Cheaper Adjoints by Reversing Address Computations

L. Hascoët[*], J. Utke[†], U. Naumann[‡]

November 19, 2007

Abstract

The reverse mode of automatic differentiation is widely used in science and engineering. A severe bottleneck for the performance of the reverse mode, however, is the necessity to recover certain intermediate values of the program in reverse order. Among these values are computed addresses, which traditionally are recovered through forward recomputation and storage in memory. We propose an alternative approach for recovery that uses inverse computation based on dependency information. Address storage constitutes a significant portion of the overall storage requirements. An example illustrates substantial gains that the proposed approach yields and we show use cases in practical applications.

Keywords: program transformation, automatic differentiation, inverse computation

1 Introduction

We consider Automatic Differentiation (AD) implemented via source code transformation of a numerical program P. The result is an augmented program P^+ that also computes derivative information. Derivatives have many uses in Scientific Computing in general. Among these are gradients, which are central to many practical applications such as for instance gradient-based optimization in CFD [16] or variational data assimilation in weather forecasting or Earth sciences [17]. A major application of AD methods is the computation of gradients using the so-called reverse mode [21, 12]. The reverse mode of AD takes advantage of the structure the underlying function, which has typically a large number of input control variables versus very few or only one output cost function. The idea of reversed differentiated programs is to propagate partial gradients backwards from the end of the given program. The reverse mode of AD is seen

[*]INRIA Sophia-Antipolis, 2004 Route des lucioles, BP 93 06902 Valbonne, France

[†]corresponding author; Argonne National Laboratory, MCS, 9700 S. Cass Avenue, Argonne, IL 60439, USA; tel.: +1 630 252 4552; utke@mcs.anl.gov

[‡]RWTH Aachen,Seffenter Weg 23 D-52074 Aachen, Germany

as the discrete equivalent of solving the adjoint equations from optimal control theory. An impressive number of successful application of AD to practically relevant problems in science and engineering can be found in the proceedings of the four international conferences on the subject held in 1991 [6], 1996 [3], 2000 [5], and 2004 [4]. Refer to [13] for a discussion of AD from a mathematical perspective. The AD community maintains a webportal [2] with a vast amount of information on research and development, tools, and applications.

In AD we consider P as a sequence of p elementary numerical operations

$$v_i = \phi_i(\ldots, v_j, \ldots),\ i = 1, \ldots, p \quad .$$

Going in reverse order through the operations sequence, we generate for each original operation ϕ_i a set of adjoint operations. Each individual set contains one adjoint operation per *active* (see below) argument v_j used in ϕ_i.

$$\forall i = p, \ldots, 1 : \forall v_j \text{ argument in } \phi_i : \bar{v}_j = \bar{v}_j + c_{i,j} * \bar{v}_i$$

where

$$c_{i,j} \equiv \frac{\partial \phi_i(\ldots, v_j, \ldots)}{\partial v_j} \quad .$$

Therefore the runtime cost of the reverse differentiated program is independent from the number of inputs, and is in theory a small fixed multiple of the runtime of P itself. In contrast, computing the gradient through classical tangent directional differentiation, or even worse through divided differences, returns the gradient elements one by one, at a cost proportional to the number of inputs. This decisive advantage of the reverse mode makes it the most practical approach for today's large-scale optimization or data assimilation problems. Moreover divided differences produce merely an often low-quality approximation of the derivatives. This inaccuracy may result in poor convergence of numerical algorithms, for example, for nonlinear optimization with second derivatives [7].

The growing number of applications of reverse mode of AD motivates research to refine the reverse differentiation algorithms. The recurrent difficulty is memory (or runtime in recomputation oriented stategies). Real-world programs have limited variable space and will overwrite values. The reverse-mode adjoint operations, however, refer to many of these intermediate values and need to recover them when they have been overwritten in the course of executing P. Fundamental recovery strategies are *(1)* to recompute or *(2)* to store these values. The computational effort implied by the first strategy is in principle quadratic to that of computing P itself. The memory requirements of the second strategy are in principle proportional to the number of floating-point operations executed in P. For large programs, a general combination strategy called "checkpointing" is unavoidable. Checkpointing balances storage requirements and recomputations, and can be made optimal in some model cases. Details on the reverse mode, checkpointing, and control flow reversal can be found in [12].

For this paper we start out with the storing strategy and aim at reducing the storage requirements through refined analysis and code transformation. For

```
if (k > 10)
   a[i] = x * b[i]
   // add i, b[i] to $\mathcal{R}$
   ...
endif
// add k to $\mathcal{R}$
```

(a)

```
// recover k
if (k > 10)
   // recover i, b[i]
   $\overline{x}$ = $\overline{x}$ + b[i]*$\overline{a}$[i]
   ...
endif
```

(b)

Figure 1: (a) Example code section: the vector `b` is not considered active; (b) values to be recovered and adjoint statements for reverse mode AD

the sake of simplicity the set $\mathcal{R}$ of values that must be recovered is detected by specific analysis, for example TBR [14]. In many applications, derivatives are desired only for a subset of program variables, the *active* variables. Activity analysis accomplishes this separation and in conjunction with TBR further shrinks $\mathcal{R}$ by identifying operations ϕ_i that do not involve any active variables. Aside from the obvious values used to compute partial derivatives $c_{i,j}$, $\mathcal{R}$ also encompasses auxiliary values such as addresses and control flow decisions. Figure 1 shows the reverse-mode AD of a small code example. While `b` itself is not active, its elements are used in the partial derivative computation, and they have to be recovered. To run the suggested adjoint code, we also must recover the values of `i` and `k`, because they are used as indices and in the branch condition. Here we assume that `k` is not overwritten within the `if` branch. If it were, then we would have to refine the strategy. For example we could introduce a temporary to retain the initial `k`, then store it just *after* the branch. We can then retrieve the proper `k` before we enter the corresponding adjoint branch.

