Code Optimization by Reducing the Number of Stack Restoration Commands

Yuval Shavitt
Department of Computer Science
Johns Hopkins University,
Baltimore, MD 21218-2691
e-mail: shavitt@cs.jhu.edu

Osnat Yehieli
Department of Electrical Engineering
Technion - Israel Institute of Technology,
Haifa 32000, ISRAEL
e-mail: osnaty@tx.technion.ac.il

Abstract

This paper addresses the problem of optimizing code by reducing the number of stack restoration commands. The optimization is done on the assembler translation of each function and thus can be applied to various compilers. This leads us into the following graph theory problem: for a given node s in a general digraph find a node t s.t. *every* path that starts from s ends in t, and *every* path that ends in t starts in s. We give an algorithm that solves the graph problem in $O(|E|)$ time complexity, where $|E|$ is the number of edges in the graph. The optimization problem is solved in $O(k \cdot |E|)$ time complexity, where k is the number of basic blocks with stack translation commands in the assembler translation of a function.

1 Introduction and Problem Description

In compilers that use a calling convention (our first motivation was a C compiler) where on a return from a called function a stack restoration is done, the number of stack restoration commands may be significantly large. For example, suppose that in the assembler output of such a compiler, two succeeding *basic blocks* (a basic block is a sequence of commands that ends with a *jump* or a *label*) include function calls, i.e., in each one of the blocks the Stack Pointer (SP) is decreased (push) and later increased (pop). A simple observation is that, if memory for the stack is sufficient, we can optimize this 'code' by increasing the stack only once in the second block by the sum of the two original increases. Of course, if there is a possibility of a jump from the first block to a third different block, this optimization is erroneous. The problem of finding the optimal number and place of stack restoration commands in a general structured high-level-function, translates into an interesting graph theory problem.

Problem Description

Consider an assembly translation of a high-level-function. We define a *basic block* (sometimes we simply say *block*) as a sequence of assembler commands that ends with a *jump* or a *label.* We associate each block with a non-negative integer *value* that represents the number by which the stack is incremented in this block. The assembly translation is represented by a Control Flow Graph (CFG) [ASU86]. The CFG is a digraph G(V,E), where V is the set of blocks, and $e = (v, u) \in E \subseteq V \times V$ iff there is a possibility that block u will be executed immediately after block v.

Optimization can be done using the following transformations:

1. All the immediate predecessors of a positive value block have out degree of one and a positive value (See the transform from figure 4 to figure 5), or the block value may be zero but all its immediate predecessors have out degree of one and the same positive value.
   Let n be the minimal value of the predecessor blocks. Reduce n from each of the predecessor blocks, and add n to their successor. As a result, at least in one of the blocks the stack increment is set to zero and thus becomes redundant.

2. Find two nodes in the graph s and t s.t. s has a positive value, and in every (not necessarily simple) path from the function entry block to the function exit block s and t appear the same number of times. (See the transforms from figure 1 to figure 2, from figure 2 to figure 3, and from figure 3 to figure 4. See also the examples in figure 7).
   The value of s is added to the value of t, and s is set to zero, thus, an increase instruction is omitted. If t had a positive value before this transformation, a command is saved, otherwise, we still do the transformation since it does not increase the number of instructions in any computation path but it might enable us to perform the first transformation for t's child.

To optimize an assembler translation of a high level function (we assume that each function has one entrance point and one exit point) we try to apply the transformations on the nodes in the digraph in a BFS order as follows:

- Try to apply transformation 2 with the function entrance node as the source node.
- Examine the other nodes in the digraph (except the exit node) in a BFS order. For each one of the nodes, try to apply transformation 1 and then try to apply transformation 2.
- Try to apply transformation 1 on the function exit node.

