Programming language features, usage patterns, and the efficiency of generated adjoint code

Laurent Hascoët, Jean Utke

To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–20

Programming Language Features, Usage Patterns, and the Efficiency of Generated Adjoint Code

Laurent Hascoët[a*] and Jean Utke[b]

[a] *Team Ecuador, INRIA Sophia-Antipolis, France*
[b] *Allstate Insurance Company, Northbrook, IL, USA*

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

The computation of gradients via the reverse mode of algorithmic differentiation is a valuable technique in modeling many science and engineering applications. This technique is particularly efficient when implemented as a source transformation, as it may use static data-flow analysis. However, some features of the major programming languages are detrimental to the efficiency of the transformed source code. This paper provides an overview of the most common problem scenarios and estimates the cost overhead incurred by using the respective language feature or employing certain common patterns. An understanding of these topics is crucial for the efficiency or even feasibility of adjoint computations, particularly for large-scale numerical simulations e.g. in geosciences. While one cannot hope to cover all effects observable with a given programming language in a given run time environment, the paper aims at providing a reasonable guide for the users of C/C++ and Fortran source transformation tools for algorithmic differentiation.

Keywords: algorithmic differentiation, automatic differentiation, adjoint computation, source transformation

AMS Subject Classification:

1. Introduction

Computing derivatives of a numerical model $\boldsymbol{f}: \boldsymbol{x} \in I\!R^n \mapsto \boldsymbol{y} \in I\!R^m$, given as a computer program P, is an important but computation-intensive task. Algorithmic differentiation (AD) [7] in *adjoint* (or *reverse*) mode provides the means to obtain gradients and is used in many science and engineering contexts (refer to the conference proceedings [2, 5] and to [1]). The most important advantage of adjoint AD is the ability to compute machine precision gradients at a cost a few times that of the model evaluation itself and *independent of the size of the input space* with respect to which the gradient is computed. Therefore it is an enabling technology for large sensitivity studies, state estimation and other applications requiring gradients with respect to large input spaces.

Two major groups of AD tool implementations are operator overloading tools and source transformation tools. The latter are the focus of this paper. As a simplified rule, if P executes at run time a sequence of p intrinsic floating-point operations (e.g., addition, multiplication, exponential, sine, cosine)

$$[\ldots, j : (\ u = \phi(v_1, \ldots, v_k)), \ldots], \quad j = 1, \ldots, p, \tag{1}$$

*Corresponding author. Email: laurent.hascoet@inria.fr

in which we singled out one such operation ϕ depending on k arguments v_k, then the generated adjoint code has to implement the following sequence that reverses the original sequence in j:

$$[\ldots, j : (\ \bar{v}_1 \mathrel{+}= \frac{\partial \phi}{\partial v_1}\bar{u}, \ldots, \bar{v}_k \mathrel{+}= \frac{\partial \phi}{\partial v_k}\bar{u}, \bar{u} = 0), \ldots], \quad j = p, \ldots, 1, \tag{2}$$

with incremental assignments of each adjoint variable $\bar{v}_k$. Each $\bar{v}_k$ is initially set to zero. If the dimension of the output space m is 1 and we set $\bar{\boldsymbol{y}} = 1$, then the adjoint code yields $\bar{\boldsymbol{x}} = \nabla \boldsymbol{f}$. As computing the $\frac{\partial \phi}{\partial v_i}$ in (2) may use values v_i from (1), these two formulas result in two successive phases, illustrated in Fig. 1. The above formulas are given here only to provide a formal basis for the goal of the transformation and to illustrate the reason for the semantic transformations we consider in this paper. Note that aspects of these transformations are used also in contexts other than AD such as program slicing, automatic generation of application specific fail-over checkpoints, and back tracing. To implement the semantic equivalent of (2), adjoint source transformation AD employs four major building blocks that distinguish it from a compiler:

