9. OP +

2, 4, 9

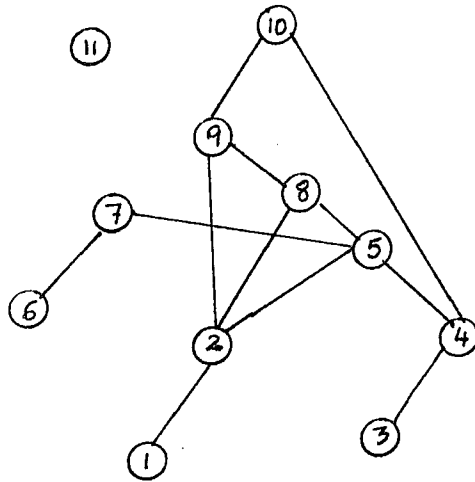


10. OP -

2, 10

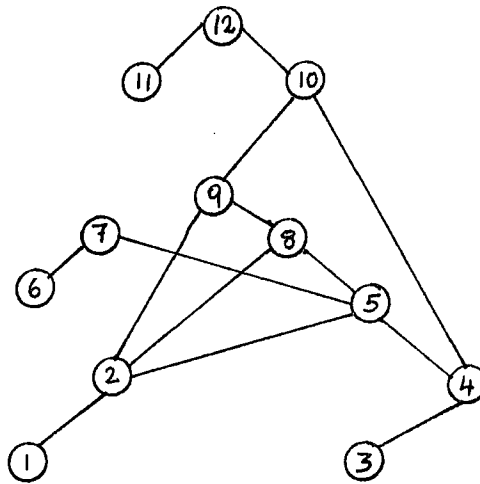


11. LIT 4



2, 10, 11

12. STORE



2

DROP

REFERENCES

1. Aho, A.V., Johnson, S.C., Ullman, J.D., [1977], "Code Generation for Expressions with Common Subexpressions," J.ACM 24:1, 146-160.
2. Aho, A.V., Ullman, J.D., [1979], Principles of Compiler Design, Addison-Wesley Publishing Company, Reading, Mass.
3. Barret, William A., Couch, John D., [1979], Compiler Construction : Theory and Practice, Science Research Associates.
4. Bruno, J.L., Lassagne, T., [1975], "The Generation of Optimal Code for Stack Machines," J. ACM 22:3, 382-397.
5. Bruno, J.L., Sethi, R., [1976], "Code Generation for One Register Machine," J. ACM 23:3 502-510.
6. Sethi, R., [1975], "Complete Register Allocation Problems," SIAM J. Computing, 4:3, 226-248.
7. Sethi, R., Ullman, J.D., [1970]. "The Generation of Optimal Code for Arithmetic Expressions," J.ACM 17:4, 715-728.