RWTH Aachen

Department of Computer Science
Technical Report

Call Tree Reversal is NP-Complete

Uwe Naumann

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-18

RWTH Aachen · Department of Computer Science · December 2007

Call Tree Reversal is NP-Complete

Uwe Naumann

LuFG Informatik 12, Department of Computer Science, RWTH Aachen University, Aachen, Germany,
naumann@stce.rwth-aachen.de

Abstract. The data-flow of a numerical program is reversed in its adjoint. We discuss the combinatorial optimization problem that aims to find optimal checkpointing schemes at the level of call trees. For a given amount of persistent memory the objective is to store selected arguments and/or results of subroutine calls such that the overall computational effort (the total number of floating-point operations performed by potentially repeated forward evaluations of the program) of the data-flow reversal is minimized. CALL TREE REVERSAL is shown to be NP-complete.

1 Background

We consider implementations of multi-variate vector functions $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ as computer programs $\mathbf{y} = F(\mathbf{x})$. The interpretation of reverse mode automatic differentiation (AD) [8] as a semantic source code transformation performed by a compiler yields an adjoint code $\bar{\mathbf{x}} += \bar{F}(\mathbf{x}, \bar{\mathbf{y}})$. For given $\mathbf{x}$ and $\bar{\mathbf{y}}$ the vector $\bar{\mathbf{x}}$ is incremented with $(F'(\mathbf{x}))^T \cdot \bar{\mathbf{y}}$ where $F'(\mathbf{x})$ denotes the Jacobian matrix of F at $\mathbf{x}$. Adjoint codes are of particular interest for the evaluation of large gradients as the complexity of the adjoint computation is independent of the gradient's size. Refer to [1–4] for an impressive collection of applications where adjoint codes are instrumental to making the transition form pure numerical simulation to optimization of model parameters or even of the model itself.

In this paper we propose an extension to the notion of joint call tree reversal [8] with the potential storage of the results of a subroutine call. We consider call trees as runtime representations of the interprocedural flow of control of a program. Each node in a call tree corresponds uniquely to a subroutine call.[1] We assume that no checkpointing is performed at the intraprocedural level, that is, a "store-all" strategy is employed inside all subroutines. A graphical notation for call tree reversal under the said constraints is proposed in Figure 1. A given subroutine can be executed without modifications ("advance") or in an augmented form where all values that are required for the evaluation of its adjoint are stored (taped) on appropriately typed stacks ("tape (store all)"). We refer to this memory as the *tape* associated with a subroutine call, not to be confused with the kind of tape as generated by AD-tools that use operator overloading such as ADOL-C [9] or variants of the differentiation-enabled NAGWare Fortran compiler [14]. The arguments of a subroutine call can be stored ("store arguments") and restored ("restore arguments"). Results of a subroutine call can be treated similarly ("store results" and "restore results"). The adjoint propagation yields the reversed data-flow due to popping the previously pushed values from the corresponding stacks ("reverse (store all)"). Subroutines that only call other subroutines without performing any local computation are represented by "dummy calls." For example, such wrappers can be used to visualize arbitrary checkpointing schemes for time evolutions (implemented as loops whose body

[1] Generalizations may introduce nodes for various parts of the program, thus yielding arbitrary checkpointing schemes.

is wrapped into a subroutine). Moreover they occur in the reduction used for proving CALL TREE REVERSAL to be NP-complete. Dummy calls can be performed in any of the other seven modes.

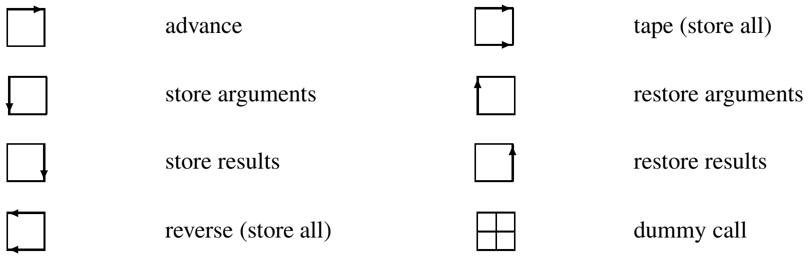

Fig. 1. Calling modes for interprocedural data-flow reversal.