In the following we will consider sections of code that contain no checkpoints, which implies split-mode reversal; see [12]. There may be checkpoints before or after, though. We distinguish two phases. First is the *forward sweep* of the section, consisting of the original code potentially augmented by statements to store certain intermediate values. Second is the *backward sweep*, consisting of the corresponding adjoint statements including control flow reversal and statements to recover values referenced by the adjoint statements, the reversed control flow, or address computations.

2 Motivating Example

Consider the example in Figure 2(a). The values of `a[j]` and `a[k]` are directly needed to compute the partial derivatives and are in $\mathcal{R}$. The values of `k` and `j` occur as indices in the adjoint computation and therefore are also in $\mathcal{R}$; see Figure 2(b) or (c) and [8].

The standard approach pushes the elements of $\mathcal{R}$ at a certain point during the forward sweep onto a stack and pops them at the corresponding point during the backward sweep, for details see [12]. The implied stack growth is typically

```
1     j=1
2     do while (j<6)
3      k=j+1
4      j=k+2
5      a[j]=a[k]*a[j]
6     end do
```
(a)

```
1     pop(j); pop(k)
2     do while (j>=1)
3      pop(a[j])
4      ā[k]=ā[k]+a[j]*ā[j]
5      ā[j]=a[k]*ā[j]
6      k=j-2
7      j=k-1
8     end do
```
(b)

```
1     pop(j)
2     do while (j>=1)
3      pop(a[j])
4      k=j-2
5      ā[k]=ā[k]+a[j]*ā[j]
6      ā[j]=a[k]*ā[j]
7      j=k-1
8     end do
```
(c)

Figure 2: Example code (a) for a loop reversal, incorrect backward sweep (b) using a "statement-level recipe", and correct backward sweep (c)

mitigated by a checkpointing scheme that trades smaller stacks for recomputations. Consequently, any reduction of the stack size reduces the required recomputations and thereby has a direct impact on the overall performance.

The use of inversion was first described in [22]. Inspection of the code in Figure 2(a) suggests that one should be able to use inversion of the address computation for `j` and `k` to recover them during the backward sweep without any storage, except for the last `j`. In other code examples, one may be able to compute that last value. The manual transformation of the code as shown here is not as simple as the usual statement-level recipe that is used for the adjoint statements. On this example such a naive inversion would produce semantically incorrect code, as shown in Figure 2(b). Figure 3 shows the values for `j,k` after executing each line for two loop iterations for the code versions (a), (b), and (c). The boxed entries show the wrong indices for the adjoint statements on lines 4 and 5 as well as the `pop` on line 3 in (b) in the second iteration, which is the adjoint of the first loop iteration in (a). The correct code in Figure 2(c) properly restores the values by observing the dependencies. Note, that for choosing the adjoint in (c) the code in (a) still has to be augmented by the proper `push` calls corresponding to the `pop`s of `j` and `a[j]` on line 1 and 3, respectively.

	(a)		(b)		(c)	
	#1	#2	$\overline{\#2}$	$\overline{\#1}$	$\overline{\#2}$	$\overline{\#1}$
1	1,.		7,5		7,.	
2	1,.	4,2	7,5	4,5	7,.	4,5
3	1,2	4,5	7,5	$\boxed{4,5}$	7,5	4,2
4	4,2	7,5	7,5	$\boxed{4,5}$	7,5	4,2
5	4,2	7,5	7,5	$\boxed{4,5}$	7,5	4,2
6	4,2	7,5	7,5	4,2	7,5	4,2
7			4,5	1,2	4,5	1,2
8			4,5	1,2	4,5	1,2

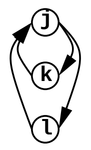
Figure 3: `j,k` per line number for two iterations (#1 and #2) for the loop in Figure 2(a) and the adjoint iterations ($\overline{\#2}$ and $\overline{\#1}$) in Figure 2(b) and (c).

For simple loop constructs such as the Fortran (`DO i=1,10`), AD tools, including OpenAD [23] and Tapenade [15] traditionally have been able to produce the proper adjoint loop avoiding storage of the loop index. This relies on data-dependence analysis and the ability to recognize a canonical loop index update and exit condition e.g. by matching a pattern. However, even a slight deviation from these patterns prevents proper handling of otherwise semantically equivalent cases.

We have to fit the inverse computation concept shown in the example into the source transformation context. Intuitively, we treat the integer assignments as an equation system that allows us to compute the successive values of address variables in reverse. We use each such assignment once to compute a value. Immediately one important restriction is revealed: The approach works only when the data dependencies imply a triangular system (as in our example) for which we can generate the explicit solution code. When the implied system is nontriangular, see, for example, Figure 4(a), this code-transformation-based approach is unable to compute the solution values because it would in principle require computer algebra or even nonlinear system solvers, which clearly go beyond the source transformation systems we consider. In the simplified dependency graph shown in Figure 4(b) we see nontriangularity reflected by the two loop-carried dependence arrows reaching `j` that cannot be inverted separately.

```
k=1; l=1;
do ...
 j=k+l
 k=j+3
 l=j+2
end do
```

(a)



(b)

Figure 4: Simple example code (a) implying a nontriangular system and the corresponding dependency graph (b)

The inverse computation that we are looking for is closely related to induction variable detection [25]. Actually, if one can discover that addresses are an affine function of the iteration counter as is the case for `j` and `k` in Figure 2(a), then one can recompute these addresses in reverse. However inversion does not even require that addresses are affine. It suffices that the values needed by inversion, e.g. the increments to `j` and `k`, retain their values during the program.

