# Generating recomputations in reverse mode AD

**Ralf Giering**[1]
**Thomas Kaminski**[2]

ABSTRACT The main challenge of the reverse (or adjoint) mode of automatic differentiation (AD) is providing the accurate values of required variables to the derivative code. We discuss different strategies to tackle this challenge. The ability to generate efficient adjoint code is crucial for handling large scale applications. For challenging applications, efficient adjoint code must provide at least a fraction of the values of required variables through recomputations, but it is essential to avoid unnecessary recomputations. This is achieved by the Efficient Recomputation Algorithm implemented in the Tangent linear and Adjoint Model Compiler and in Transformation of Algorithms in Fortran, which are source-to-source translation AD tools for Fortran programs. We describe the algorithm and discuss possible improvements.

**Keywords:** automatic differentiation, reverse mode, adjoint model, recomputations, source-to-source translation, Fortran, program slicing

## 1 Introduction

For a differentiable function represented by a numerical algorithm, automatic differentiation (AD) [7] is the process of constructing a second algorithm that represents the function's derivative. This second algorithm is an implementation of the chain rule, i.e. of a product of the Jacobian matrices representing the derivatives of the individual steps in the algorithm that defines the function. If the number of the function's input variables exceeds the number of output variables, it is favorable to have this derivative algorithm operate in the reverse mode of AD (adjoint code), i.e. to evaluate this matrix product in the order reverse to that of the function algorithm. The entries of these Jacobian matrices, however, contain values of variables of the function code, which are called required variables. Another type of required variables are those whose values determine the control flow in the function code [2]. Providing the accurate values of required variables to

[1]FastOpt, Hamburg, Germany, (Ralf.Giering@FastOpt.de).
[2]Max-Planck-Institut für Meteorologie, Hamburg, Germany, (kaminski@dkrz.de).

the derivative code in an efficient way is by no means trivial but the key challenge for constructing efficient adjoint code. Given that for many applications the function code alone consumes a large fraction of the available computer resources already, efficiency of the adjoint code is indispensable.

A way to provide the values of required variables is to store them in memory or on disk during an initial function evaluation and then access them while evaluating the derivative. For large scale applications so called check pointing algorithms [8] arrange multiple use of the same storing space. Since this multiple use requires multiple forward integrations of the function code, the additional CPU requirements still make it infeasible to provide all required values exclusively by storing. An alternative consists in recomputing the values of the required variables within the derivative code. But again, it is not difficult to imagine relevant examples for which the cost of these recomputations exceeds the available CPU time. Hence, many large scale applications can only be handled, if an efficient combination of storing and recomputation is applied [6].

The Tangent linear and Adjoint Model Compiler (TAMC, [4]) is a source-to-source translation AD tool for Fortran routines, which supports a combination of storing and recomputation. By default TAMC generates code for recomputation based on the Efficient Recomputation Algorithm (ERA), which we present in this paper. In addition, storing and reading of required variables is handled automatically by a few library functions, which arrange the necessary bookkeeping. This feature is triggered for variables indicated by the user through directives in the function code.

In section 2 we use a simple example to introduce the basic strategies of storing and recomputation. Section 3 discusses the analysis steps and techniques that ERA is based upon and presents the algorithm itself. Eventually section 4 contains our conclusions.

## 2 Strategies of providing required variables

Using a simple example of a sequence of statements, this section introduces the general strategies for generating corresponding adjoint code. Although a *sequence of statements* or *block*[3] is no explicit statement in some programming languages (e.g. Fortran), in terms of reverse mode AD it is one of the most important structures. Such a block is a group of statements that is executed consecutively, i.e. there is no control flow changing statement inside a block. The statements a block is composed of are elementary statements like assignments, subroutine calls, conditional statements, or loops, which can contain blocks themselves. Hence, blocks define a multi level hierarchy of partitions of the function code, which on the coarsest level partitions the

[3] Note, the block we define here is different from a basic block defined in code analysis.

```
1    y = 2*x
2    w = cos(y)
3    y = sin(y)
4    z = y*w
```

