Source Templates for the Automatic Generation of Adjoint Code Through Static Call Graph Reversal

Uwe Naumann[1] and Jean Utke[2]

[1] Software and Tools for Computational Engineering,
RWTH Aachen University, D-52056 Aachen, Germany
naumann@stce.rwth-aachen.de,
http://www.stce.rwth-aachen.de
[2] Mathematics and Computer Science Division,
Argonne National Laboratory, 9700 S. Cass Avenue,
Argonne, IL 60439, USA
utke@mcs.anl.gov,
http://www.mcs.anl.gov

Abstract. We present a new approach to the automatic generation of adjoint codes using automatic differentiation by source transformation. Our method relies on static checkpointing techniques applied to an extended version of the program's call graph. A code template is provided to implement a control structure governing the execution of the adjoint and augmented forward versions of each subroutine in the program. These code variants are generated automatically by algorithms that are independent of the programming language of the original code. The major advantage of this new approach is its flexibility with respect to various reversal schemes.

1 Context and Outline

This paper discusses novel algorithmic choices made in the context of the ongoing work on OpenAD (see www.mcs.anl.gov/OpenAD) – a software tool for the automatic generation of adjoint codes. OpenAD is being developed as part of the Adjoint Compiler Technology and Standards (ACTS, see www.autodiff.org/ACTS) project. The main application of this tool within the ACTS project is the MIT General Circulation Model (MITgcm, see mitgcm.org).

The structure of the paper is as follows. In Section 2 we discuss the basics of adjoint code construction and present an example for an adjoint code at the level of single subroutines. Two static call graph reversal modes are explained in Section 3. In Section 4 we consider a new approach to the generation of adjoint code based on code templates. A successful application of the new method is presented in Section 5.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3514, pp. 338–346, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 Introduction

Inverse methods [1] play an increasingly important role in various application areas of numerical scientific computing. Such methods are of interest, for example, in the context of large-scale gradient-based optimization methods as they eliminate the dependence of the complexity of the gradient computation on the dimension of the parameter space. We assume that we are given a computer program that implements some mathematical model

$$\mathbf{y} = F(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m \quad . \tag{1}$$

Our aim is to derive an adjoint program that computes the product of the transposed Jacobian matrix $(F'(\mathbf{x}))^T$ with an adjoint vector $\bar{\mathbf{y}}$ in the image space $\mathbb{R}^m$. This modification of the program's semantics is to be performed by a source transformation tool for automatic differentiation. In a compiler like fashion the original program for F is parsed into an abstract intermediate representation (ir). The source transformation is performed based on a set of static analyses [2] on ir. The modified internal representation $\bar{ir}$ is unparsed to obtain the adjoint program. This process is illustrated in Figure 1. Automatic differentiation (AD)

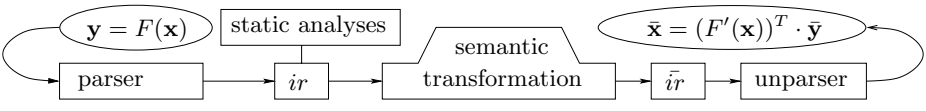


Fig. 1. Source transformation tool for the automatic generation of adjoint codes

[3, 4, 5] is a set of techniques for transforming numerical programs into derivative code that can be used to compute derivatives of vector functions such as Jacobians, Hessians, or higher-order Taylor coefficients. A detailed description of the mathematical foundations of AD is beyond the scope of this paper. Refer to [6] for a discussion of the theory.

The adjoint of a program implementing a vector function as in Equation (1) is obtained by the reverse mode of AD. Given values for $\mathbf{x}$ and the adjoints of the original outputs $\bar{\mathbf{y}}$, the adjoint program computes the transposed Jacobian vector product

$$\bar{\mathbf{x}} = (F')^T \cdot \bar{\mathbf{y}} \quad . \tag{2}$$

This process is best introduced with the help of a simple example that illustrates the basic features. Consider the Fortran implementation of a function $\mathbf{y} = F(\mathbf{x})$ depicted in Figure 2(a), where $\mathbf{x} \in \mathbb{R}^4$, $\mathbf{y} \in \mathbb{R}$. Often the program that implements F consists of a possibly large number of subroutines calling each other. In this paper we consider methods for implementing adjoint codes at the *interprocedural* level. Our approach uses a given solution for the problem of constructing adjoints at the *intraprocedural* level like the one shown in Figure 2(b),(c).