We aim to explore the semantic conditions that allow storage-saving inversions. These provide a conceptual framework and algorithms for analysis and code generation that include recognizing nontriangular systems and the ability to fall back to storing values if necessary.

3 Framework

We define the set $\mathcal{R}_A$ of the address variables used by the adjoint of active statements, following the principal approach in [8, 14]. Without going into much detail, we first determine the set of active variables by means of a particular data flow analysis. For each of these active variables we determine whether it uses a computed address. We consider a computed address any address that is not a fixed offset into a stackframe, a fixed heap address, or a fixed offset into the text segment. A fixed offset into the stack is, for instance, any local scalar variable, but a local array `A` with nonconstant subscript `A[i]` has variable offset. For dynamically allocated (heap) variables it is not quite as simple because they are all allocated at runtime. However, when an array `A[c]` is allocated once, not deallocated before process cleanup, and `c` is a constant, then we consider it a fixed heap address. A branch condition involving only constant values would be considered a fixed offset into the text segment. The last case is sometimes used for model reconfiguration in lieu of preprocessing the source code. All such computed address values are added to $\mathcal{R}_A$.

We have three principal options to recover the $r \in \mathcal{R}_A$:

1. *Inversion:* We can invert an assignment statement $w = f(v_i, i = 1..N)$ that uses $r \equiv v_k$ in the right-hand side provided the right-hand side expression f is indeed invertible and the left-hand side value w and all the other argument values $v_{i,i\neq k}$ are known. Invertibility of the expression graph f is given when all elementary operations ϕ on the path $r \rightarrow f$ are invertible wrt the predecessor of ϕ in $r \rightarrow f$ in the usual mathematical sense. Examples for the most common noninvertible practical operations are `max` and integer division.

2. *Forward recomputation:* We can execute a statement (assignment) that defines r provided all inputs are known.

3. *Storage:* We can store r. This is the fall-back option, if the previous two are not applicable.

Aside from handling certain canonical loops, AD tools use only storage and forward recomputation. For example, TAF and TAMC [9] by default rely on forward recomputation, whereas Tapenade and OpenAD by default rely on storage.

Figure 2 illustrates the importance of the data dependencies for the inversion approach. The data-dependence graph as defined for example in [19] has been used extensively to reschedule program statements. A prominent application is parallelization [1, 24]. In the present context of reverse-mode AD, the data-dependence graph appears as the appropriate data structure for our transformation algorithm to find a scheduling of inversions and forward recomputation. We shall thus consider the subset of the data-dependence graph that covers the definitions and the uses of the $r \in \mathcal{R}_A$.

As indicated earlier, we also consider control flow computations as generalized address computations (of stack addresses). Therefore, we also add such values to $\mathcal{R}_A$, that compute control flow affecting any source code section containing active variables.

In the following, we will refer to variables or values in $\mathcal{R}_A$ interchangeably and understand them as a pair of address (name) and definition locations. In practice this requires use-def chains [19] to traverse from a variable use to the set of its possible definitions. For example, the use of variable `j` on the right-hand side of line 3 in Figure 2(a) has a two element chain of definition locations consisting of the assignment on line 1 and the assignment on line 4. Thinking in terms of address-definition pairs simplifies the distinction between the successive values that a variable holds before and after it is overwritten.

Given the use-def chains, we build the $\mathcal{R}_A$-dependency graph $G_A = (\mathcal{V}, \mathcal{E})$ using the expression graphs for the $r \in \mathcal{R}_A$ extending from the leafs in these expression graphs (uses) to the definitions and adding the defining statement's right-hand-side expression graph. In other words, the nodes $v \in \mathcal{V}$ are the expression graph nodes of address computation expressions, and the edges $e \in \mathcal{E} = \mathcal{E}_c \cup \mathcal{E}_f$ are the computation dependencies $\mathcal{E}_c$ from the expression graphs and flow dependencies $\mathcal{E}_f$ from a given definition to the possible uses. As an illustration, Figure 5 shows G_A for the loop in our motivating example of Figure 2(a). All nodes defining a value are shown with a thick border. All uses of values are shown with a thin border. While G_A can in principle be made to represent an entire program, we consider a subgraph representative for a section of code such as the loop shown in Figure 2(a). Therefore G_A has a set of input nodes $\mathcal{V}_I \subset \mathcal{V}$, that is, nodes that have definition locations outside the code section of interest. $\mathcal{V}_I$ are definition nodes; see the thick-bordered `j` at the top of Figure 5. Likewise G_A has a set of output nodes $\mathcal{V}_O \subset \mathcal{V}$, that is, nodes with uses outside the code section of interest. $\mathcal{V}_O$ are use nodes; see the thin-bordered `j` at the very bottom of Figure 5. The $\mathcal{R}_A$ set itself consists of the two placeholders (shown on the bottom right) for uses originating in line 5 in Figure 2(a). The actual adjoint uses are visible in lines 5 and 6 of Figure 2(c). In our example all computation dependencies are invertible (denoted by a thick arrow), but invertible edges $\mathcal{E}_I$ generally are a subset of $\mathcal{E}_c$. One choice of options for a given G_A is called an *inversion strategy*, denoted by $S(G_A)$. For a given strategy S we count the number of stored values (option 3) in $|S|_s$ and the number of forward computed values (option 2) as $|S|_f$. The general cost of a strategy can be written as $c(S) = |S|_s + \alpha|S|_f$, where $\alpha \geq 0$ is a tradeoff factor weighing computations versus storage. It is of course rather difficult to determine α because it depends on many factors such as the hardware, the compiler, and the order of magnitude of $|S|_s$. Even a refined recomputation strategy such as [10] can lead to substantial runtime penalties and therefore non negligible α. As long as we choose to exclude in particular loops from the scope of the G_A we can assume that forward computations are vastly less expensive than storage, that is, set $\alpha \equiv 0$. Given the cost function and the above framework, we formulate our problem as follows.