- **Specific data-flow analysis:** All source transformation AD requires a central data-flow analysis that finds out whether a given program variable at a given point in the execution is *active*, i.e., has a nontrivial derivative that is needed for the final desired derivative result. Activity is a data-flow property defined naturally only on variables of continuously differentiable type such as `float, double` (in C/C++), or `real` in Fortran. Certain programming language features and usage patterns obfuscate the transmission of these values see (Sec. 2.4) or make the analysis grossly imprecise[1] (see Sec. 2.1 through Sec. 2.4). Adjoint AD also requires another specific data-flow analysis for trajectory restoration (see below) that is similar to read-write or liveness analysis, both commonly found in compilers. While a good implementation of the analysis is neither harder nor easier to achieve than in a compiler, the unavoidable conservative overestimations will have more severe consequences on the very efficiency of the adjoint code, which we characterize in Sec. 2. Large-scale applications are often implemented as parallel programs by using MPI [8]. AD data-flow analysis must accept MPI calls without loosing too much precision; see [9].
- **Control- and data-flow reversal:** As part of the sequence reversal, adjoint code must reverse the flow of control of the given code. In other words, the transformed source code must at run time record sufficient information on the path through the control flow graph so as to drive the flow of control for the reverse sequence. The code that actually computes the adjoint values is therefore called the *reverse sweep*. However, reversal of some patterns in the control flow graph can be hard to implement and expensive to execute. We discuss typical cases in Sec. 3.1. Even at the elementary level of a single statement, the derivatives computation reverses the data-flow order of the original statement (cf. the incremental updates of the adjoint arguments of ϕ in (2)). As a reversed tree is not a tree, temporary variables must be introduced, causing problems when their type is nonscalar; see Sec. 3.2. Data flow induced by parallel message passing is a common usage pattern and has to be reversed as well [10].
- **Trajectory restoration:** The partial derivatives $\frac{\partial \phi}{\partial v_i}$ in (2) use values computed by the original (forward) computation (1), in the reverse of their computation order. Since most imperative programs reassign their variables, the intermediate values (the *trajectory*) are lost and must be restored. This can be done by running the forward code

[1] while still conservatively correct

repeatedly from a stored initial point (e.g., at $j = 1$) to the currently needed point, but this is nontrivial and raises problems similar to *program slicing* and *checkpointing*, described in the next item. Alternatively, one may execute (1) in a *forward sweep* that pushes the intermediate values on a stack before they are overwritten. While executing (2) the values are popped from the stack as needed to compute the partial derivatives. The stack size evolves in time as shown in Fig. 1. To reduce the stack size, AD tools

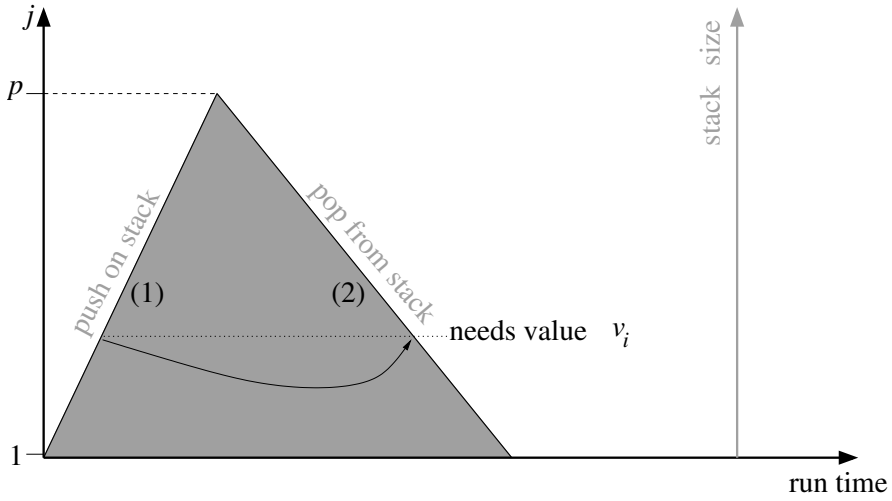

Figure 1. Phases (1) and (2) of the adjoint computation, showing the values dependencies between them and the evolution of the stack that carries these values. Phase (1) pushes values v_i and phase (2) pops them.

run data-flow analysis to reduce the set of values to be restored. For convenience, the recording of the control flow path is considered a part of the trajectory reversal, as is the recording of computed addresses for memory references that use indices and pointer values. Details are given in Sec. 3.3 and Sec. 4.