An execution of the *augmented forward code* (as in Figure 2(b)) stores the flow of control [2] in an integer stack (`IS`) as well as all numerical values (in a

(a)
```
y=1.
do i=1,4
  if (i<3) then


    x(i)=sin(x(i))
  else


    x(i)=x(i)*x(i)
  end if

  y=y*x(i)
end do
```

(b)
```
y=1.
do i=1,4
  if (i<3) then
    push(IS,1)
    push(FS,x(i))
    x(i)=sin(x(i))
  else
    push(IS,0)
    push(FS,x(i))
    x(i):=x(i)*x(i)
  end if
  push(FS,y)
  y=y*x(i)
end do
```

(c)
```
do i=4,1,-1
  call pop(FS,y)
  x_b(i)=x_b(i)+y*y_b
  y_b=x(i)*y_b
  call pop(IS,branchId)
  if (branchId==1) then
    call pop(FS,x(i))
    x_b(i)=cos(x(i))*x_b(i)
  else
    call pop(FS,x(i))
    x_b(i)=2*x(i)*x_b(i)
  end if
end do
```

Fig. 2. Intraprocedural adjoint code: original (a), augmented forward (b), adjoint (c)

floating-point stack `FS`) needed for the adjoint computation during the following reverse sweep. In principle the adjoint code is obtained by applying Equation (2) to each statement $y = \phi(x_1, \ldots, x_k)$ of the original code in reverse order (w.l.o.g. we consider scalar assignments). This application yields $\bar{x}_i = \bar{x}_i + \frac{\partial \phi}{\partial x_i} \bar{y}$ for $i = 1, \ldots, k$. The increment of the current value of $\bar{x}_i$ is due to the fact that x_i can occur on several right-hand sides. Adjoint versions of variables in the original code are marked by the suffix `_b`. For example, the code in the example shown in Figure 2(c) requires the values of `x(i)` and `y`, which are therefore stored in Figure 2(b). A detailed discussion of the reversal of the flow of control inside a subroutine can be found in [7].

Adjoint codes permit the accumulation of the Jacobian at a cost proportional to the number of outputs m. Of particular interest are gradients ($m = 1$) obtained at a small constant multiple of the cost of evaluating the function itself, for example, in the context of data assimilation in the MITgcm. The gradient of some objective with respect to parameters at the grid points of a very fine discretization leads to a number of input variables n on the order of 10^9. Neither forward-mode AD nor approximation by finite difference quotients represents a feasible approach, as either has a complexity of $O(n)$.

3 Call Graph Reversal Modes

In the remainder of this paper we assume that adjoint code is available for all subroutines in the given program generated by a method similar to the one sketched in the previous section. The *call graph* (CG) is usually defined as a graph with nodes representing subroutines and edges representing potential (direct) calls [2]. It is a static entity and generally not acyclic. To be useful for the static construction of adjoint code at the interprocedural level, we need to break the cycles, a process generally possible only for a concrete execution of the program. The result is the *dynamic call tree* (DCT). Edges represent subroutine calls and the order of execution in the context of a depth-first traversal. The vertices represent executions of variants of subroutine code (forward, augmented forward,

```
subroutine 1
  call 2; ... call 4; ... call 2;
end subroutine 1
subroutine 2
   call 3
end subroutine 2
subroutine 4
  call 5
end subroutine 4
```