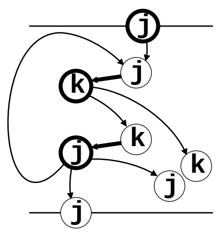


Figure 5: G_A for Figure 2(a)

Recovery Problem: Assuming the values of a subset of $\mathcal{V}$ are known, how can we combine recovery options 1–3 to obtain the values of all nodes in G_A at minimal cost?

While not formally proven here, we conjecture that the recovery problem is NP-hard for general G_A as shown under certain conditions for $\alpha \equiv 1$ in [20]. For the practical applications in this paper, we concentrate on loop bodies and therefore can often use a narrower formulation relating to loop-carried flow dependencies [19, Section 9.3], namely, dependencies that go from a variable definition during some iteration of a loop to a use of this variable at a following iteration.

Loop Recovery Problem: Assuming the values of all outputs $\mathcal{V}_O$ are known, how can we combine recovery options 1–3 to obtain the values of nodes in $\mathcal{R}_A$ and the values of the nodes in $\mathcal{V}_I$ that are destinations of loop-carried dependencies originating from $\mathcal{V}_O$ at minimal cost?

Section 4 introduces an algorithm to determine a strategy that relies on heuristics.

4 Algorithm for the Token Game

A known value of a node v in graph G_A can be illustrated by a "token" placed on that node. We define the predicate $T(v)$ to be true iff node v has a token; false (that is, the node does not have a token) is denoted by $\neg T(v)$. Computing more values allows us to place more tokens on nodes according to the rules implied by the three recovery options, hence the name *token game*. The graph-based approach provides more flexibility than the stack of states used in [22].

The goal of the algorithm is to find an inversion strategy $S(G_A)$ approximating minimal cost for a given G_A and a given set of known values. In Section 3 we characterized the cost of the strategy by the storage it implies. Considering the practical case of starting with known output values, a second-order consideration, given the choice, can be the reduction of forward computations. To find a "good" $S(G_A)$, the algorithm chooses inversion (option 1) whenever possible, then forward recomputation (option 2), and falls back on storage (option 3) only as a last resort.

Before we describe the option selection, we define how tokens propagate across flow dependency edges $e \in \mathcal{E}_f$. The token propagation is a simple process we call *saturation*, formally defined as $sat(G_A)$.

```
Algorithm $sat(G_A)$ :
  repeat:
    propagate forward along flow dependencies:
        $\mathcal{D}_f := \{v \in V : \neg T(v) \wedge \forall w : (w,v) \in \mathcal{E}_f, T(w)\}$
        $\forall v \in \mathcal{D}_f$ set $T(v) :=$ true
    propagate backward along flow dependencies:
        $\mathcal{D}_b := \{v \in V : \neg T(v) \wedge \{w : (v,w) \in \mathcal{E}_f \wedge T(w)\}$ is sufficient$\}$
        $\forall v \in \mathcal{D}_b$ set $T(v) :=$ true
  until $\mathcal{D}_f \equiv \mathcal{D}_b \equiv \emptyset$   $\square$
```

This definition of $sat(G_A)$ aims at addressing the possibility of ambiguous definitions caused by control flow and aliasing. Saturation iterates until there are no more nodes to which tokens can be propagated. Intuitively, a token propagates in the direction of flow edges to use node v if v doesn't have a token but all use-def chain predecessors of v have tokens. Likewise, a token certainly propagates in the opposite direction of flow edges to definition node v if v doesn't have a token but all def-use chain successors of v have tokens. However, because uses do not mask one another, we can relax the condition for backward propagation: in many occurrences, only a "sufficient" subset of the successors of v can be

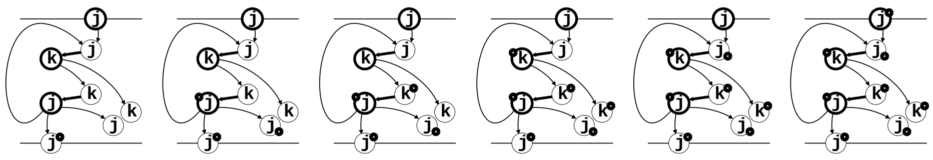

(a) (b) (c) (d) (e) (f)

Figure 6: Stages of $tg(G_A)$ for the initial G_A from Figure 5

enough to place a token on v. Specifically a subset is sufficient if the basic blocks that contain these uses post-dominate the basic block that contains the definition that is, all control paths flowing from the definition go through one of these uses. For instance in the frequent case where all uses and the definition belong to the same basic block, any single use is a sufficient subset. Because of the source transformation context we have to make conservative assumptions and account for all ambiguities. However, finding these sufficient subsets of uses just depends on the structure of G_A and of the control-flow graph. These subsets can be precomputed so that they add very little runtime cost to our algorithm.

