

# The Recompute-Or-Store Alternative in reverse Automatic Differentiation

Laurent Hascoët

INRIA Sophia-Antipolis, TROPICS team

## 1 Introduction

The reverse mode of Automatic Differentiation (AD) is an appealing strategy to compute gradients, which suffers unfortunately from its utilization of intermediate values of the original program in the *reverse* of their computation order. AD tools use a number of strategies to cope with that. There are coarse-grain strategies based on *checkpointing*, most often *nested*, with variants such as *reverse* checkpointing. There are fine-grain strategies, dealing with a single subroutine or with a single block of instructions, that make intermediate values available in reverse order either (*Store-All* or SA) by storing them into additional variables or external storage, or (*Recompute-All* or RA) by recomputing them repeatedly from a chosen point using instructions copy. The coarse and fine-grain strategies are generally combined to produce an acceptable adjoint code.

All these strategies have a cost e.g. extra memory consumption, extra run time, or both. Apparently, no strategy is best on all application programs. We are convinced that for each application program, there is an optimal strategy that depends on the application, and this strategy is probably none of the above, but a combination of them. However, there is no framework to specify, compare, and optimize combinations of the RA and SA strategy.

This paper presents an attempt to such a common framework, that we call *ROSA*, standing for *Recompute-Or-Store Alternative*. Since the difficulty of restoring previous intermediate values comes from program overwriting variables, and since overwriting is well captured by data-dependence graphs, our idea is to represent the restoration algorithm and the associated choices on this data-dependence graph.

The next section 2 presents the ROSA framework. Section 3 shows how the pure RA and SA strategies translate into extreme cases of ROSA, and how we could come closer to the optimal ROSA strategy, probably with heuristics expressed on the ROSA framework. Section 4 concludes on the possible extensions of ROSA, and on the link with the coarse-grain strategies based on checkpointing.

## 2 The ROSA framework

We consider a piece of program, which can be a Basic Block of instructions for a simple case, but which can be more general. Suppose we split this Block into

atomic nodes, e.g. instructions. For each node  $I_n$ , we define a differentiated node  $I'_n$  which takes as inputs (a subset of) the inputs of  $I_n$ . Since we must execute the differentiated nodes  $I'_n$  in the reverse order, the problem is how to make available (a subset of) the inputs of the  $I_n$  in the reverse order as well.

We build the data-dependence graph (DDG) [1] whose nodes are the  $I_n$ , and whose arrows are the classical *true* (write-to-read), *anti* (read-to-override), and *output* (write-to-override) dependences. We define a notion of state. The state is indeed the mapping from variables to values, but we rather define it as the nodes that produce these values. We call these nodes “*exposed*”. Figure 1 shows an example DDG, in the state after running  $I_1$  through  $I_7$ . Only true dependences are drawn for clarity. Instructions that overwrite the same variable  $v_j$  are on the same dotted line, and are therefore linked by an output dependence, not shown. The anti dependences, not shown, can be deduced easily. Nodes  $I_3$ ,  $I_5$ ,  $I_6$ , and  $I_7$  are exposed (shown with back dots). Nodes  $I_1$ ,  $I_2$ , and  $I_4$  are not exposed any more because of subsequent overwriting. To execute an instruction