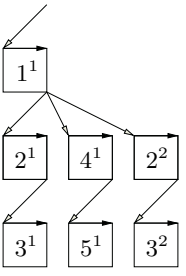


Fig. 3. Dynamic call tree of a simple calling hierarchy

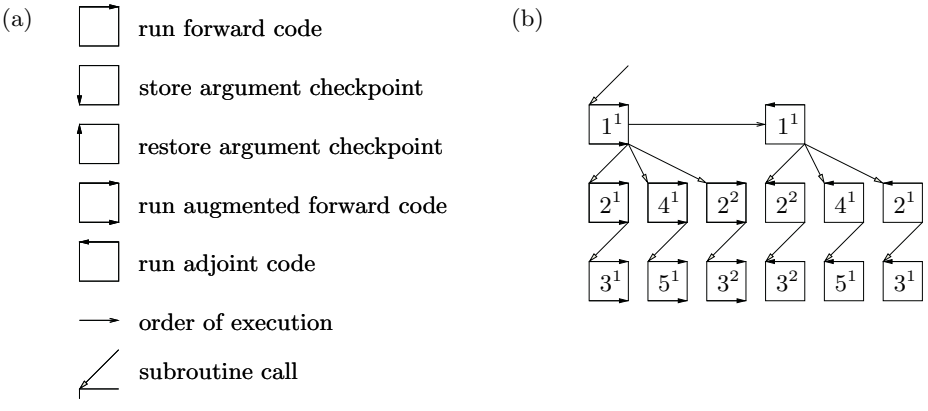


Fig. 4. Symbols for reversal mode graphs (a), DCT of split reversal mode (b)

adjoint). The order of calls to other subroutines shown on the next lower level is implied by the left to right order of edges emanating from a given vertex. An edge pointing to a vertex on the same level indicates the execution of a different code variant of the same subroutine. Hence, the depth-first traversal is well defined. Figure 3 shows the DCT for a simple example. Vertices are labeled with the identifier of the respective subroutine. Superscripts denote the instance of the call, as one and the same subroutine can be called repeatedly. In the example, subroutines 2 and 3 are called twice. The symbols are explained in Figure 4(a). Note that for the purpose of static reversal schemes the DCT is merely a conceptual tool and does not need to be instantiated in practice.

Two basic call graph reversal modes have been proposed in the literature [6]. In *split* reversal mode the adjoint computation is preceded by an execution of the augmented forward code of the entire program. All values that are needed for the correct reversal of the intraprocedural flow of control, as well as those required to evaluate the adjoint assignments correctly, are stored. The split reversal of the example in Figure 3 is shown in Figure 4(b). For computationally challenging problems this approach usually leads to prohibitively large memory requirements.

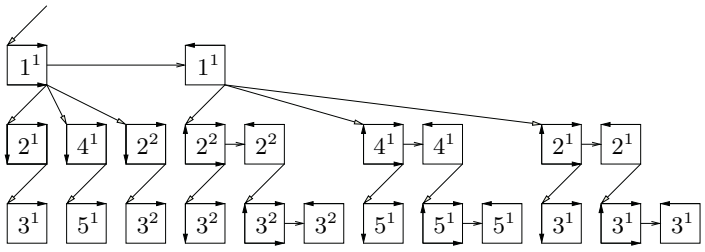


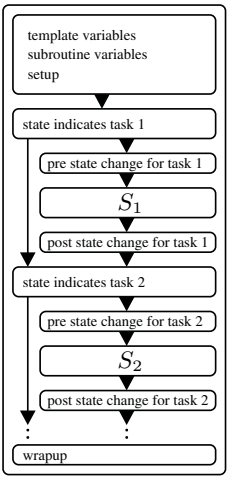
Fig. 5. DCT of adjoint obtained by joint reversal mode

This problem can be mitigated by the *joint* reversal mode, where a trade-off between memory requirement and computational complexity is achieved by *checkpointing* [8] at the level of subroutine arguments. The augmented forward code variant directly followed by the adjoint code variant of the same subroutine is run only when the adjoint values of the inputs of this subroutine are required for executing the adjoint code variant of the calling subroutine. While traversing the call graph backwards, one needs to know the input values to a subroutine to run its augmented forward code. Rather than recomputing these inputs, we first store them as *argument checkpoints* and later restore them for use with the augmented forward run. Figure 5 illustrates this statement for the example in Figure 3. Obviously, joint reversal represents a trade-off between the consumption of memory resources and the number of instructions performed by the adjoint code. Split and joint reversal modes can be combined in a hierarchical fashion as described in [6] and Section 5.