Figure 2 illustrates the reversal in split (b), classical joint (c), and joint with result checkpointing (d) modes for the call tree in (a). The order of the calls is from left to right and depth-first.

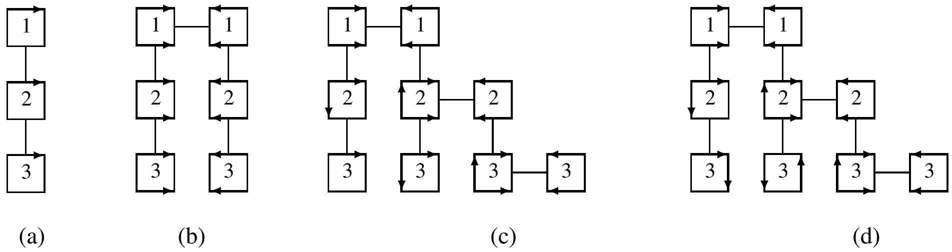

Fig. 2. Interprocedural data-flow reversal modes: Original call tree (a), split reversal (b), joint reversal with argument checkpointing (c), joint reversal with result checkpointing (d).

For the purpose of conceptual illustration we assume that the sizes of the tapes of all three subroutine calls in Figure 2 (a) as well as the corresponding computational complexities are identically equal to 2 (memory units / floating-point operation (flop) units). The respective calls are assumed to occur in the middle, e.g. the tape associated with the statements performed by subroutine 1 prior to the call of subroutine 2 has size 1. Consequently the remainder of the tape has the same size. One flop unit is performed prior to a subroutine call which is followed by another unit. The size of argument and result checkpoints is assumed to be considerably smaller than that of the tapes. Refer also to footnotes 2 and 3.

Split call tree reversal minimizes the number of flops performed by the forward calculation (6 flop units). However an image of the entire program execution (6 memory units) needs to fit into persistent memory which is infeasible for most relevant problems. This shortcoming is addressed by classical joint reversal (based solely on argument checkpointing). The maximum amount of persistent memory needed is reduced to 4 (half of subroutine 1 plus half of subroutine 2 plus subroutine 3)[2] at the cost of additional

[2]...provided that the size of an argument checkpoint of subroutine 3 is less than or equal to one memory unit, i.e. $\texttt{sizeof}(argchp_3) \leq 1$, and that $\texttt{sizeof}(argchp_2) \leq 2$.

6 flop units (a total of 12 flop units is performed). This number can be reduced to 10 flop units (while the maximum memory requirement remains unchanged[3]) by storing the result of subroutine 3 and using it for taping subroutine 2 in Figure 2 (d). The impact of these savings grows with the depth of the call tree.

It is trivial to design toy problems that illustrate this effect impressively. An example can be found in the appendix. The computation of the partial derivative of y with respect to x as arguments of the top-level routine f0 in adjoint mode requires the reversal of a call tree (a simple chain in this case) of depth five. The leaf routine f5 is computationally much more expensive than the others. Classical joint reversal takes about 0.6 seconds whereas additional result checkpointing reduces the runtime to 0.25 seconds. These results were obtained on a state-of-the-art Intel PC. The full code can be obtained by sending an email to the author. The use of result checkpointing in software tools for AD such as Tapenade [12], OpenAD [15], or the differentiation-enabled NAGWare Fortran compiler [14] is the subject of ongoing research and development.

