Optimizing Register Spills for Eager Functional Languages

S. Mishra[1], K. Sikdar[1], and M. Satpathy[2]

[1] Stat-Math Unit, Indian Statistical Institute, 203 B. T. Road, Calcutta-700 035
[2] Dept. of Computer Science, University of Reading, Reading RG6 6AY, UK
(res9513, sikdar)@isical.ac.in, M.Satpathy@reading.ac.uk

Abstract. Functional programs are *referentially transparent* in the sense that the order of evaluation of subexpressions in an expression does not matter. Any order of evaluation leads to the same result. In the context of a compilation strategy for eager functional languages, we discuss an optimization problem, which we call the *Optimal Call Ordering Problem*. We shaow that the problem is NP-complete and discuss heuristics to solve this problem.

Keywords: *NP-Completeness; Functional Languages; Register Utilization*

1 Introduction

Functional programs are *referentially transparent* [3]. This means, the value of an expression never changes throughout its computational context. As a consequence, we can evaluate an expression in any order. Consider the expression $f(x, y) + g(x, y)$. If it is an expression in a functional program, the values of x and y remain the same throughout the computation of the expression. So, we can do any of the following to get the final result: (i) first make the call to f, then make the call to g and add their results, or alternatively, (ii) first evaluate the call to g, then the call to f and add their results.

So far as evaluation order is concerned, functional languages are divided into *eager functional languages* and the *lazy functional languages*. In an eager functional language, the arguments of a function are evaluated before the call to the function is made, whereas, in a lazy functional language, an argument of a function is evaluated only if it is needed. If two operands are present in machine registers, operations like additions and subtraction usually take one machine cycle. On the other hand, if the operands are present in memory, they are first brought to registers, and then the addition or the subtraction operation is performed. Thus memory operations are usually far more costlier than the register operations. So it is always preferable that, as much operations as possible should be done in registers. But it is not always possible since the number of registers is small, and the number of values which are *live* at a given point of time could be high [2]. Further, function calls and programming constructs like *recursion* complicate the matter. When we cannot accommodate a *live* value in

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2074, pp. 128–137, 2001.
© Springer-Verlag Berlin Heidelberg 2001

a register, we need to store it in memory. Storing a register value in memory and retrieving it later to a register constitute a *register spill* or simply a *spill*. It is desired that a program should be computed with minimum number of spills.

In this paper, we will discuss an optimization problem which aims at minimizing the number of register spills in the context of a compilation strategy for eager functional programs. Section 2 introduces the problem. Section 3 discusses the context of the problem. Section 4 shows that the problem is NP-complete. In Section 5, we discus about heuristics for this problem. Section 6 concludes the paper.

2 Our Problem

We make the following assumptions: (i) The language is an eager functional language, and (ii) a function call may destroy the contents of the live registers. The later assumption is necessary because we shall have to deal with recursion. We now illustrate our problem through an example.

Example 1: Figure 1 shows the *directed acyclic graph* (DAG) for the following expression. The arguments of calls F_1 and F_2 share subexpressions.
$F_1(1 + (x * (2 - (y * z)))) + F_2((x * (2 - y * z)) + ((2 - y * z) + (y * z/3)))$
We can evaluate calls F_1 and F_2 in any order. For evaluating F_1, we need to evaluate the argument of F_1 and hence the expression DAG enclosed within **LP2**. Similarly, before making the call to F_2, we have to evaluate the argument of F_2 which is the expression DAG enclosed within **LP3**. LP2 and LP3 have many shared subexpressions between them. Now let us see the relative merits and demerits of the two possible orders of evaluation. Note that we are only considering the number of spills which result due to the shared nodes between the argument DAGs of F_1 and F_2.