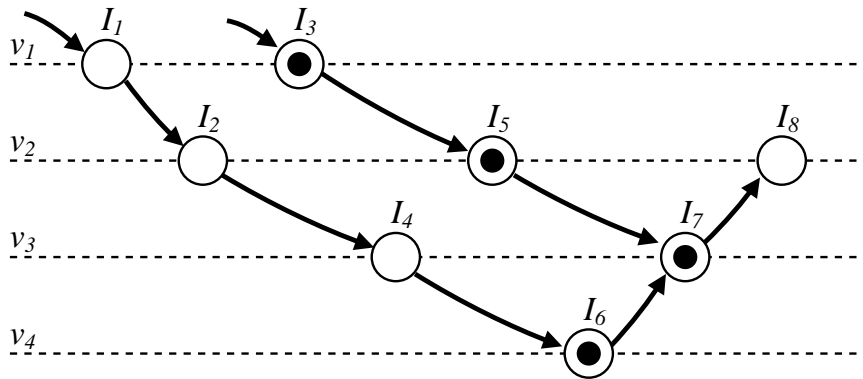


Figure 1: Exposed and invertible nodes in the ROSA framework

$I_n$  requires that the origins of all true dependences reaching  $I_n$  are exposed. The same requirement holds to run  $I'_n$ , at least for a subset of the dependences. In this case, we will say that node  $I_n$  is “*invertible*”. To keep things simple, we will suppose in our example that each  $I'_n$  uses all the inputs of  $I_n$ . On figure 1, node  $I_8$  is invertible, and so is  $I_7$ . But the displayed state does not make  $I_6$  invertible because  $I_4$  needs to be exposed. There are essentially two tactics to expose a node:

- **Store:** the value computed by the node was saved before it was overwritten, so that we can restore it.
- **Recompute:** first, all origins of incoming true dependences must be exposed and if not, the exposition mechanism must be called recursively. Second, we run again the instruction in the node.

Thus one ROSA strategy consists in choosing, for each output dependence, which tactic will be taken to expose the origin node when needed. The Store tactic has limited consequences on the current state. It just moves one black dot backwards. However, it uses memory, and saving and restoring may take time. The Recompute strategy may recursively change the state on many preceding nodes. It also costs extra execution of instructions, but uses no memory. On figure 1, the Recompute strategy to expose  $I_4$  would run  $I_1$ ,  $I_2$ , and  $I_4$ , exposing the three of them.  $I_3$ ,  $I_5$ , and  $I_7$  would not be exposed any more. This has a positive consequence for  $I_6$ , which becomes invertible, as well as  $I_2$  and  $I_4$ . On the other hand there are negative consequences for  $I_5$ , which becomes not invertible. On our example, maybe a better strategy would be to use “Store” on the  $I_4 \rightarrow I_7$  output dependence, and “Recompute” on  $I_2 \rightarrow I_5$ .

### 3 Evaluating and comparing ROSA strategies

The ROSA framework can capture both the RA and SA strategies. If the “Store” tactic is chosen for each output dependence, then we end up with the SA strategy. Storage is done only if the origin  $I_o$  of the output dependence needs to be exposed, i.e. when the derivative of some node actually needs the output of  $I_o$ . This is the goal of the TBR analysis in TAPENADE [4]. If the “Recompute” tactic is chosen all the time, we end up with the RA strategy. Using the DDG ensures that only necessary instructions are run again, and only if their output is not already exposed. This is nothing but an equivalent DDG-based description of the extended ERA algorithm in TAF [6]. However in the general case, the optimal strategy is neither RA nor SA.

Finding the cost of one given ROSA strategy is simply a matter of counting. Therefore, finding the optimal strategy is possible theoretically. In the general case, the optimal ROSA strategy depends on the shape of the DDG. This is a combinatorial problem since choosing some tactic for one output dependence may change the relative cost of the alternatives for another output dependence. For example choosing “Store” for dependence  $I_2 \rightarrow I_5$  changes the costs of the “Recompute” alternative on  $I_2 \rightarrow I_5$ , because running  $I_1$  is not necessary any more to expose  $I_2$ . Presumably, finding the global optimal strategy is out of reach for large graphs.