- **Checkpointing:** A crucial building block for adjoint computations of large-scale applications is the restart of (1) from certain checkpoints. The stack that stores the trajectory, typically held in main memory for efficiency, grows linearly with the execution time of P and therefore quickly outgrows all realistic main memory. To reduce the stack size, one stores the trajectory only for a small subsequence $j = r, \ldots, r+t$ of (1) at a time, immediately followed by the corresponding subsequence $j = r+t, \ldots, r$ in (2) that uses the stored trajectory and releases the memory. See Fig. 2 for an illustration on a simple example. For the entire computation of (2), this implies taking checkpoints to enable restarts at certain j. In Fig. 2 we assume a very simple equidistant placement of four checkpoints with a shorter remainder section for j near p and the assumption that initial state can be restored from program data without an extra checkpoint. Large problems will require hierarchical checkpointing implying additional recomputations. With a special checkpoint placement scheme [6] both the run time ratio (adjoint on original) and the storage requirement can be kept growing only by a factor logarithmic in the original run time. Yet, some subsequences have to be executed multiple times, implying restrictions detailed in Sec. 5. Checkpoints can be stored on slower secondary storage such as disk and thus provide a good tradeoff between execution time and memory consumption, especially when they are kept small by specialized data-flow analysis [4]. It is important that the reverse differentiated code correctly reads and writes the appropriate subset of program variables identified by this data flow analysis. Details are given in Sec. 4.

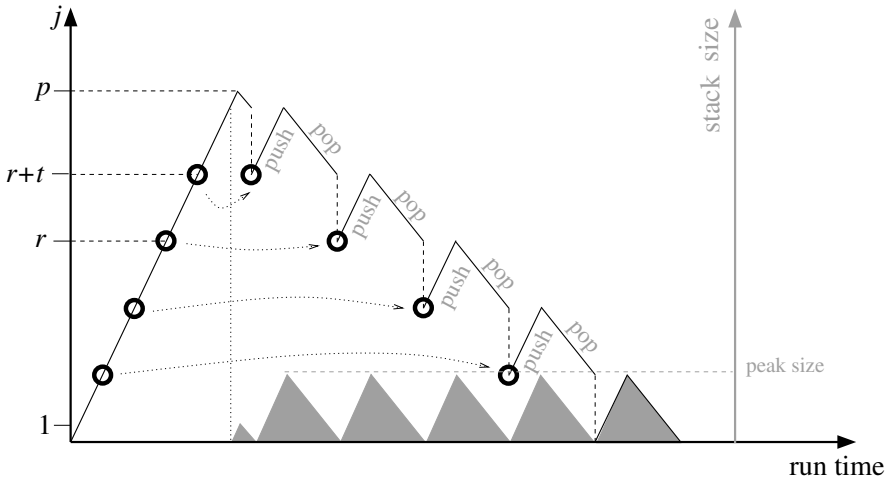

Figure 2. Stack size reduction compared to Fig. 1 by recomputation from four checkpoints (depicted by the circles).