FIGURE 1. A simple example of a block with overwriting of y, all variables are active

```
S1   s1 = y
1    y = 2*x
S2   s2 = w
2    w = cos(y)
S3   s3 = y
3    y = sin(y)
S4   s4 = z
4    z = y*w

R4   z = s4
A4   adw += y*adz
     ady += w*adz
     adz = 0

R3   y = s3
A3   ady = cos(y)*ady

R2   w = s2
A2   ady += -sin(y)*adw
     adw = 0

R1   y = s1
A1   adx += 2*ady
     ady = 0
```

```
1    y = 2*x
2    w = cos(y)
S3   s3 = y
3    y = sin(y)
4    z = y*w

A4   adw += y*adz
     ady += w*adz
     adz = 0

R3   y = s3
A3   ady = cos(y)*ady

A2   ady += -sin(y)*adw
     adw = 0

A1   adx += 2*ady
     ady = 0
```

FIGURE 2. Adjoint of the block of Fig. 1 generated by a *store-all-modified-variables* strategy (SA, left panel) and a *minimal-store* strategy (SM, right panel). `a += b` is a shorthand notation for `a = a+b`.

top level routine to be differentiated and on the lowest level reaches the elementary assignments.

The block shown in Fig. 1 contains four assignments, one of which (statement 3) is overwriting the variable y, which was defined previously (statement 1).

The first algorithm employs the *store-all-modified-variables* (SA) strategy implemented in the AD tool Odyssee[11] for blocks defined by subroutine bodies and yields the code depicted in the left hand panel of Fig. 2. The adjoint statements (denoted by a leading A) reference both adjoint vari-

```
E4.1 y = 2*x
E4.2 w = cos(y)
E4.3 y = sin(y)
A4   adw += y*adz
     ady += w*adz
     adz = 0

E3.1 y = 2*x
E3.2 w = cos(y)
A3   ady = cos(y)*ady

E2.1 y = 2*x
A2   ady += -sin(y)*adw
     adw = 0

A1   adx += 2*ady
     ady = 0
```

```
E4.1 y = 2*x
E4.2 w = cos(y)
E4.3 y = sin(y)
A4   adw += y*adz
     ady += w*adz
     adz = 0

E3.1 y = 2*x

A3   ady = cos(y)*ady

A2   ady += -sin(y)*adw
     adw = 0

A1   adx += 2*ady
     ady = 0
```

FIGURE 3. Adjoint of the block of Fig. 1 generated by the *recompute-all* strategy (RA, left panel) and the *minimum-recomputation* strategy (RM, right panel).

ables (marked by the prefix `ad`) and required variables. The generation of the pure adjoint statements, i.e. without recomputations, is discussed in more detail by Giering and Kaminski [5]. In the so-called split mode [9] during a preceding forward sweep the value of every variable is saved to an auxiliary variable (S1, ..., S4) before a new value is assigned. During the following reverse sweep through the adjoints of the individual statements the value previously saved in front of a statement is restored in front of the corresponding adjoint statement (R4, ..., R1). By this algorithm it is ensured that all required variables have their correct values.

In the previous example several saves and restores are unnecessary. Applying a *minimum-store* strategy (SM), only stores and restores of those variables that are referenced by the adjoint statements will be inserted. In our example the restores R1, R2, R4 and corresponding stores are not needed, because the restored value is not required. An alternative algorithm is described by Faure and Naumann [3].

Alternatively, instead of saving required variables, code for recomputing their values can be inserted. Applying the straight forward *recompute-all* strategy (RA) to a block consists in preceding every adjoint statement by the fraction of the block which precedes the corresponding statement. For our code example this yields the adjoint code shown in Fig. 3.

Obviously, some of the recomputations are unnecessary and can be avoided by a more sophisticated strategy. In the adjoint code shown in the lefthand panel of Fig. 3 recomputations E3.2 and E.2.1 are not needed, because variable `w` is not referenced thereafter and variable `y` already has

its required value. This *minimum-recomputation* strategy (RM) yields the code shown in the righthand panel of Fig. 3. ERA follows this strategy.