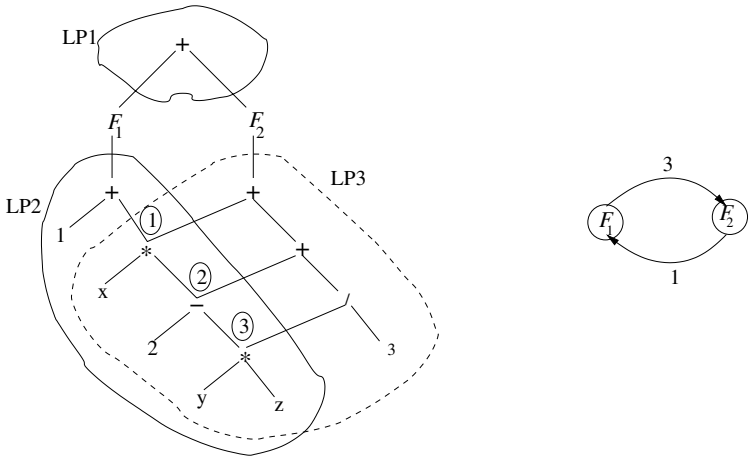

Fig. 1. Sharing between the arguments of two calls F_1 and F_2

- *Evaluation of F_1 is followed by F_2:* We have to first evaluate LP_2. Then the call to F_1 will be made. Next, LP_3 will be evaluated and it will be followed by the call to F_2. Observe that, during the evaluation of the DAG within LP_2, it will evaluate three shared computations, marked by circled 1, 2 and 3 in the figure. Immediately after the evaluation of LP_2, they are in registers. But following our assumptions, the call to F_2 may destroy them. So when LP_3 will be evaluated they may not in the registers in which they got evaluated. So we need to store them in memory. In conclusion, we need to make 3 spills.
- *Evaluation of F_2 is followed by F_1:* We have to first evaluate LP_3. Then the call to F_2 will be made. After that LP_2 will be evaluated and it will be followed by the call to F_1. Observe that, during the course of LP_3's evaluation, it will evaluate only one shared computations, marked by circled 1 in the figure, which will be needed by LP_2. So in memory we need to store its value and retrieve it when LP_2 needs it. In conclusion, we will make 1 spill.

We observed that the second approach is more efficient than the first approach since we need to make lesser number of spills. Note that we are not just saving two or three spills. Such calls may occur in a recursive environment. In such a case the number of spills that we save by choosing a better evaluation order could be very very large. So, choosing the better evaluation order is important.

3 The Problem Context

For computing an arithmetic expression, it is usually represented as a DAG [2]. Such a DAG is simple in the sense that the interior nodes are machine operators and the leaf-nodes are either literals or memory locations. The problem of computing a simple expression DAG with minimum number of instructions (or minimum number of registers such that no spill occurs) is NP-complete [1]. Aho, Johnsson and Ullman have discussed various heuristics for computing an expression DAG. One such heuristic is called the *top down greedy* and we will refer it as the AJU algorithm in our discussion. Satpathy et al. [8] have extended the AJU algorithm to generate code for an expression in an eager functional language. The DAG that can represent a generalized expression in a functional language may have function calls or if-expressions as interior nodes. Figure 2 shows one such DAG. A node representing a function call or an if-expression is termed as a non-linear node. The strategy as mentioned in [8] partitions a generalized expression DAG into linear regions. The linear regions are the maximal regions of a DAG such that its interior nodes are all machine operators, and their leaves could be non-linear nodes. Figure 2 shows the linear regions as the regions in dashed lines.

A strategy for evaluating a generalized expression DAG could be as follows. Here, for simplicity, let us assume that the DAG has function calls as the only non-linear nodes. Our analysis remains valid in presence of if-expressions but their presence will make the analysis look complicated.

Step 1: Partition the DAG into linear regions. [8] discusses such a partitioning algorithm.

Step 2: Select any linear region that lies at the leaf-level of the DAG (such a linear region do not have any non-linear node at any of its leaves). Evaluate this non-linear region using the AJU algorithm and store its result in memory. Now replace this non-linear region in the DAG by a leaf node labeled with the above memory location.