Finding an optimal (or at least near-optimal) distribution of the checkpoints or, equivalently, corresponding combinations of split and joint (with argument checkpointing) reversal applied to subgraphs of the call tree has been an open problem for many years. In this paper we show that a generalization of this problem that allows for subsets of subroutine arguments and/or results to be taped is NP-complete. Hence, we believe that the likelihood of an efficient exact solution of this problem is low. Heuristics for finding good reversal schemes are currently being developed in collaboration with colleagues at INRIA, France, and at Argonne National Laboratory, USA.

2 Data-Flow Reversal is NP-Complete

The program that implements F should decompose into a straight-line evaluation procedure

$$v_j = \varphi_j(v_i)_{i \prec j} \tag{1}$$

for $j = 1, \ldots, q$. We follow the notation in [8]. Hence, $i \prec j$ denotes a direct dependence of v_j on v_i. Equation (1) induces a directed acyclic graph (DAG) $G = (V, E)$ where $V = \{1-n, \ldots, q\}$ and $(i, j) \in E \Leftrightarrow i \prec j$. We consider independent (without predecessors), intermediate, and dependent (without successors) vertices. Without loss of generality, the m results are assumed to be represented by the dependent vertices. We set $p = q - m$. An example is shown in Figure 3 (a) representing, e.g.,

$$x_0 = x_0 \cdot \sin(x_0 \cdot x_1);\; x_1 = x_0/x_1;\; x_0 = \cos(x_0);\; x_0 = \sin(x_0);\; x_1 = \cos(x_1) \quad . \tag{2}$$

A representation as in Equation (1) is obtained easily by mapping the physical memory space (x_0, x_1) onto the single-assignment memory space $(v_{-1}, \ldots, v_7)$.

The problem faced by all developers of adjoint code compiler technology is to generate the code such that for a given amount of persistent memory the values required for a correct evaluation of the adjoints can be recovered efficiently by combinations of storing and recomputing [6, 10, 11]. Load and store costs (both ≥ 0) are associated with single read and write accesses to the persistent memory, respectively. Floating-point operations have nontrivial cost > 0. The program's physical memory $\mathbf{p} = (p_1, \ldots, p_\mu)$ is

[3]...provided that $\texttt{sizeof}(argchp_2) + \texttt{sizeof}(reschp_3) \leq 2$ and $\texttt{sizeof}(argchp_3) + \texttt{sizeof}(reschp_3) \leq 2$, where $\texttt{sizeof}(reschp_i)$ denotes the size of a result checkpoint of subroutine i (in memory units).

considered to be nonpersistent, i.e. one does not count on any of the p_i holding useful values except right after their computation.

A *data-flow reversal* is an algorithm that makes the values of the intermediate variables of a given program run (equivalently, its DAG) available in reverse order.

In [13] we propose a proof for the NP-completeness of the DAG REVERSAL problem. The argument is based on the assumption that writing to persistent memory as well as performing a floating-point operation have both unit cost while the load cost vanishes identically, e.g. due to prefetching. This special case turns out to be computationally hard. Hence, the general case cannot be easier. However, there are other special cases for which efficient algorithms do exist [7].

If the size of the available memory is equal to $n+p$, then a store-all (last-in-first-out) strategy recovers the p intermediate values of a DAG in reverse order at optimal cost $n+p$ (store operations) – a sharp lower bound for the solution of the DAG REVERSAL problem under the made assumptions. The values of the m results are assumed to be available at the end of the single function evaluation that is required in any case. One can now ask for a reversal scheme (assignment of vertices in the DAG to persistent memory) where the memory consumption is minimized while the total cost remains equal to $n+p$. A formal statement of this FIXED COST DAG REVERSAL (FCDR) problem is given in Section 2.1. It turns out that FCDR is equivalent to VERTEX COVER [5] on the subgraph induced by the intermediate vertices. The values of the independent vertices need to be stored in any case as there is no way to recompute them.

Example Consider the DAG in Figure 3 (a) for an intuitive illustration of the idea behind the proof in [13]. A store-all strategy requires a persistent memory of size seven.