The strategies SA and RA are easy to implement, while the corresponding strategies SM and RM require a sophisticated data flow analysis. However, this additional analysis improves the efficiency of the code. Comparing both more sophisticated strategies RM and SM, clearly RM is more efficient in terms of memory, while SM is more efficient in terms of arithmetic operations. For a given problem, machine characteristics, namely the access time to memory and the speed of arithmetic operations, determine the ratio of the execution times of RM and SM.

# 3 Efficient Recomputation Algorithm

This section presents ERA, which is based on the multi-level hierarchy of partitions of the function code into blocks introduced in section 2. ERA is based on a data flow analysis, which analyzes the memory accesses of statements. In general there are two kinds of accesses namely read and write. In higher programming languages, memory is accessed by means of variables which can be, e.g., scalar variables or array variables. Especially for array variables, their memory access pattern can be very complex. In general every reference of a variable accesses a set of memory locations.

We consider here neither the representation of the sets of memory locations nor the implementation of the operations (union, intersection) on them. Also the representation of incomplete knowledge and its influence on the operations in order to gain must and may information is not discussed here. For more details on these topics see e.g. [1]. We assume in the following that there is a representation of memory accesses including all necessary operations available and that the knowledge is complete. For readability we will denote the set of memory locations that variables access simply as *set of variables*.

Fig. 4 shows the heart of ERA, the function AD_EFFREC represented in the notation ICAN [10]. The algorithm uses several functions explained briefly.

**SLICE( stmt, vars )** SLICE is a function that takes as input a statement or sequence of statements (stmt) and a set of variables (vars). It computes the subset of variables that can be computed by the input set of statements and then returns the subsequence of statements that generates this subset of variables (by removing all 'unnecessary statements').

**IS_ACTIVE( stmt )** IS_ACTIVE is a function that returns true if the statement is active, i.e. computes active variables. This attribute is determined in a foregoing data flow analysis.

```
function AD_EFFREC(S, F) : A
if S is not block then
  A = AD_ELEMENTARY(S,F)
else
  required := ∅; killed := ∅; A := ⟨⟩
  for i = N to 1 step -1 do
    if required ≠ ∅ then
      B_i := SLICE( S_i, required)
      required := (required - GEN(B_i)) ∪ USE(B_i)
      killed ∪ = KILL(B_i); A := B_i ⊙ A
    end if
    if IS_ACTIVE(S_i) then
      A_i := AD_EFFREC(S_i, F ⊙ ⨁_{k=1}^{i-1} S_k)
      invalid := killed ⋂ USE(A_i)
      valid := USE(A_i) - invalid
      if invalid ≠ ∅ then
        lost := ∅
        repeat
          invalid ∪ = lost
          valid − = lost
          E_i := SLICE( F⊙ ⨁_{k=1}^{i-1} S_k, invalid)
          lost := KILL(E_i) ⋂ valid
        until lost = ∅
        killed ∪ = KILL(E_i); A := A ⊙ E_i
      end if
      required ∪ = valid; killed ∪ = KILL(A_i); A := A ⊙ A_i
    end if
  end for
end if
return A
```

FIGURE 4. Function AD_EFFREC, which forms the heart of ERA. Sequences of statements are denoted by upper case letters; $\odot$ denotes concatenation of two sequences and $\bigoplus_{k=1}^{i}$ denotes a concatenation of $i$ of these sequences. $\langle\rangle$ is the empty sequence. Functions are denoted by upper case words, they are explained in the text. Sets of variables are denoted by lower case words; the standard notation for the operators is used.

**AD_ELEMENTARY( stmt, leading_stmt )** AD_ELEMENTARY is a function, that generates the adjoint of an elementary statement. For more complex statements containing blocks themselves, AD_ELEMENTARY will call AD_EFFREC, in which case the adjoint statement might contain recomputations. In some cases leading statements (a slice of leading_stmt) have to be inserted to recompute lost values.