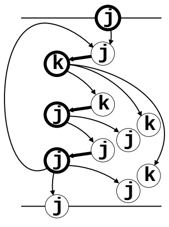
The next step is the actual choice of options in the token game algorithm $tg(G_A)$ where the initially known nodes v in graph G_A already have tokens $T(v)$ and we have classified the invertible computation dependencies $\mathcal{E}_I \subseteq \mathcal{E}_c$.

```
Algorithm $tg(G_A)$ :
  repeat:
    perform $sat(G_A)$
    if $\exists v \in \mathcal{V}$ where $\neg T(v) \wedge \exists (v,w) \in \mathcal{E}_I : (T(w) \wedge \forall (u \neq v, w) \in \mathcal{E}_c, T(u))$
      choose inversion to recover $v$ and set $T(v) :=$ true
    else
      if $\exists w \in \mathcal{V}$ where $\neg T(w) \wedge \forall (v,w) \in \mathcal{E}_c, T(v)$
        choose forward recomputation to recover $w$ and set $T(w) :=$ true
      else
          use heuristic $H(G_A)$ to select a $v \in \mathcal{V}$ where $\neg T(v)$
          choose storage to recover $v$ and set $T(v) :=$ true
  until $\forall v \in \mathcal{V}, T(v)$   $\square$
```

To illustrate the practical use of the algorithm, we go back to the loop in our example Figure 2(a). We assume we know the value of $\mathcal{V}_O = \{\texttt{j}\}$ (see Figure 6(a)) and perform $sat(G_A)$, resulting in two more tokens shown in Figure 6(b). Note that *sat* does not forward propagate through the loop-carried dependency. Now $tg(G_A)$ finds $v \equiv \texttt{k}$ as inversion target, we can generate the inversion code on line 4 in Figure 2(c) and place the token $T(\texttt{k})$; see Figure 6(c). In the repeat loop of $tg(G_A)$ we again perform $sat(G_A)$, which gives us two more tokens, as shown in Figure 6(d). Now we have tokens in both nodes representing the uses in the adjoint statements; that is, the code generator can now insert lines 5 and 6 in Figure 2(c). In $tg(G_A)$ we then find $v \equiv \texttt{j}$ as inversion target; see Figure 6(e). We generate line 7 in Figure 2(c), perform another $sat(G_A)$ in the

```
1     j=1
2     do while (j<6)
3      k=j+1
4      j=k+2
5      a[j]=a[k]*a[j]
4      j=j+1
5      a[k]=2*a[j]
6     end do
```

(a)



(b)

```
1     pop(j)
2     do while (j>=1)
3      pop(a[j])
4      t=j-1
5      k=t-2
6      ā[j]=ā[j]+ā[k]*2;
7      j=t
8      ā[k]=a[j]*ā[j]
9      ā[j]=a[k]*ā[j]
10     j=k-1
11    end do
```

(c)

Figure 7: Example code from Figure 2(a) with an overwrite of `j` and additional uses in (a), the corresponding G_A in (b), and the corresponding adjoint code using a temporary `t` for the inversion in (c)

$tg(G_A)$ repeat loop, and finish as all nodes have tokens; see Figure 6(f). Our inversion strategy for this simple example has a cost of zero. In practice the code transformation uses program variables that often will be overwritten as is the case here with `j`. Data dependence graphs typically represent overwrites with dependencies from variable uses to variable overwrites, known as antidependencies. For simplicity we did not show those in our examples. They are, however, easily bypassed by introducing temporary variables whenever the inversion strategy implies an overwrite before all uses are accommodated. These temporaries are single-assignment variables with a local scope limited to the adjoint sweep, in other words of very little cost when compared to the stack storage we aim to minimize. For example, if we modify Figure 2(a) slighly by inserting an overwrite of `j` and another use of `j` and `k`, see lines 4 and 5 in Figure 7(a) we still can accomplish the inversion by introducing a temporary `t` into the adjoint shown in Figure 7(c).

The heuristic $H(G_A)$, referenced in $tg(G_A)$, should select a node v that permits the most subsequent inversions and forward recomputations which in the following formal description are collected in sets $\mathcal{T}(v)$. The heuristic returns a v with $\mathcal{T}(v)$ of maximal size.

```
Heuristic $H(G_A)$ :
$\forall v \in \mathcal{V}$ in reverse topological order:
    $\mathcal{T}(v) := \emptyset$
    if $\neg T(v)$
        $\mathcal{T}(v) := \{v\}$
        $\forall (v,w) \in \mathcal{E}_c$:
            if $\forall (u \neq v, w) \in \mathcal{E}_c, T(u)$ (forward computation)
                $\mathcal{T}(v) := \mathcal{T}(v) \cup \mathcal{T}(w)$
        $\forall (v,w) \in \mathcal{E}_f$:
            if $\forall (u \neq v, w) \in \mathcal{E}_f, T(u)$ (forward saturation)
                $\mathcal{T}(v) := \mathcal{T}(v) \cup \mathcal{T}(w)$
        $\forall (w,v) \in \mathcal{E}_f$:
```

```
 1    l=0
 2    do while (l<ilB)
 3      k = i-j
 4      m = k+3+l
 5      i = k+2*l
 6      push(a[i]) // augm.
 7      a[i] = a[j]*p[k]*a[m]
 8      j = j+m+4
 9      i = i+j+1
10      push(a[j]) // augm.
11      a[j] = a[i]*p[k]*a[m]
12      l = l+1
13    end do
14    push(i,j) // augm.
```