Step 3: Evaluate all function calls whose arguments are available in memory locations. Replace the calls in the DAG by their results in memory locations.

Step 4: Continue steps 2 and 3 till the result of the original DAG is available in a single memory location.

Alternatively, we can express the above strategy as follows:

- First evaluate all the nonlinear nodes lying at the leaves of a linear region one by one and spill their results. Then evaluate rest of the linear region using the AJU algorithm assuming that the results of all the nonlinear nodes at the leaves are in memory.

Note that when all the non-linear nodes of a linear region are already evaluated, their results are in memory. So, at this point of time, the linear region under evaluation is a simple expression DAG and so it could be evaluated using the AJU algorithm. Further notice that, this is a bottom-up strategy. We elaborate it through the evaluation of the DAG in figure 2. In the figure, the linear regions LR1 till LR4 are shown as the regions in dotted lines. For evaluating such a DAG, evaluation always starts with a linear region such that none of its leaf-level nodes is a non-linear node. In the present figure, LR2 and LR4 are the only candidates for evaluation. We can evaluate any of them first. Let us choose to evaluate LR4 (the argument of function J). After evaluation of LR4 (its result is in memory), a call to J is made since it satisfies the step 3 of the evaluation strategy. Let the result of J be stored in memory location m_2. Now LR3 is one linear region whose only non-linear node is available in memory. So LR3 could be evaluated by using AJU algorithm. After the evaluation of LR3, a call to H is made and let us store its result in m_3. Next the linear region which is a candidate for evaluation is LR2. It is evaluated and a call to G is made. Let the result of this call be stored in m_1. At this stage, LR1 could be evaluated using AJU algorithm since the results of all of its non-linear nodes are in memory. Let us store the result of LR1 in location m_0. And this the desired result.

3.1 Our Problem in General Form

In Figure 3, function calls $F_1, F_2, \ldots, F_n$ lie at the leaf-levels of the linear region LP. DAG_1,..., DAG_n respectively represent the arguments of such calls. For simplicity, we have assumed that all such functions are single argument functions. Further, we can assume that there are no non-linear nodes present in such argument DAGs (if they are present, they will have already been evaluated and their results will be residing in memory). So, all such argument DAGs are simple expression DAGs and they could be evaluated using the AJU algorithm in any order.

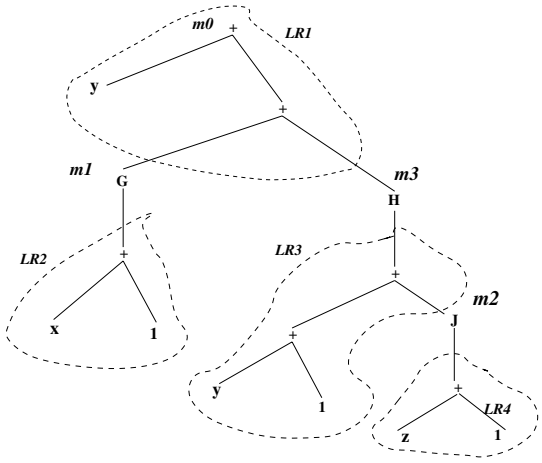

Fig. 2. Evaluation of a generalized DAG through a bottom-up strategy

$F_1, F_2, \ldots, F_n$ will induce, due to the sharing between the argument DAGs, certain number of spills which are as discussed in the previous section. So now the problem is: how to choose an evaluation order of the calls to $F_1, ..., F_n$ such that the number of spills due to the sharing among argument DAGs is minimum. From now on we will refer to this problem as the *optimal evaluation ordering problem (OEOP)*. Formally, OEOP can be defined as follows:

Instance : A linear region whose leaves have function calls $F_1, F_2, \ldots, F_n$ at its