**KILL( stmt )** KILL is a function that returns the set of variables overwritten by the input sequence of statements (stmt). Note that adjoint variables do not need to be taken into account here.

**GEN( stmt )** GEN is a function that returns the set of variables, of which every variable is defined by a statement of the input sequence of statements (stmt) but not overwritten by any of the following statements of the sequence.

**USE( stmt )** USE is a function that returns the set of variables that are used before possibly being overwritten by the input sequence of statements (stmt). Note that adjoint variables do not need to be taken into account here.

The input of AD_EFFREC is a block of statements $S = \langle S_1, ..., S_N \rangle$ and all code F that precedes S. The output is the adjoint A of the block. ERA starts by calling AD_EFFREC with the coarsest partition of the top level routine of the function code and $F=F_0$, which is the code to reconstruct the values of the independent variables[4]. Then AD_EFFREC works top down by recursively calling itself, with the elements of the partition as arguments. When the level of the simple assignment is reached, AD_EFFREC constructs the corresponding adjoint. Going bottom up again, AD_EFFREC then level by level composes the adjoint code and propagates all information that is necessary for doing so.

When called with a block S that contains only an elementary statement AD_EFFREC calls the function AD_ELEMENTARY (see function description below), which constructs the corresponding adjoint code according to the rules given in [5]. Otherwise AD_EFFREC walks backward through the block statement by statement and updates three quantities: the sequence of generated statements (A), the set of variables (required) required by the sequence, and the set of variables (killed) overwritten by it. For the current statement, A is enlarged in two steps as depicted by Fig. 5: Firstly, for the current statement $S_i$ a slice of it ($B_i$) is generated that computes accumulated required variables and prepended to A. Secondly, if the statement $S_i$ is active (see [5]), the adjoint statement ($A_i$) is generated. The adjoint

[4]Since, in most cases, there is no code given to recompute the independent variables they have to be stored at the very beginning of the adjoint top-level routine.

$$\mathrm{B}_i \odot \boxed{\mathrm{B}_{i+1} \ldots \mathrm{B}_{N-1} \odot \mathrm{A}_N \ldots \mathrm{E}_{i+1} \odot \mathrm{A}_{i+1}} \odot \mathrm{E}_i \odot \mathrm{A}_i$$

FIGURE 5. Principle of enlargement of A, the sequence of statements forming the adjoint of a block S. For a given statement $\mathrm{S}_i$ of the block, $\mathrm{A}_i$ is the adjoint, $\mathrm{B}_i$ a slice of $\mathrm{S}_i$, and $\mathrm{E}_i$ a slice of all statements in the entire function before $\mathrm{S}_i$ (see text). Note, $\mathrm{B}_N$ and $\mathrm{E}_N$ are empty because initially required is the empty set.

statement might require variables ($\mathrm{USE}(\mathrm{A}_i) \subset \mathrm{USE}(\mathrm{S}_i)$) which must be provided by statements to be included. But some of these required variables (invalid) might have been overwritten by recomputations of previously generated adjoint statements ($\mathrm{A}_k$, $k > i$) or by previously included slices of recomputations ($\mathrm{B}_k$, $k > i$), in which case they cannot yet be provided by the first slicing. For this set of variables additional recomputations are generated by slicing the statements of the sequence up to but not including the current statement of interest ($\bigoplus_{k=1}^{i-1} S_k$) concatenated to all statements (F) preceding the block. Note that $\mathrm{USE}(\mathrm{E}_i) = \emptyset$, since $\mathrm{USE}(\mathrm{F}) = \emptyset$. This slicing is done in a repeat loop in order to find the minimum of recomputations that do not overwrite required variables. The resulting statements are appended to A, and the sets of variables killed and required by A are updated accordingly.

In summary, A is build by walking backwards through the sequence of statements. Recomputations are added on the left and, if necessary, on the right and adjoint statements are added on the right. The sets of killed and required variables are stored and thus are accessible by the functions USE(A) and KILL(A).