(a)

```
 1    pop(j,i)
 2    l = ilB
 3    do while (l>=0)
 4      t = i-j-1; // 1:i
 5      l = l-1 // 2:i
 6      k = t1-2*l; // 3:i
 7      m = k+3+l; // 4:f
 8      pop(a[j])
 9      ā[j]+=ā[i]*p[k]*a[m]
10      ā[m]+=ā[i]*p[k]*a[j]
11      ā[i]=0;
12      i = t;
13      j = j-m-4; // 5:i
14      pop(a[i])
15      ā[j]+=ā[i]*p[k]*a[m]
16      ā[m]+=ā[i]*p[k]*a[j]
17      ā[i]=0;
18      i = k+j; // 6:i
19    end do
```

(b)

Figure 8: Loop L augmented with `push` statements for the forward sweep in (a) and its adjoint in (b)

```
        if $\{v\} \cup \{u : (w,u) \in \mathcal{E}_f \wedge T(u)\}$ *is sufficient* (backward saturation)
            $\mathcal{T}(v) := \mathcal{T}(v) \cup \mathcal{T}(w)$
return one of the $v$ for which $|\mathcal{T}(v)| = \max_{w \in \mathcal{V}} |\mathcal{T}(w)|$   □
```

Because inversion is not considered, this heuristic can be computed by a single sweep over G_A in reverse topological order. Figure 8 shows a loop a little more complex than that in Figure 2(a), which we will use to illustrate the role of forward computation and the heuristic. Figure 9(a) shows G_A for the loop body. For simplicity we left out the loop-carried dependencies and the extra use nodes for the adjoints of lines 7 and 11. G_A shows the known output values for `i`, `j`, and `l`. Figure 9(b) shows the state in the fourth iteration when no inversion is possible and we have to pick the forward computation of `m`. Figure 9(c) has the final state after the token algorithm runs to completion. Figure 8(b) shows the resulting statements for the address computations along with the actual adjoint statements for lines 7 and 11 of (a). Again no storage is required. However, if we change line 5 of Figure 8(a) to something noninvertible, for example `i = max (k, 2*l)`, then in the third iteration we have to resort to our heuristic H. It finds the definition node for `k` because its trigger set $\mathcal{T}(\mathtt{k})$ is the only one with the maximal cardinality 5. We place the token as shown in Figure 9(d).

To illustrate the amount of storage saved and the ensuing consequences for the adjoint computation, we wrap the loop L shown in Figure 8(a) into an outer loop that initializes `i` and `j` so that the accesses into array `a` are spread out; see Figure 10. The par-

```
o=0
do while(o<oLB)
   i=j=o
   L
   o=o+1
end do
```

Figure 10: Wrapper for loop L from Figure 8(a)

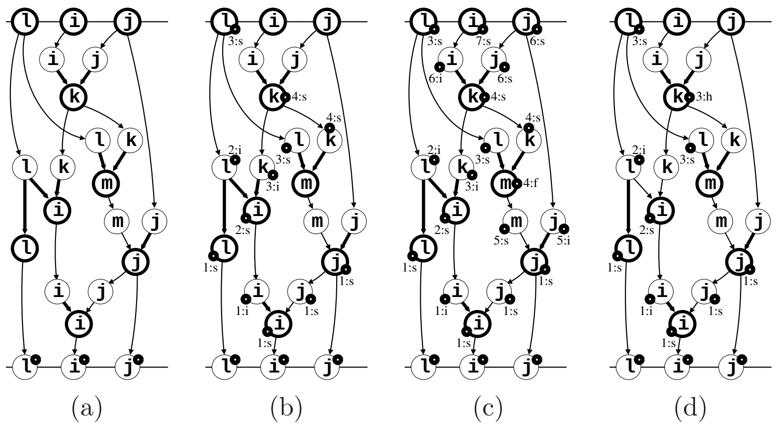


Figure 9: Performing $tg(G_A)$ for the example in Figure 8 at initial (a), intermediate (b) and final (c) state; a modified example uses the heuristic (d)

tial derivatives in the adjoint statements refer to the products `p[k]*a(m)`, `a[i]*p[k]`, and `a[j]*p[k]`, and the adjoint variables $\overline{\texttt{a}}$`(.)` need the indices `i`, `j`, `k`, and `m`. Simple forward recomputation of the indices has an effort $\mathcal{O}((\texttt{ilB} * \texttt{olB})^2)$. Storing all indices requires a memory size of $\mathcal{O}(\texttt{ilB} * \texttt{olB})$. A few experiments with `ilB` fixed to 10 show the practical effects on memory consumption in the following table:

	Time in μs			Memory			
`olB`	P	P_s	P_i	size of `a`	t_i in P_s	t_i in P_i	t_f
10^2	27	100	93	599	5200	$2 \cdot 10^2$	$2 \cdot 10^3$
10^4	2865	10672	9477	10499	5200000	$2 \cdot 10^4$	$2 \cdot 10^5$
10^6	297109	1,154878	1,133269	1000499	52000000	$2 \cdot 10^6$	$2 \cdot 10^7$
10^7	2,985838	-	11,612644	10000499	-	$2 \cdot 10^7$	$2 \cdot 10^8$

The integers stack size is t_i, the floats stack size is t_f. Already for a total computing time of less than 3 seconds run on an average laptop, the store-all approach (P_s) fails to allocate sufficient memory to contain all the required integer values for the indices, while the inversion-based approach (P_i) is still able to complete the adjoint computation because it requires an integer tape that is an order of magnitude smaller. Larger problem sizes (i.e., longer execution times) will cause the plain inversion-based approach to fail as well because of the float tape size and will require (hierarchical) checkpointing. The table shows no significant runtime savings in P_i over P_s because the dominating factor is the nonlocal access pattern to the vector `a` and, in the reverse sweep, its corresponding adjoint vector $\overline{\texttt{a}}$. However, when the lack of memory forces the use of hierarchical checkpointing, we will observe a significant increase in execution time. Consequently, any reduction in taping memory requirements will reduce the number of recomputations incurred by checkpointing and thereby result in runtime savings.