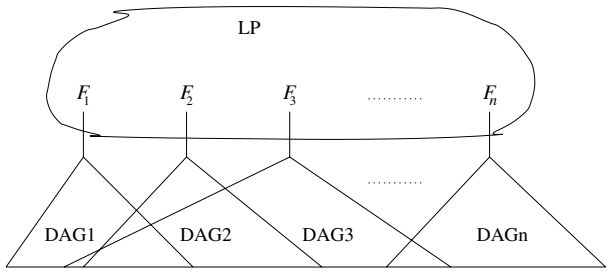

Fig. 3. Sharing between arguments of n calls $F_1, F_2, \ldots, F_n$

leaf-level, and the calls respectively have DAG_1, ..., DAG_n as argument DAGs. All such DAGs are simple expression DAGs (refer to Figure 3).

Solution : A permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ of $(1, 2, \ldots, n)$.

Cost : $w(\pi)$ = number of shared nodes to be spilled to evaluate n function calls (due to sharing between the argument DAGs) in the order $F_{\pi_1}, F_{\pi_2}, \ldots, F_{\pi_n}$.

Goal: Minimize the cost function.

3.2 Assumptions and Problem Formulation

To analyse the OEOP, we will make the following simplifying assumptions.

- *All the functions are of one arguments:* This is to make the analysis simpler.
- *Assumption of simple-sharing:* An expression is shared between at most two function DAGs. Further, if a subexpression is shared by two calls F_i and F_j then no subexpression of it is shared by any function F_k $(k \neq i, j)$.

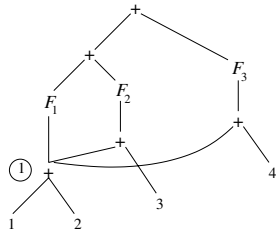

Fig. 4. DAGs of A, B and C not satisfying the assumption of simple sharing

In Figure 4, the node marked 1 is shared by the three functions A, B and C. Let B be evaluated before C. Now, whether B will spill node 1 for C depends on whether A has been evaluated already or not (i.e. whether A has spilled the node 1 or not). In other words, the spilling decisions between B and C depends on the context in which they are evaluated. To make such spilling decision between any two function calls independent of the context in which they are evaluated, we have introduced the assumption of simple sharing. We shall denote the OEOP satisfying above assumptions as OEOP(S). Obviously, OEOP(S) is in class NP.

4 NP-Completeness of OEOP

Let Figure 3 represent an instance of OEOP(S). Given n calls as in the figure, we can obtain a weighted symmetric digraph $G = (V_n, A, w)$ which will preserve all the sharing information. In the digraph, the node set $V_n = \{1, 2, \ldots, n\}$, where node i corresponds to the function F_i. There will be a directed edge from node i to node j with weight w_{ij} if the argument DAG of F_i *cuts* the argument DAG of F_j at w_{ij} points. What it means is that if the call to F_i is evaluated ahead of the call to F_j, then w_{ij} shared nodes will have to be spilled to memory. Note that the weight of the edge from i to j may be different from the weight of the edge from j to i. For instance, if we construct a digraph for the functions A and B in the figure 1, then there will be an edge from node A to B with weight 3 and the reverse edge will have weight 1.

Let us evaluate the calls $F_1, \ldots F_n$ in the figure in the order $F_{\pi_1}, F_{\pi_2} \ldots F_{\pi_n}$, where $\pi = (\pi_1, \ldots \pi_n)$ is a permutation of $(1, 2, \ldots n)$. Then the number of spills