However, we can devise heuristics. We can start from a SA strategy and switch some dependences to “Recompute” when the benefit is high. Symmetrically, we can start from a RA strategy. We may also consider each output dependence  $I_b \rightarrow I_d$  separately, along with the neighbor dependences as shown on figure 2. Exposing  $I_b$  will be necessary when the rightmost node that depends on it is differentiated. On the figure, we can evaluate the costs of “Store” and “Recompute”, but only approximately because global effects are neglected. The “Store” tactic on  $I_b \rightarrow I_d$  has a fixed cost: it just uses one memory cell, and the black dot moves from  $I_d$  back to  $I_b$  as wanted. The “Recompute” tactic costs one execution of  $I_b$ , plus recursively the cost of exposing all required nodes such as  $I_a$ . If this moves the exposition (black dot) from  $I_{c2}$  to  $I_a$ , the

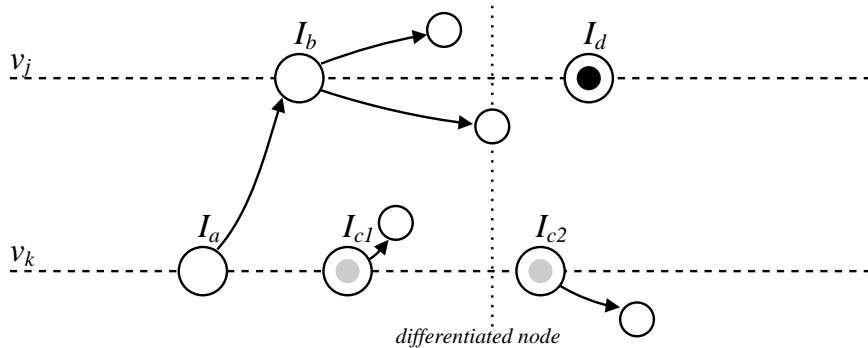


Figure 2: Local analysis of one output dependence

cost is decreased because exposition of  $I_{c2}$  was not useful any more. Conversely if exposition moves from  $I_{c1}$  to  $I_a$ , the cost is increased because  $I_{c1}$  could still be required by derivative instructions to come.

One typical example where “Recompute” is the best choice is for instructions that precompute useful values from constants. For example at the beginning of gather-scatter loops on unstructured meshes, instructions compute useful array indices from constant indirection arrays. If we suppose that  $I_a$  is such an instruction, we realize that the next overwrite  $I_c$  is very far away, probably at next loop cycle, so  $I_c$  is more like  $I_{c2}$ . Moreover, running  $I_a$  twice is very cheap (cheaper than storing), and requires no exposed node. The proposed heuristic captures this situation and chooses “Recompute”.

## 4 Conclusion

We propose a framework based on the data-dependence graph, on which we can define and evaluate reversal strategies that combine storage and recomputation of intermediate variables. We explore several methods to get as close as possible to the optimal strategy. This is a contribution to solving the old argument between Recompute-All and Store-All approaches in the reverse mode of AD tools. ROSA strategies blend with coarse-grain checkpointing, just as well as RA or SA strategies.

This framework can be extended in several ways. For example we didn’t consider arrays, that introduce more complexity in the output dependences. Also, the order of the differentiated instructions is not necessarily the reverse of the original instructions’ order. A more flexible framework could use the data-dependence graph to explore reorderings of the differentiated instructions, which in turn could need fewer recomputation or storage.

## References

- [1] J. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.
- [2] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.
- [3] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann (Editors). *Automatic Differentiation of Algorithms, from Simulation to Optimization*. LNCSE. Springer, 2002. selected papers from the AD2000 conference, Nice, France.
- [4] C. Faure and U. Naumann. The taping problem in automatic differentiation. In [3] *Automatic Differentiation of Algorithms, from Simulation to Optimization*, pages 293–298, 2002.
- [5] R. Giering. Tangent linear and adjoint model compiler, users manual. Technical report, 1997. [www <http://www.autodiff.com/tamc>].
- [6] R. Giering and T. Kaminski. Recomputations in reverse mode AD. In [3] *Automatic Differentiation of Algorithms, from Simulation to Optimization*, pages 283–291, 2002.
- [7] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Frontiers in Applied Mathematics, 2000.
- [8] INRIA Tropics team. On-line documentation of the Tapenade AD tool. Technical report. [www <http://www.inria.fr/tropics>].