While our example is somewhat academic, it illustrates nicely the direct effect of the inversion approach on the efficiency of the resulting derivative computation.

5 Practical Examples

This section shows the occurrence of index computations, similar to the academic example in the previous sections, for practical uses. We emphasize once more that, aside from the quantitative savings, we aim at qualitatively characterizing the possibility of inversion in cases that are not amenable to the pattern-based approach used so far.

5.1 Irregular Meshes

Irregular meshes made of triangles or tetrahedra are more flexible than regular meshes. In particular, irregular meshes lend themselves easily to incremental mesh refinement or adaption, yielding more accurate results. With irregular meshes, access to a neighbor mesh element uses tables known as indirection arrays, rather than mere index offsets. For our work, this makes inversion the choice option only for loop indices, whereas forward recomputation is the option for computing neighbor indices. Figure 11 shows a typical loop from a 2D Navier-Stokes solver [11] running on irregular meshes. Loop indices `ic` and `iseg` are easily inverted, and index variables `nsg1`, `nsg2`, `nuor`, and `nuex` are cheaply recomputed forward from `ic`, `iseg`, and constant indirection arrays `icola` and `nubo`. For the complete Navier-Stokes solver corresponding to Figure 11 we measured the total memory traffic on the storage stack, that is, the number of bytes pushed on the stack while computing the gradient. With the default storage-based strategy, this is 3204 Mbytes. When inversion and forward recomputation are incorporated, traffic goes down to 2992 Mbytes. Runtime also improves, here by 8%, because recomputing index variables forward is actually much cheaper than storing them. Memory traffic and runtime would decrease further if recomputation was applied to cheap float variables such as `vx`, but this is beyond the scope of this paper.

```
DO ic=1,nca
  nsg1 = icola(ic-1)+1
  nsg2 = icola(ic)
  DO iseg=nsg1,nsg2
    nuor = nubo(1,iseg)
    nuex = nubo(2,iseg)
    vx=coor(1,nuex)-coor(1,nuor)
    vy=coor(2,nuex)-coor(2,nuor)
    dpex=f1(nuex)*vx+f2(nuex)*vy
    ...
    rh4(nuor) = rh4(nuor)+dpl
    rh4(nuex) = rh4(nuex)-dpl
  ENDDO
ENDDO
```

Figure 11: Code section from a 2D Navier-Stokes solver on irregular meshes

5.2 Ocean and Weather Modeling

The MIT general circulation model [18] is designed for study of the atmosphere, ocean, and climate. It consists of a number of source code packages each repre-

```
do ...
  kUp = 1+MOD(k+1,2)
  kDown = 1+MOD(k,2)
  ...
  *( fVerT(i,j,kDown)...
  ...
end do
```

(a)

```
DO j = max(jts,jbe-spec_zone+1), jtf
  b_dist = jbe - j
  b_limit = b_dist
  DO k = kts, ktf
    DO i = max(its,b_limit+ibs), min(itf,ibe-b_limit)
      i_inner = max(i,ibs+spec_zone)
      i_inner = min(i_inner,ibe-spec_zone)
      field(i,k,j) = field(i_inner,k,jbe-spec_zone)
    ENDDO
  ENDDO
ENDDO
```

(b)

Figure 12: Code sections with loop constructs from MITgcm (a) and WRF (b)

senting various aspects of the physics and chemistry processes in the ocean and atmosphere using regular discretizations. Therefore, most of the iteration loops follow a very regular pattern, with a few exceptions as shown in Figure 12(a). Without going into much detail of the particular purpose of the code, we see that an inversion of the loop variable combined with forward computation of `kUp` and `kDown` avoids storing the values of both of these variables. The amount of saved storage is two times the problem-dependent number of vertical layers times the number of invocations of that routine per time step where the time step is the natural choice for a checkpointed segment. Figure 12(b) shows an example of similar nature with an index computation that is found with minor variations throughout the source files in the Weather Research and Forecasting [26] model. Here too the savings in storage for just this loop can be expressed as the product of the number of iterations in each of the three nested loops for the `i_inner` index and additionally the recomputation of the loop bounds for the two inner loops.

5.3 Livermore Loops Examples

Both examples shown in Figure 14 exhibit complicated index computations that do not lend themselves to any pattern-based search. Using the token game algorithm, one will find that for example Figure 14 (a) we will need to store the final value of `ipntp` and the values of `ii` because of the integer division.

The data dependencies in Figure 14(b) are rather obscured by the control flow. However, aside from the loop variable `k` and `l`, we have `m`, `k2`, and `k3` loop-carried by the inner loop. While `k2` and `k3` lend themselves to inversion given their final values, we also see that `m` cannot be recovered in this fashion because the update at label `460` depends on the condition involving `m` in the line above. Therefore

```
DO k = k1, ktf
  k2 = ktf
  DO WHILE( z_base(k2) .gt. z00(k) )
    k2 = k2 - 1
  ENDDO
  if(k2+1.gt.ktf)then
    u00(k) = u_base(k2) + ( u_base(k2)
         ...
   ....
  endif
ENDDO
```