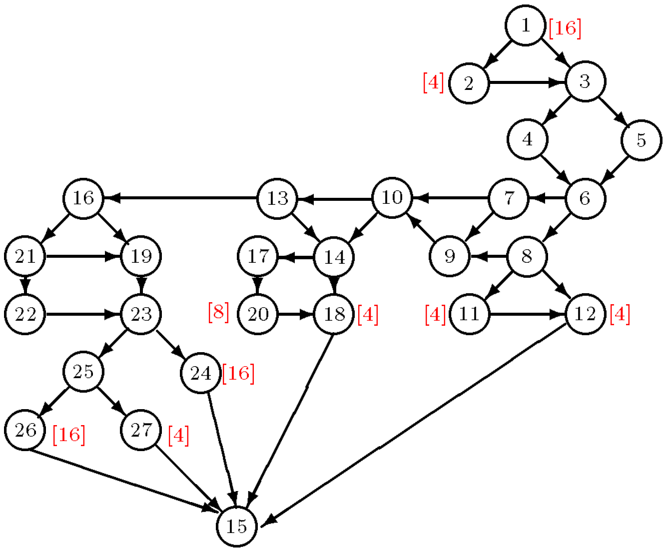

Figure 1: The original digraph

Adhering to this order, enables us to propagate the stack increase commands through several successive transformations. We believe that the above scheme brings us to a global optimum in one pass. Figures 1 – 5 show a complete example of the algorithm operation. The value in brackets near a node is the offset by which the stack is restored in the block.

2 Algorithm Description

To create a BFS order of the nodes we use a simple BFS algorithm that requires $O(|E|)$ operations. While scanning the nodes we also create for each node a pointer list of all its direct predecessors (nodes that point to it) and update an $InDegree$ counter without additional complexity cost.

To check if transformation 1 can be applied we simply check all the node's direct predecessors. This costs us $O(InDegree)$ for a single node, and $O(|E|)$ for all the nodes since every edge is checked at most once (if the first predecessor node is found to have a 0 value or more than one successor, no further checking is performed).

For the second transformation we give an algorithm that finds a sink if one exists, and otherwise returns a negative result, with $O(|E|)$ worst case time complexity. Alternatively this problem can be solved for node s by finding a node t s.t.

- s is the immediate dominator of t in the program digraph,

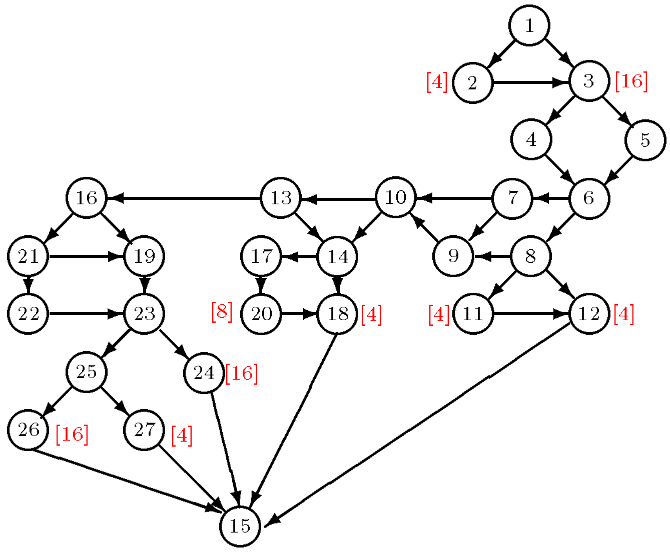

Figure 2: Applying transformation 2 to push a SR (Stack Restoration) operation from node 1 to node 3