4 Code Templates

Section 3 outlines the principal approach to control a reversal scheme. The basic building blocks are variants S_i of the original subroutine code S_0, each accomplishing one of the tasks shown as a subroutine symbol in Figure 4(a). The S_i are created by transforming the source code contained in the respective subroutine bodies. To integrate the S_i into a particular reversal scheme, we need to be able to make all subroutine calls in the same fashion as in the original code and, at the same time, control which task each subroutine call accomplishes. We replace the original subroutine body with a branch structure in which each branch contains one S_i. The execution of each branch is determined by a global control structure whose members represent the state of execution in the reversal scheme. The branches contain code for pre- and post-state transitions enclosing the respective S_i. This ensures that the transformations producing the S_i do not depend on any particular reversal scheme. In practice we insert the S_i in a subroutine template, shown in Figure 6(a). The template is written in the target language, and the insertion is done by a postprocessing step that identifies specific pragmas in the template code as insertion points for the S_i. Anything related to setting up storage for taping

(a)



(b)

```
subroutine template()
  use OpenAD_tape ! tape storage
  use OpenAD_rev  ! state structure
!$TEMPLATE_PRAGMA_DECLARATIONS

  if (rev_mode%tape) then
! the state component
! 'taping' is true
!$PLACEHOLDER_PRAGMA$ id=2
  end if

  if (rev_mode%adjoint) then
! the state component
! 'adjoint' run is true
!$PLACEHOLDER_PRAGMA$ id=3
  end if end subroutine template
```

Fig. 6. Subroutine template components (a), split-mode Fortran90 template (b)

or checkpointing, such as declaring data arrays or referring to external modules, can be coded directly into the template. The following paragraphs show examples.

Split Reversal: Split reversal is the simplest static reversal mode. We first execute the entire computation with the augmented forward code (S_2) and then follow with the adjoint (S_3). From the task pattern shown in Figure 4(b) it is apparent that, aside from the top-level routine, there is no change to the state structure within the call tree. Therefore, there is no need for state changes within the template. Since no checkpointing is needed either, we have only two tasks: producing the tape and the adjoint run. Figure 6(b) shows a simplified version of the split-mode template used in OpenAD. The different S_i in the `PLACEHOLDER_PRAGMA` are identified by their respective $i =$`id`. The state is contained in `rev_mode`, a static Fortran90 variable in module `OpenAD_rev` of type `modeType` also defined in this module. In order to perform a split-mode reversal for the entire computation, a driver routine calls the top-level subroutine first in taping mode and then in adjoint mode.

Joint Reversal with Argument Checkpointing: Figure 5 illustrates the task pattern for a joint reversal scheme that requires state changes in the template as well as the insertion of more tasks. Figure 7 shows a simplified version of the template used in OpenAD. The state transitions in the template directly relate to the pattern shown in Figure 5. Each prestate change applies to the callees of the current subroutine. Since the argument store (S_4) and restore (S_6) do not contain any subroutine calls they do not need state changes. Looking at Figure 5, one realizes that the callees of any subroutine executed in plain forward mode (S_1) never store the arguments (only callees of subroutines in taping mode do). This explains lines 18, 25, and 30. Further-

more, all callees of a routine currently in taping mode are not to be taped but instead run in plain forward mode, as reflected in lines 27 and 28. Joint mode in particular means that a subroutine called in taping mode (S_2) has its adjoint (S_3) executed immediately after S_2. This is facilitated by line 33, which makes the condition in line 35 true, and we execute S_3 without leaving the subroutine. Any subroutine executed in adjoint mode has its direct callees called in taping mode, which in turn triggers their respective adjoint run. This is done in lines 37–39. Finally, we have to account for sequence of callees in a subroutine; that is, when we are done with this subroutine, the next subroutine (in reverse order) needs to be adjoined. This process is triggered by calling the subroutine in taping mode, as done in lines 41–43. The respective top-level routine is called by the driver with the state structure having both `tape` and `adjoint` set to `true`.

```
 1:subroutine template()
 2:  use OpenAD_tape
 3:  use OpenAD_rev
 4:  use OpenAD_checkpoints
 5: !$TEMPLATE_PRAGMA_DECLARATIONS
 6:  type(modeType) :: orig_mode
 7:
 8:  if (rev_mode%arg_store) then
 9: ! store arguments
10: !$PLACEHOLDER_PRAGMA$ id=4
11:  end if
12:  if (rev_mode%arg_restore) then
13: ! restore arguments
14: !$PLACEHOLDER_PRAGMA$ id=6
15:  end if
16:  if (rev_mode%plain) then
17:    orig_mode=rev_mode
18:    rev_mode%arg_store=.FALSE.
19: ! run the original code
20: !$PLACEHOLDER_PRAGMA$ id=1
21:    rev_mode=orig_mode
22:  end if
23:  if (rev_mode%tape) then
24: ! run augmented forward code
25:    rev_mode%arg_store=.TRUE.
26:    rev_mode%arg_restore=.FALSE.
27:    rev_mode%plain=.TRUE.
28:    rev_mode%tape=.FALSE.
29: !$PLACEHOLDER_PRAGMA$ id=2
30:    rev_mode%arg_store=.FALSE.
31:    rev_mode%arg_restore=.FALSE.
32:    rev_mode%plain=.FALSE.
33:    rev_mode%adjoint=.TRUE.
34:  end if
35:  if (rev_mode%adjoint) then
36: ! run the adjoint code
37:    rev_mode%arg_restore=.TRUE.
38:    rev_mode%tape=.TRUE.
39:    rev_mode%adjoint=.FALSE.
40: !$PLACEHOLDER_PRAGMA$ id=3
41:    rev_mode%plain=.FALSE.
42:    rev_mode%tape=.TRUE.
43:    rev_mode%adjoint=.FALSE.
44:  end if
45:end subroutine template
```