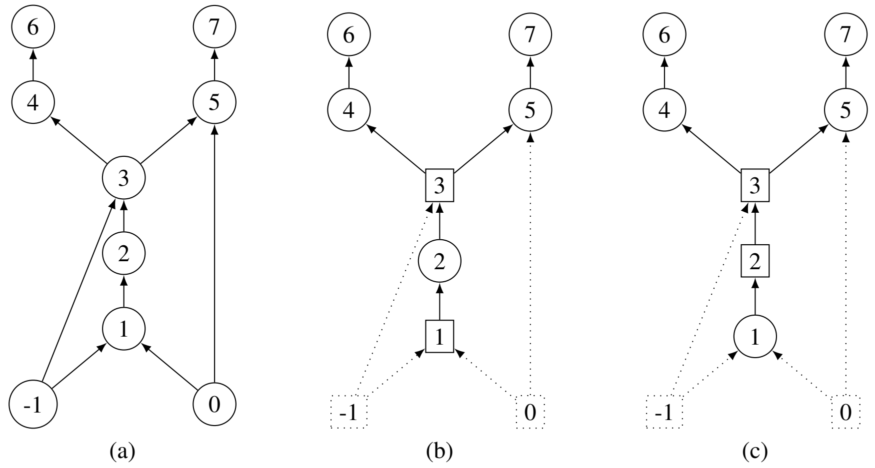

Fig. 3. DAG and minimal vertex covers (rectangular nodes) restricted to the intermediate vertices.

Alternatively, after storing the two independent values the five intermediate values can be recovered from stored v_1 and v_3 as in Figure 3 (b) (similarly v_2 and v_3 in Figure 3 (c)). Known values of v_0 and v_3 allow us to recompute v_4 and v_5 at a total cost of two flops. The value of v_2 can be recomputed from v_1 at the cost of a single flop making the overall cost add up to seven. Both $\{1,3\}$ and $\{2,3\}$ are minimal vertex covers in the graph spanned by vertices $1,\ldots,5$.

FCDR is not the problem that we are actually interested in. Proving FCDR to be hard is simply a vehicle for studying the computational complexity of the relevant DAG REVERSAL (DAGR) problem. It turns out that a given algorithm for DAGR can be used to solve FCDR. In conclusion DAGR must be at least as hard as FCDR.

2.1 FIXED COST DAG REVERSAL

Given are a DAG G and an integer $n \leq K \leq n+p$. Is there a data-flow reversal with cost $n+p$ that uses $k \leq K$ memory units?

Theorem 1. *FCDR is NP-complete.*

Proof. The proof is by reduction from VERTEX COVER as described in [13]. ■

2.2 DAG REVERSAL

Given are a DAG G and integers K and C such that $n \leq K \leq n+p$ and $K \leq C$. Is there a data-flow reversal that uses at most K memory units and costs $c \leq C$?

Theorem 2. *DAGR is NP-complete.*

Proof. The idea behind the proof in [13] is the following.

An algorithm for DAGR can be used to solve FCDR as follows: For $K = n+p$ "store-all" is a solution of DAGR for $C = n+p$. Now decrease K by one at a time as long as there is a solution of FCDR for $C = n+p$. Obviously, the smallest K for which such a solution exists is the solution of the minimization version of FCDR. A given solution is trivially verified in polynomial time by counting the number of flops performed by the respective code. ■

3 Call Tree Reversal is NP-Complete

An *interprocedural data-flow reversal* for a program run (or, equivalently, for its DAG) is a data-flow reversal that stores only subsets of the inputs or outputs of certain subroutine calls while recomputing the other values from the stored ones.

A *subroutine result checkpointing scheme* is an interprocedural data-flow reversal for the corresponding DAG which recovers all intermediate values in reverse order by storing only subsets of outputs of certain subroutines and by recomputing the other values from the stored ones. It can be regarded as a special case of DAGR where the values that are allowed to be stored are restricted to the results computed by the performed subroutine calls.