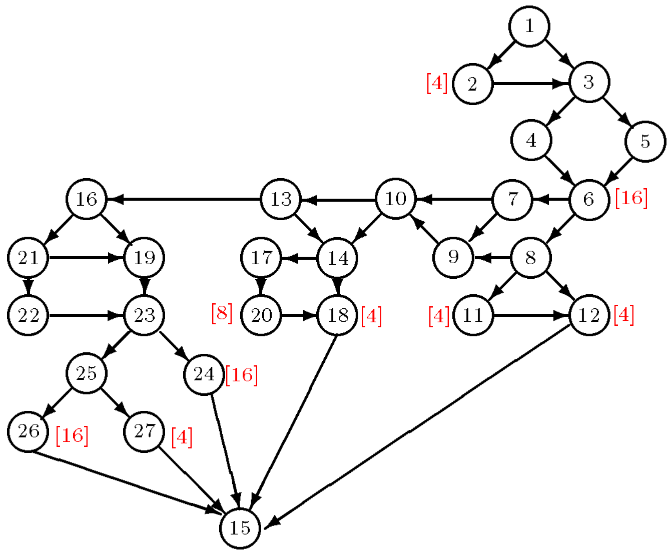

Figure 3: Applying transformation 2 to push a SR operation from node 3 to node 6

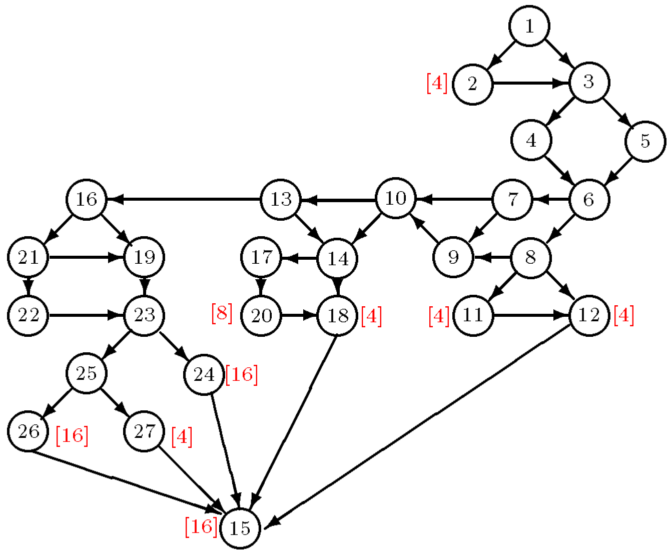

Figure 4: Applying transformation 2 to push a SR operation from node 6 to node 15

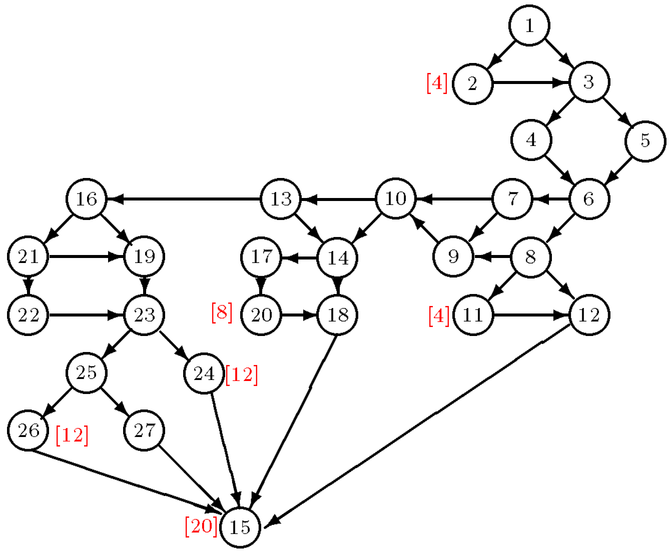

Figure 5: Applying transformation 1 to push SR operations from nodes 12, 18, 24, 26, and 27 to node 15

- t is the immediate dominator of s in the reversed program digraph (the program digraph with the edges reversed),
- For every edge $(x,\ s)$, $x \neq t$: s does not dominate x in the program digraph, and
- For every edge $(t,\ y)$, $y \neq s$: t does not dominate y in the reversed program digraph.

Tarjan [Tar74] showed how dominator trees can be built in $O(|V| \cdot \log|V| + |E|)$ operations. Lengauer and Tarjan [LT79] find dominator trees in $O(|E| \cdot \log|V|)$ operations or in $O(|E| \cdot \alpha(|E|,\ |V|))$ operations for a more sophisticated algorithm, where $\alpha(m,\ n)$ is the functional inverse of Ackerman's function [CLR90]. For the case where $|E| < O(|V| \cdot \log|V|)$ it improves the previous algorithm by Tarjan [Tar74]. Wegman's algorithm [Weg83] can be also used to find dominator trees. However, for general digraphs it might require $O(|V|^3)$ operations. In practice, we believe that the search for the sink should be depth limited. Most of the gain from this transformation is from what can be saved in internal loops. Limiting the search depth can reduce the algorithm execution time while the cost of not detecting some source-sink pairs is negligible. A limited search will result in execution time that is linear in the number of edges.