we will have to make immediately after the evaluation of F_{π_1} is the sum of the weights of all the out-going edges from π_1. The number of spills we will make immediately after the evaluation of F_{π_2} is the sum of weights of the out-going edges from π_2 to all nodes excepting π_1. This is because F_{π_1} has already been evaluated, and π_2 should not bother about F_{π_1}. Continuing in this manner, the number of spills we will make after the evaluation of F_{π_j} is the sum of weights of all outgoing edges from π_j other than the edges to $\pi_1, \pi_2, \ldots \pi_{j-1}$. Let us call the number of spills that are made after the evaluation of F_{π_i} be SPILL_i. So the evaluation order $F_{\pi_1}, F_{\pi_2}, \ldots F_{\pi_n}$ will make it necessary to make $\text{SPILL}(F_{\pi_1}, F_{\pi_2}, \ldots F_{\pi_n}) = \text{SPILL}\pi_1 + \text{SPILL}\pi_2 + \ldots + \text{SPILL}\pi_n$ spills. Note that the value of $\text{SPILL}\pi_n$ will be zero since after its evaluation we need not make any spill. To find out the minimum number of spills we shall have to find out the value of $\text{SPILL}(F_{\pi_1}, F_{\pi_2}, \ldots, F_{\pi_n})$ for all possible evaluation order $F_{\pi_1}, F_{\pi_2}, \ldots, F_{\pi_n}$ and take the minimum of these values.

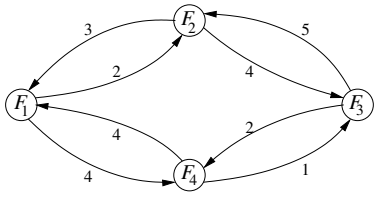

Fig. 5. Graph showing sharing information of an instance of OEOP(S).

Example: Let A, B, C and D be functions satisfying the simple sharing assumption (Figure 5). Let the evaluation order be B, D, C, and A. Then:

$$\text{SPILL}(B, C, D, A) = \text{SPILL}_B + \text{SPILL}_C + \text{SPILL}_D + \text{SPILL}_A = 13.$$

We shall convert this arc weighted symmetric digraph G to a nonnegative arc weighted complete digraph $G_n = (V_n, A_n, w)$ by adding new arcs to G with zero arc weights.

Lemma 1. *The number of spills in the complete graph G_n due to the evaluation order $F_{\pi_1}, F_{\pi_2}, \ldots, F_{\pi_n}$ is same as the number of spills in G (the n function calls with sharing as represented by G) under the same evaluation order.* □

For computing the total number of spills for the order $F_{\pi_1}, F_{\pi_2}, \ldots, F_{\pi_n}$; the weights of the arcs of the form (π_i, π_j) with $j > i$ are considered in deciding on the number of spills. It is easy to see that these arcs corresponding to the above evaluation order form an acyclic tournament on V_n in G_n [7]. This we shall refer to as the acyclic tournament $T_\pi = \{(\pi_i, \pi_j) | i < j\}$ on V_n in G_n represented by the permutation π. A permutation of nodes of V_n defines a unique acyclic tournament on V_n and *vice versa*. In conclusion, for obtaining the optimal evaluation order in relation to the graph G, it would be enough if we obtain an

acyclic tournament on V_n in $G_n = (V_n, A_n, w)$ with minimum total arc weight. This problem is known as MINLOP and it is known to be NP-complete [6]. MINLOP_{12} is same as MINLOP with only difference that weight of an arc is either 1 or 2. We will show that MINLOP_{12} is NP-complete by reducing it from MINFAS. In MINFAS, for a given digraph $G = (V_n, A)$ we are asked to find a subset B of A with minimum cardinality such that $(V_n, A - B)$ is acyclic. MINFAS is known to be NP-complete [5].

Lemma 2. *Let $G = (V_n, A)$ be a digraph and $B \subseteq A$ be a FAS (feedback arc set) of G. Then there exists a $C \subseteq B$ such that (V_n, C) is acyclic and C is a FAS of G.*

Proof If (V_n, B) is acyclic then $C = B$. Otherwise, (V_n, B) has at least one directed cycle. Since B is a FAS, $(V_n, A-B)$ is acyclic. Now construct an acyclic tournament (V_n, T) of G_n with $(A - B) \subseteq T$. We define $C = T^c \cap B$ where $T^c = \{(j, i) | (i, j) \in T\}$. C is FAS of G because $A - C = A \cap T$ and $A \cap T$ is acyclic (see Figure 6). □