ERA is not always able to generate the code according to the RM strategy of section 2. For example, in Fig. 3 ERA will unnecessarily include statement E2.1. A more sophisticated algorithm requires a more detailed analysis of the validity of values that already have been recomputed TAMC applies an extension of ERA which handles this case. Further discussion is beyond the scope of this paper.

# 4 Conclusions

We demonstrated that to generate efficient adjoint code, which is indispensable for large scale applications, a sophisticated analysis of the function code is needed. We have described the Efficient Recomputation Algorithm (ERA) which, based on a multilevel hierarchy of partitions of the function code into blocks, works recursively from the top level routine to the elementary statements on the lowest level (top down) and then walks its way up (bottom up) again generating the adjoint code. It requires a data flow and data dependence analysis of the code and the ability to do program slicing for arbitrary code segments.

The current version of the Tangent linear and Adjoint Model Compiler (TAMC) uses an extended version of ERA, which in some cases is capable of avoiding recomputations of values that are still valid. Other possible extensions are not yet included in TAMC: The minimum recomputation strategy could be modified to work on loops. Those can be looked upon as sequences of statements of which each statement is the kernel of the loop (loop unrolling). TAMC currently does not take into account the array access patterns and thus may generate unnecessary recomputations, if array sections are required which do not overlap with further array sections that are killed. Currently the scope of ERA is limited to Fortran routines (subroutines and functions), i.e. the starting point of recomputations begins with the input variables of routines, which are stored and restored if necessary. The extended algorithm would require slicing of routines to generate clones of them that compute only a subset of their original output.

For most large scale problems, the most efficient adjoint code, however, applies a combination of storing and recomputation. TAMC supports storing and restoring of a subset of the required variables that is indicated by the user through directives in the function code. For those required variables, storing and restoring is handled automatically through library routines. The new AD tool Transformation of Algorithms in Fortran (TAF), by default already chooses a suboptimal combination of storing and recomputation automatically. Still the user has the opportunity to improve this combination by inserting directives in the function code. To determine the optimal combination would require the ability to quantify the cost of recomputations and storing/restoring for a particular platform and compiler.

Although TAMC and TAF are restricted to Fortran routines, ERA can be implemented to differentiate functions represented in other numerical programming languages.

## Acknowledgments

Thomas Kaminski was supported by the Bundesministerium für Bildung und Forschung (BMBF) under contract number 01LA9898/9.

## 5 References

[1] Beatrice Creusillet and F. Irigoin. Interprocedural array region analysis. Rapport cri, A-282, Ecole des Mines de Paris, FRANCE, January 1996.

[2] Christele Faure. Adjoining strategies for multi-layered programs. Rapport de recherche 3781, INRIA, BP 105-78153 Le Chesnay Ceded, FRANCE, Oktober 1999.

[3] Christele Faure and Uwe Naumann. The trajectory problem in AD. In Laurent Hascoet and Christele Faure, editors, *AD 2000*, pages 111–222. Springer Verlag, Berlin, Germany, 2000.

[4] Ralf Giering. *Tangent linear and Adjoint Model Compiler, Users manual*, 1997. unpublished, available at `http://puddle.mit.edu/~ralf/tamc`.

[5] Ralf Giering and Thomas Kaminski. Recipes for Adjoint Code Construction. *ACM Trans. On Math. Software*, 24(4):437–474, 1998.

[6] Ralf Giering and Thomas Kaminski. On the performance of derivative code generated by TAMC, 2000. submitted to Optimization Methods and Software.

[7] Andreas Griewank. On automatic differentiation. In Masao Iri and Kunio Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, Dordrecht, 1989.

[8] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[9] Andreas Griewank. *Computational Differentiation.* Springer, New York, Berlin, 2000.

[10] Steven S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[11] Nicole Rostaing, Stéphane Dalmas, and André Galligo. Automatic differentiation in Odyssée. *Tellus*, 45A:558–568, 1993.