RESULT CHECKPOINTING *(RC) Problem:* Given are a DAG G and a call tree T of a program run and integers K and C such that $n \leq K \leq n+p$ and $K \leq C$. Is there a subroutine result checkpointing scheme that uses at most K memory units and that performs $c \leq C$ flops?

Theorem 3. *RC is NP-complete.*

```
program main

  real p(3)

  call f1(); call f2(); call f3(); call f4();
  call f5(); call f6(); call f7();

contains

  subroutine f1()p
    p(3)=p(1)*p(2)
  end subroutine f1

  subroutine f2()
    p(3)=sin(p(3))
  end subroutine f2

  subroutine f3()
    p(3)=p(1)*p(3)
  end subroutine f3

  subroutine f4()
    p(1)=cos(p(3))
  end subroutine f4

  subroutine f5()
    p(2)=p(3)/p(2)
  end subroutine f5

  subroutine f6()
    p(1)=sin(p(1))
  end subroutine f6

  subroutine f7()
    p(2)=cos(p(2))
  end subroutine f7

end
```

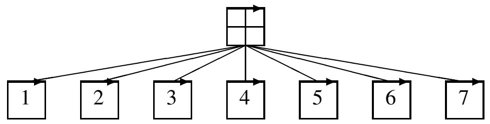

Fig. 4. Reduction from DAG REVERSAL to RESULT CHECKPOINTING and Call Tree

Proof. The proof constructs a bijection between RC and DAGR. Consider an arbitrary DAG as in DAGR. Let all intermediate and maximal vertices represent calls to multivariate scalar functions f_i, $i = 1, \ldots, q$, operating on a global memory space $\mathbf{p} \in I\!R^{\mu}$. The f_i are assumed to encapsulate the φ_i from Equation (1). Hence, the local tapes are empty since the single output is computed without evaluation of intermediate values directly from the inputs of f_i. Any given instance of DAGR can thus be mapped uniquely to an instance of RC and vice versa. A solution for DAGR can be obtained by solving the corresponding RC problem. Therefore RC must be at least as hard as DAGR. A given solution to RC is trivially verified in polynomial time by counting the number of flops performed. ■

Example To obtain the graph in Figure 3 (a) we require seven subroutines operating on a nonpersistent global memory of size three and called in sequence by the main program as shown in Figure 4. The tapes of all subroutines are empty. Hence, the cost function is composed of the costs of executing the subroutines for a given set of inputs (unit cost per subroutine) in addition to the cost of generating the required result checkpoints (unit cost per checkpoint). The values $v_1, \ldots, v_5$ need to be restored in reverse order. The input values v_{-1} and v_0 are stored in any case.

With a stack of size seven at our disposal a (result-)checkpoint-all strategy solves the FCDR problem. The same optimal cost can be achieved with a stack of size four. For example, checkpoint the results of calling f1 and f3 and recompute v_5 as a function of v_3 and v_0, v_4 as a function of v_3, and v_2 as a function of v_1. We note that $\{1,3\}$ is a vertex cover in the subgraph of G spanned by its intermediate vertices whereas any single vertex is not.

CALL TREE REVERSAL *(CTR) Problem:* Given are a DAG G and a call tree T of a program run and integers K and C such that $n \leq K \leq n + p$ and $K \leq C$. Is there an interprocedural data-flow reversal for G that uses at most K memory units and that performs $c \leq C$ flops?

Theorem 4. CTR *is NP-complete.*

Proof. With the reduction used in the proof of Theorem 3 any interprocedural data-flow reversal is equivalent to a subroutine result checkpointing scheme. All relevant subroutine arguments are outputs of other subroutines. ■

The key prerequisite for the above argument is the relaxation of argument checkpointing to subsets of the subroutine inputs.