The following sections describe how commonly used language features and usage patterns can impede the function of these building blocks, thereby reducing the efficiency of the adjoint computation or even making it impossible. Therefore, the essential justification for our recommendations for AD tool users will be qualitative improvements to the efficiency of the generated adjoint code. More precisely, our objective is to produce AD code that is maintainable and easy for the compiler to optimise, rather than efficient per se. However this cannot be quantified because many compiler optimization algorithms are not divulged. Even if their relative importance strongly depends on the structure of the numerical model in question, the recommendations given in this paper for the use or avoidance of various patterns and language features have a clear qualitative impact on the efficency. While we use C, C++, and Fortran as examples, all the explanations could be formulated in terms of abstract programming language concepts and applied to other programming languages where these concepts are relevant. The paper is structured to minimize forward references. In Sec. 2 we describe how certain idioms impact the efficiency by diminishing the accuracy of the data flow analysis and in Sec. 3 we look at their impact on the reversal of control and data flow. The effects of certain choices in an application's data model, the effects of scoping and the use of dynamic resources are described in Sec. 4 and Sec. 5. In Sec. 6, we recapitulate the recommendations in the terms introduced in the preceding sections.

2. Impact of Data-Flow Analysis on Activity and Trajectory Restoration

A staple of AD source transformation (and compiler optimization) is data-flow analysis. To ensure semantic correctness e.g. in the presence of aliasing, any data-flow analysis will incur a certain overestimate. Aliasing describes the possibility that a given storage location is referenced by multiple distinct syntactic elements (e.g., pointers or C++ references). Considering the main differentiation-specific data-flow analyses, overestimation can increase both the wall clock run time and the memory consumption as described below. As these analyses are chained in the indicated order, overestimation can occur inside one particular analysis but can also have a compounding effect from overestimates

in a previous one.

(i) Activity analysis detects, for each location in the code, the derivatives that are always null or useless for the differentiation goal. Consequently some statements in (2) can be simplified, yielding a faster code. Moreover, no memory is needed for derivatives of variables that are never active anywhere in the code, thus reducing memory use.

(ii) Differentiable-Liveness detects that some computation in (1), although possibly live for the original code, is not live for phase (2) because it computes no value used either in (2) or in the differentiably-live sequel of (1). If only derivatives are requested, this code can be removed from (1), yielding a faster code. Obviously this analysis will not perform as well if overestimates in activity have left unnecessary code in (2).

(iii) To-Be-Restored analysis detects that some values about to be overwritten in (1) need not be stored for Trajectory Restoration because they are not needed in (2). This reduces the memory used by the trajectory stack. It also reduces the memory traffic between the differentiated code and this stack, thus yielding a faster code. Obviously these benefits will be degraded if activity has left unnecessary code in (2) thus needing more values from (1), or if Differentiable-Liveness has left unnecessary statements in (1) that overwrite needed values.

(iv) Detection of the values to store at a checkpoint partly depends on the results of To-Be-Restored analysis. Overestimation will again lead to storing larger checkpoints, thus requiring more memory and also yielding a slower code.

Table 1 shows the magnitude of the performance overhead incurred by degradation of the differentiation-specific data-flow analyses. Precise measuments would be illusory here, as too much depends on the structure and style of the numerical model: table 1 compares the run time and memory use of the adjoints of a few codes of varied sizes, with the differentiation-specific data-flow analyses turned on and off. The table thus gives an upper bound of the performance overhead incurred by data-flow overestimates. Note however that this upper bound can very well be reached in practice, as several forms of aliasing can completely ruin the accuracy of analyses. One can observe that a run time overhead factor of two is not uncommon.

	#lines	T_o	T_{adj}	Traffic	Peak stack
lidar	879	0.3	0.6 → 1.8	6 M → 1.2 G	6 M → 1.2 G
1D	896			55 K → 150 K	49 K → 82 K
opticurve	1029	0.3	0.9 → 1.3	12 M → 93 M	0.6 M → 1.4 M
burgers	1060	2.2	3.8 → 4.6	0.4 M → 1.4 M	0.4 K → 1.2 K
nsc2ke	3533	0.3	2.7 → 4.2	2.9 G → 5.5 G	176 K → 176 K
multisail	4014	2.8	11.1 → 15.1	23 M → 560 M	1.6 M → 3.2 M
uns2d	5279	1.2	6.5 → 10.5	1.3 G → 12 G	253 K → 763 K
lagoon	6835	0.5	36.8 → 42.5	110 G → 116 G	809 M → 851 M
winnie	12877	1.6	5.6 → 13.7	0.6 G → 3.1 G	0.4 G → 1.3 G
sonicboom	21435	2.0	5.0 → 10.6	5.4 M → 34 M	1.5 M → 18 M
margaret	24033	1.5	11.7 → 18.1	2 K → 10 K	