Figure 13: Example from WRF

```
ii = n;
ipntp = 0;
do {
  ipnt = ipntp;
  ipntp += ii;
  ii /= 2;
  i = ipntp - 1;
  for ( k=ipnt+1; k<ipntp; k=k+2) {
    i++;
    x[i] = x[k]-v[k]*x[k-1]-v[k+1]*x[k+1];
  }
} while ( ii>0 );
```

(a)

```
    DO 485 L= 1,Loop
      m= 1
405 i1= m
410 j2= (n+n)*(m-1)+1
    DO 470 k= 1,n
     k2= k2+1
     j4= j2+k+k
     j5= ZONE(j4)
     IF( j5-n ) 420,475,450
415 IF( j5-n+II ) 430,425,425
420 IF( j5-n+LB ) 435,415,415
425 IF( PLAN(j5)-R) 445,480,440
430 IF( PLAN(j5)-S) 445,480,440
435 IF( PLAN(j5)-T) 445,480,440
440 IF( ZONE(j4-1)) 455,485,470
445 IF( ZONE(j4-1)) 470,485,455
450 k3= k3+1
     IF( D(j5)-(D(j5-1)*
   .(T-D(j5-2))**2+(S-D(j5-3))**2
   .+(R-D(j5-4))**2)) 445,480,440
455 m= m+1
     IF( m-ZONE(1) ) 465,465,460
460 m= 1
465 IF( i1-m) 410,480,410
470 CONTINUE
475 CONTINUE
480 CONTINUE
485 CONTINUE
```

(b)

Figure 14: Examples from the Livermore Loops: Incomplete Cholesky Conjugate Gradient (a), Monte Carlo search loop (b)

we have to either store `m` in every inner loop iteration or store a Boolean value indicating whether the condition evaluated to true or false. This scenario is not uncommon. For instance, it occurs in the WRF code as shown in Figure 13, although one might argue that the update loop for `k2` is in principle inexpensive to recompute assuming that the array values in `z_base` and `z00` are available.

6 Summary and Outlook

The approach presented here has two main benefits. First, we introduce a criterion for loop inversions that is based solely on dependency information. It does not rely on any particular pattern for loop constructs with designated loop variables and restrictions on updates. Second, we show how the inversion operations can be combined with forward computations to reduce the amount of stored values, which in turn yields an efficiency gain for the adjoint computation. Because we rely on dependency information provided by a compiler-style source code analysis as input, practical concerns particularly about the possible aliasing of variables are covered. Replacing storage operations with inversions and recomputations is beneficial as long as the recomputations are "cheap", as is typically the case for address computations. In such circumstances (implying $\alpha \equiv 0$), the algorithm introduced here will *always result in an improvement* of the generated adjoint code in particular when it enables recovery purely by inversion. It can therefore be generically applied to the adjoint transformation engine and does not require validation by runtime measurements.

If one considers more expensive recomputations, e.g. involving the reexecution of loops as suggest by the last example in Section 5, then the problem lies with finding the proper $\alpha > 0$. Further research will concentrate on the question what categories of more complex recomputations, in particular cases involving subroutine calls and loops can be recognized at transformation time and be included while assuming $\alpha \equiv 0$. Another part of our ongoing research aims at integrating the token game algorithm into an interprocedural framework.

Acknowledgements

Jean Utke was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357.

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] Autodiff.org portal. `http://www.autodiff.org`.

[3] M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series. SIAM, 1996.

[4] M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, volume 50 of *LNCSE*, Berlin, 2005. Springer.

[5] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, Computer and Information Science, New York, 2002. Springer.

[6] G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series. SIAM, 1991.

[7] R. Dembo and T. Steihaug. Truncated-newton algorithms for large-scale optimization. *Math. Prog.*, 26:190–212, 1982.

[8] C. Faure and U. Naumann. Minimizing the tape size. pages 293–298. Chapter 34 in [5].

[9] R. Giering. Tangent linear and Adjoint Model Compiler, Users manual. Technical report, 1997. http://www.autodiff.com/tamc.

[10] R. Giering and T. Kaminski. Recomputations in reverse mode AD. pages 283–291. Chapter 34 in [5].

[11] M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics*. SIAM, Philadephia, 1998.

[12] A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*, volume 19 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, 2000.

[13] A. Griewank. A mathematical view of automatic differentiation. In *Acta Numerica*, volume 12, pages 321–398. Cambridge University Press, 2003.

[14] L. Hascoët, U. Naumann, and V. Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Comp. Syst.*, 21(8):1401–1417, 2005.

[15] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004. `http://www.inria.fr/rrrt/rt-0300.html`.

[16] A. Jameson. Aerodynamic design via control theory. *Journal of Scientific Computing*, 3:233–260, 1988.

[17] F.-X. le Dimet and O. Talagrand. Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects. *Tellus*, 38A:97–110, 1986.

[18] MITgcm. `http://mitgcm.org`.

[19] S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, San Diego, 1997.

[20] U. Naumann. On optimal dag reversal. Technical Report AIB-2007-05, RWTH Aachen, March 2007.

[21] G. Ostrowski, Y. Volin, and W. Borisov. Über die Berechnung von Ableitungen. *Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna Merseburg*, 13:382–384, 1971.

[22] Dmitri Shiriaev. *Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment*. PhD thesis, Institute für angewandte Mathematik, Universität Karlsruhe, 1993.

[23] 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. Technical Report AIB-2007-14, RWTH Aachen, July 2007. Submitted.

[24] M. Wolf and M Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE transactions on parallel and distributed systems*, 2(4):452–471, 1991.

[25] M. Wolfe. Beyond induction variables. In *Proceedings of ACM Sigplan Programming Languages Design and Implementation*, 1992.

[26] WRF. `http://www.wrf-model.org`.