Fig. 7. Joint mode Fortran90 template with argument checkpointing

5 Application

A simplified version of the MITgcm code is the so-called shallow water model, which can be used for problems of variable complexity. A simple split or joint mode implies storage or runtime requirements far beyond what is feasible. Hierarchical checkpointing has traditionally been used to allow for a trade-off between storage and runtime requirements. For the shallow water code we implemented a reversal scheme with a two-level hierarchical checkpointing splitting the main time-stepping loop into an inner loop i and an outer loop o. Wrapping the respective loop bodies into subroutines allows the use of the subroutine-level template approach illustrated in Figure 8. We show two loop iterations for each level, whereas the real-world problem potentially has thousands of steps at each level. Subroutine calls located outside the outer loop are nonrepetitive allowing for use of the

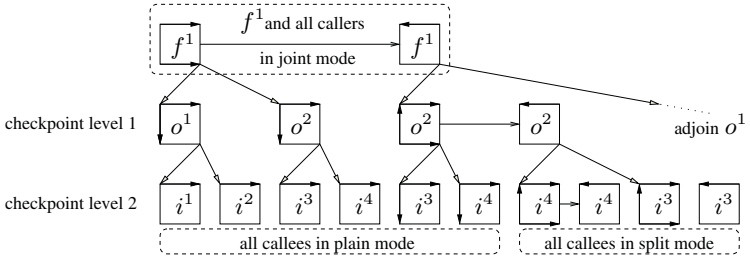


Fig. 8. Two-level hierarchical checkpointing scheme

joint mode because the few checkpoints require little memory. A single execution of the inner-loop body and its callees is short enough to fit all required values on the tape stack, allowing for use of the split mode.

The code transformation uses the templates introduced above for the subroutines handled in either mode respectively. To control the outer and inner level checkpointing, we use two additional templates that are similar to the joint-mode template but have small adjustments in the state transitions to control the checkpointing. The templates and their use in applying OpenAD to the shallow water model code are part of the case studies presented on the OpenAD website.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38 and by NSF under ITR contract OCE-0205590.

References

1. Wunsch, C.: The Ocean Circulation Inverse Problem. Cambridge U. Press, Cambridge (1996)
2. Aho, A., Sethi, R., Ullman, J.: Compilers. Principles, Techniques, and Tools. Addison-Wesley, Reading, MA (1986)
3. Berz, M., Bischof, C., Corliss, G., Griewank, A., eds.: Computational Differentiation: Techniques, Applications, and Tools. SIAM, Philadelphia (1996)
4. Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U., eds.: Automatic Differentiation of Algorithms: From Simulation to Optimization. Springer, New York (2002)
5. Griewank, A., Corliss, G., eds.: Automatic Differentiation of Algorithms: Theory, Implementation, and Application. SIAM, Philadelphia (1991)
6. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia (2000)

7. Naumann, U., Utke, J., Lyons, A., Fagan, M.: Control flow reversal for adjoint code generation. In: Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004), Los Alamitos, CA, USA, IEEE Computer Society (2004) 55–64
8. Griewank, A.: Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Optimization Methods and Software **1** (1992) 35–54