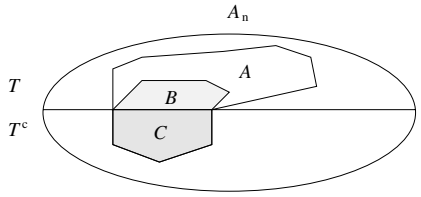

Fig. 6. $C \subseteq B \subseteq A$ and $T \cup T^c = A_n$

Given an FAS B of G the set C in the Lemma 2 can be constructed in polynomial time as follows. Let $(v_1, v_2, \ldots, v_n)$ be a permutation of V_n for which v_1 is a node in $(V_n, A - B)$ with zero indegree, v_2 is a node with zero indegree in the acyclic subdigraph of $(V_n, A - B)$ induced by the node set $V_n - \{v_1\}$ and so on. Note that $v_1, v_2, \ldots, v_n$ can be chosen in polynomial time. Let $T = \{(v_i, v_j) | i < j\}$. Clearly $A - B \subseteq T$ and T is an acyclic tournament on V_n. $C = T^c \cap B$ can be constructed in polynomial time. In conclusion, every FAS induces another FAS which is acyclic. *From now on, by FAS we will mean this induced acyclic FAS.*

Lemma 3. *The no. of arcs in an acyclic tournament on V_n in G_n is $\frac{1}{2}n(n-1)$.*

Theorem 1. *MINLOP_{12} is NP-complete.*

Proof - MINLOP_{12} is in NP since MINLOP is. To show that MINLOP_{12} is NP-hard we shall reduce MINFAS to MINLOP_{12}.

Let $G = (V_n, A)$ be an instance of MINFAS. From this we shall construct an instance $G_n = (V_n, A_n, w)$ of MINLOP_{12} with $w(e) = 1$ if $e \notin A$ and $w(e) = 2$ if $e \in A$. Let (V_n, T) be an acyclic tournament of G_n. From the proof of Lemma

2, it can be easily seen that $T \cap A$ is a FAS of G. By Lemma 3 and from the definition of the weight function w it follows that $w(T) = \frac{1}{2}n(n-1) + |T \cap A|$.

Next we shall show that, B is a FAS of G with $|B| \leq k$ if and only if there exists an acyclic tournament (V_n, T_B) of G_n with $w(T_B) \leq \frac{1}{2}n(n-1) + k$.
$\Leftarrow$: Let (V_n, T_B) be an acyclic tournament of G_n with $w(T_B) \leq \frac{1}{2}n(n-1) + k$.

$$w(T_B) = \frac{1}{2}n(n-1) + |T_B \cap A| \leq \frac{1}{2}n(n-1) + k$$
$$i.e.\ |T_B \cap A| \leq k$$

Choose $B = T_B \cap A$. Since B is an FAS of G we are done.
$\Rightarrow$: Let B be a FAS with $|B| \leq k$. By Lemma 2 (V_n, B) is acyclic. From definition of feedback arc set, $(V_n, A - B)$ is acyclic. Let (V_n, T) be an acyclic tournament with $(A - B) \subseteq T$. This arc set T can be constructed in polynomial time. Define $T_B = T^c$. Clearly $T_B \cap B \subseteq B$ and $(T_B \cap B) = (T_B \cap A)$.

$$\text{i.e.} \quad w(T_B) = \frac{1}{2}n(n-1) + |T_B \cap A|$$
$$= \frac{1}{2}n(n-1) + |T_B \cap B| \ \leq \ \frac{1}{2}n(n-1) + k.$$