Suppose first that the digraph contains no cycles. To look for a sink we do the following. First we disconnect all the edges pointing to the source by deleting them from the outgoing edge list. These edges are considered separately in the algorithm operation. Next we let the source node inject a unit flow to the network. The flow is divided (equally) among all the outgoing edges and the flow entered to each node is accumulated in this node (in the variable $Weight$). A node that receives flow from all its incoming edges (enters state *Ready*) divides the sum of all the flows it receives among its out-going edges. The algorithm fails if such a node has a direct outgoing link to the source. The algorithm terminates with success if the sum of the flow a node receives equals to 1.

A cycle in the digraph, may deadlock the algorithm as demonstrated in the example of figure 6. Both node 1 and node 2 receive flow $\frac{1}{2}$ from the source S. At this point the algorithm cannot proceed as both nodes are waiting to receive flow from their incoming unmarked link, and no node is in a state that permits it to inject flow (or in the terms of the formal description both nodes are in state $InWork$, and no node is in state *Ready*). When such a condition is reached, a cycle detection algorithm is activated. The algorithm uses DFS on the digraph made from the edges that are still in their initial state (in state *Basic*) when the edge direction is reversed. When a cycle is detected, all the nodes that form the cycle are joined to one virtual node, i.e., their $Weight$ is summed, and their incoming and outgoing edge lists are joined. For a formal description of the algorithm see the appendix. Since no edge is considered more than once, the algorithm complexity is $O(|E|)$.

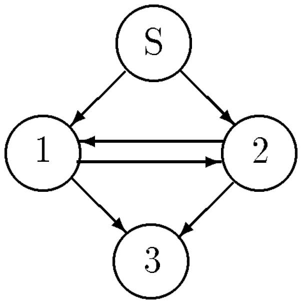

Figure 6: An example of a cycle in the digraph

3 Concluding Remarks

We suggested a code optimization technique that reduces code execution time (and space) by reducing the number of stack restoration commands. To this end we defined the problem of finding a source-sink pair in a function digraph such that in every path from the function ebtry point to the function exit point the two nodes of the pair appear the same number of times. We described two solutions to this problem: one using dominator trees and the other based on a new flow-oriented approach. We believe that this new approach may be useful for a wide range of applications, and therefor is important.

4 Acknowledgments

We would like to thank Intel, Israel for the use of their compiler generated assembler code.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[CLR90] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121 – 141, July 1979.

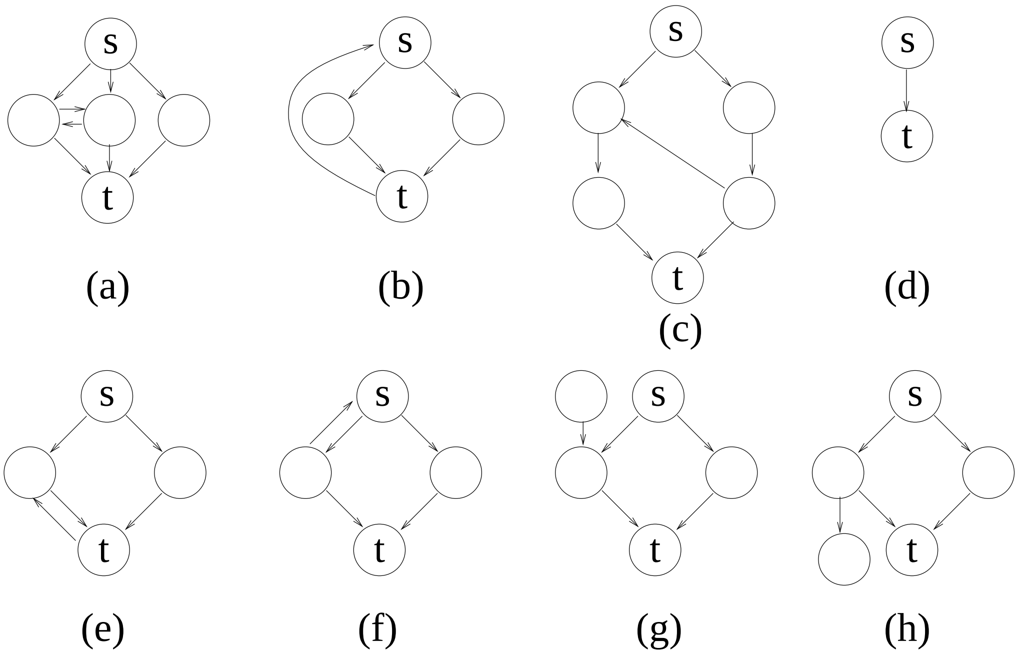

Figure 7: Examples of cases where transformation 2 is legal (a) – (d) and illegal (e) – (h).