4 Conclusion

NP-completeness proofs for problems that have been targeted with heuristics for some time can be regarded as late justification for such an approach. The algorithmic impact should not be overestimated unless the proof technique yields ideas for the design of new (better) heuristics and/or approximation algorithms. The evaluation of our paper's contribution from this perspective is still outstanding. Work on robust and efficient heuristics, in particular for interprocedural data-flow reversals that involve result checkpointing, has only just started.

Adjoint codes do not necessarily use the values of the variables in Equation (1) in strictly reverse order. For example, the adjoint of Equation (2) uses the value of v_{-1} prior to that of v_1. In order to establish the link between strict data-flow reversal and adjoint codes one needs to construct numerical programs whose adjoints exhibit a suitable data access pattern. This is done in [13].

Compiler-based code generation needs to be conservative. It is based on some sort of call graph possibly resulting in different call trees for varying values of the program's inputs. Such call trees do not exists at compile time. The solutions to a generally undecidable problem yield a computationally hard problem. Developers of adjoint compiler technology will have to deal with this additional complication.

References

1. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series. SIAM, 1996.
2. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, number 50 in Lecture Notes in Computational Science and Engineering, Berlin, 2005. Springer.
3. G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*. Springer, 2002.
4. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series. SIAM, 1991.
5. M. Garey and D. Johnson. *Computers and Intractability - A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.
6. R. Giering and T. Kaminski. Recomputations in reverse mode AD. In *[3]*, chapter 33, pages 283–291. 2001.
7. A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.
8. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*. SIAM, Apr. 2000.
9. A. Griewank, D. Juedes, and J. Utke. ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.
10. L. Hascoët and M. Araya-Polo. The adjoint data-flow analyses: Formalization, properties, and applications. In *[2]*, pages 135–146. Springer, 2005.
11. L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.
12. L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.
13. U. Naumann. On optimal DAG reversal. Technical Report AIB-2007-05, RWTH Aachen, 2007. Submitted. The preprint can be downloaded from the CS department's AIB site `http://www.cs.rwth-aachen.de/Forschung/aib.php`.
14. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, 2005.
15. J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4), 2008. To appear.

A Reference Code for Result Checkpointing

```
subroutine f0(x,y)
  double precision x,y
  call f1(x,y)
  y=sin(y)
end subroutine f0

subroutine f1(x,y)
  double precision x,y
  call f2(x,y)
  y=sin(y)
end subroutine f1

subroutine f2(x,y)
  double precision x,y
  call f3(x,y)
  y=sin(y)
end subroutine f2

subroutine f3(x,y)
  double precision x,y
  call f4(x,y)
  y=sin(y)
end subroutine f3

subroutine f4(x,y)
  double precision x,y
  call f5(x,y)
  y=sin(y)
end subroutine f4

subroutine f5(x,y)
  double precision x,y
  integer i
  y=0
  do 10 i=1,10000000
    y=y+x
10  continue
end subroutine f5
```

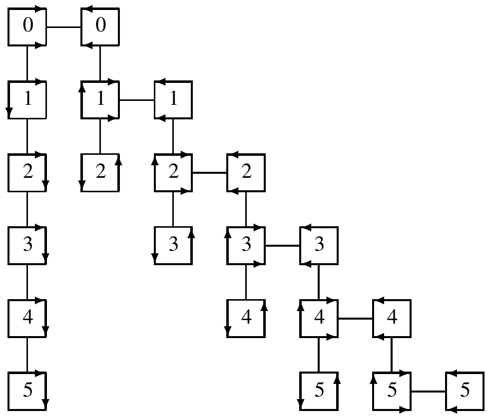

Fig. 5. Subroutine result checkpointing scheme for the reference code: Each subroutine is executed twice ("advance" and "tape (store all)" once, respectively) instead of $d+1$ times where d is the depth in the call tree (starting with zero). Additional persistent memory is needed to store the results of all subroutine calls. The maximum amount of persistent memory required by the adjoint code may not be affected as illustrated by the example in Figure 2.