Theorem 2. *OEOP(S) is NP-hard.*

Proof sketch: We will reduce MINLOP_{12} to OEOP(S). Let $G_n = (V_n, A_n, w)$ be an instance (call it X) of MINLOP_{12}, where $A_n = \{(i,j) | i \neq j \text{ and } i, j \in V_n\}$. Let $V_n = \{1, 2, \ldots, n\}$. Construct an instance of OEOP(S) from X. The subdigraph induced by any pair of nodes i and j will into one of:
Case 1: $w(i,j) = w(j,i) = 1$; **Case 2:** $w(i,j) = w(j,i) = 2$ and
Case 3: either $w(i,j) = 1$ and $w(j,i) = 2$ or $w(i,j) = 2$ and $w(j,i) = 1$.
For Case 1, we include the subexpressions $t_k^1 = a_k * (b_k + c_k)$ and $t_k^2 = (b_k + c_k) + d_k$ in the functions F_i and F_j respectively. It can be seen easily that if we evaluate t_k^1 and t_k^2 in any order only one memory spill will be required and it fits with the arc weights. Similarly the expressions t_k^1 and t_k^2 are chosen for Case 2 and Case 3. The idea is that the weights between nodes i and j in X fall into one of the three cases. The corresponding expressions will be linked to the arguments of the function calls F_i and F_j such that the simple sharing assumption is not violated. Doing this for all pair of nodes in the instance X, we can obtain Y, the instance of OEOP(S). Y will have function calls F_1 till F_n and their arguments will be shared. And then it can be established that for any permutation $\pi = (\pi_1, \pi_2, \ldots \pi_n)$ of $(1, 2, \ldots n)$, $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w(F_{\pi_i}, F_{\pi_j})$ is same as the number of spills that will be made if the functions of Y are evaluated according to the same permutation. □

5 Heuristics for OEOP

The amount of sharing between the arguments of functions in user programs is usually not high. However, such programs, for optimization purposes, are

transformed into equivalent programs. Function unfolding is one such important transformation technique, and this technique introduces sharing [4]. Our observation is that the number of function call nodes at the leaf-level of an expression DAG (after unfolding) does not go beyond 8 or 9, and further, the number of shared nodes between the argument DAGs of two functions also hardly goes beyond 8 or 9. So, the graph that we will construct will not have nodes more than 9 and further the weights of the edges in it will be within 0 to 9. With such restrictions, the following greedy heuristic will give satisfactory results.

Construct the weighted graph from the DAG of the linear region. Then find the node such that the sum of the outgoing arc weights for this node is minimum. Evaluate the corresponding call. Then the call is removed from the linear region and it is replaced with a memory location. Get the new weighted graph, and repeat the process till a single call remains. This is the last call to be evaluated.

6 Conclusion

We have discussed an optimization problem that occurs while following a compilation technique for eager functional languages. We have shown that the problem is NP–Complete. The context in which problem arises, keeps the dimension of the problem small. Our experiments show that greedy heuristics provide satisfactory results.

Acknowledgements: The authors would like to thank A. Sanyal, A. Diwan and C.R. Subramanian for useful discussions.

References

1. Aho A.V., Johnson S.C. & Ullman J.D., Code generation for expressions with common subexpressions, JACM, Vol. 24(1), January 1977, pp. 146-160.
2. Aho A.V., Sethi R. & Ullman J.D.,*Compilers : Principles, Techniques and Tools*, Addison Wesley, 1986.
3. Bird R. & Wadler P., *Introduction to Functional Programming*, Printice Hall, 1988.
4. Davidson J.W. & Holler A.M., Subprogram Inlining: A study of its effect on program execution time, IEEE Tr. on *Software Engineering*, Vol. 18(2), Feb. 1992, pp. 89-102.
5. Garey M.R. & Johnson D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman and Company, New York, 1979.
6. Grotschel M., Junger M. & Reinelt G., On the acyclic subgraph polytope, *Math. Programming* 33 (1985), pp. 28-42.
7. Harary F., *Graph Theory*, Addition-Wesley, Reading, MA, 1969.
8. Satpathy M., Sanyal A & Venkatesh G., Improved Register Usage of Functional Programs through multiple function versions, *Journal of Functional and Logic Programming*, December 1998, MIT Press.