[Tar74] Robert Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62 – 89, March 1974.

[Weg83] Mark Wegman. Summerizing graphs by regular expressions. In *10th Sym. on Principles of Programming Languages*, pages 203 – 216. ACM, 1983.

Appendix

A Transformation 2 - Formal Description

Node states:

Basic Initial state.

InWork A node enters this state the first time the algorithm has reached it.

Ready A node enters this state when all its incoming edges were marked.

Done A node enters this state when the algorithm has finished marking all its outgoing edges.

Edge states:

Basic Initial state.

Marked When marked by the 'flow algorithm'.

Candidate Marked by the cycle detection algorithm. as candidate to participate in a cycle.

Cycle An edge that is part of a cycle.

Data structure for each node v:

$State(v)$ Keeps the node state. Can get one of the four values: *Basic*, *InWork*, *Ready*, *Done*. Initialized to *Basic*.

$In(v)$ A list of edges pointing to v.

$Out(v)$ A list of edges emanating from v.

$InDegree(v)$ The number of in-coming edges to v in state *Basic*.

$OutDegree(v)$ The number of out-going edges of v in state *Basic*.

$Weight(v)$ A quota used by the algorithm to describe the portion of the flow emanating from the source and passing through v.

$Value(v)$ The value that is added to the stack-pointer at this node. Can take values $4k$, $k = 1, 2, 3, \ldots$.

$Virtual(v)$ A flag that is set when the node is virtual, i.e., a union of nodes.

Data structure for each edge e:

$State(e)$ Keeps the state of the edge. Can get one of the four values: *Basic*, *Marked*, *Cycle*, *Candidate*.

Other data structures:

Ready A list of nodes in state *Ready*.

InWork A list of nodes in state *InWork*.

Source The source node.

The algorithm for a given source s:

```
State(s) ← Ready
Weight(s) ← 1
Ready ← s
InWork ← φ
for every e ∈ In(s) do
begin /* Disconnect all edges pointing to s */
    state(e) ← source
    let u be the node e is emanating from
    Out(u) ← Out(u) \ {e}
    OutDegree(u) ← OutDegree(u) − 1
end
Repeat forever
    Select v from Ready
    Ready ← Ready \ {v}
    for every e ∈ Out(v) s.t. state(e) = Basic do
    begin
        Let u be the node pointed by e
        Weight(u) ← Weight(u) + Weight(v)/OutDegree(v)
        InDegree(u) ← InDegree(u) − 1
        if InDegree(u) = 0 then
        begin
            if State(u) = InWork then
                InWork ← InWork \ {u}
            if (exist e′ emanating from u s.t. State(e′) = Source AND Weight(u) < 1) then
                exit(Fail)
            Ready ← Ready ∪ {u}
            State(u) ← Ready
        end
        else /* InDegree(u) > 0 */
            if state(u) = Basic then
            begin
                InWork ← InWork ∪ {u}
                state(u) = InWork
            end
        /* check termination */
        if Weight(u) = 1 then
            if State(u) = Ready then
                if Virtual(u) = false then
                    CheckSuccess(u) /* Algorithm succeed ?*/
                /* in case of a virtual node - termination cannot be achieved */
    end
    if Ready = φ then
        CycleDetection()
```

end repeat

Cycle detection algorithm (DFS):

CycleDetection
For every node $v \in InWork$
do DFS in the reverse edge direction on *Basic* edges only that are not connected to node with id smaller than s.
if detected a cycle
    join all nodes in the cycle to a virtual node
    return
else
    exit(Fail)

$CheckSuccess(u)$:

for every $e \in In(s)$ do
begin
    let v be the node e is emanating from
    if $state(v)! = Basic$ AND $v! = u$ then
        exit(Fail)
end
exit(u)