Table 1. Effect of turning off Activity, Differentiable-Liveness, and To-Be-Restored analyses on various application codes. The first two columns characterize each code with its size in lines and original function run time (T_o seconds). The next columns show the overhead in adjoint run time (T_{adj} seconds), total memory push/pop Traffic (K-,M-, or G-bytes), and Peak stack memory use: for each measure we compare the values with analyses turned on, then off. Blank cells indicate measurements too small for comparison.

To provide any general quantitative estimate of the efficiency penalty we have to make

some abstractions. Typically the overestimate in any of the data-flow analyses is expressed as sets of variables or values for which a property may (but not necessarily must) hold. The two dominating effects are related to the activity analysis and the data flow analysis that prepares the trajectory restoration. Let O_a^c be the compile-time (static) activity analysis overestimate expressed as a set of program variables to be active and let $|O_a^c|$ be the set size. Depending on problem parameters, let O_a^r be the runtime overhead (induced by the activity analysis overestimate) expressed as a set of *scalar* program variables to be active and $|O_a^r|$ the maximal set size during the execution of the program. Because of dynamically sized arrays in the program, the addition of a single array variable to O_a^c may cause a large change in $|O_a^r|$. Note that $|O_a^r|$ can be affected by recursion as well.

Let O_t^c be the overestimate of the data flow analysis geared at trajectory restoration expressed as a set of assignment statements overwriting values that need to be saved to restore the trajectory, and let $|O_t^c|$ be the set size. Let O_t^r be the corresponding runtime overestimate expressed as set of *scalar* values to be stored and restored, and let $|O_t^r|$ be the set size. Because of loops, array assignments, and recursion, the addition of a single assignment to O_t^r may cause a large change in $|O_t^r|$.

Other than monotonicity in the relations between $|O_a^c|$ and $|O_a^r|$ (and $|O_t^c|$ and $|O_t^r|$, respectively) no statement about the nature of the dependency can be made in the general case, and one always has to assume **small increases in the compile-time analysis overestimate can result in large increases in the runtime overhead**. However, excluding effects from recursion one can state that, depending on the problem parameters, $|O_a^c|$ **is proportional to the original program memory footprint** and typically $|O_t^r|$ **is proportional to the original program execution time**. With this in mind we use O_a^r and O_t^r for the following statements about the efficiency penalty. For some applications the analyzed properties (i.e. activity, restoration need) semantically hold for all program data. Then there is no room for overestimation. Still, for numerical models that have a complicate (large) source code base and are also computationally complex, the derivative computations are often limited to certain input and output variables under the assumption that a tight analysis leads to minimized computational cost for the generated derivative code (informed by activity analysis) and minimized effort to restore the trajectory. Because of checkpointing one can trade temporal for spatial complexity, but here we assume a fixed checkpointing scheme. The overhead components characterized below are cumulative.

- The **overestimate O_a^c causes a spatial overhead in the program memory proportional to** $|O_a^r|$ because of the association of derivative values with program variables and the thereby increased memory footprint of the program.
- The **overestimate O_a^c causes a temporal overhead in the direct computation of derivative values from the restored trajectory that is at least linearly proportional to** $|O_a^r|$. Not all operations executed in the program have the same cost. The temporal overhead will be larger if the unnecessary derivative computation triggers particularly costly operations such as solving an equation system.
- The **overestimate O_t^c causes a spatial overhead for the trajectory restoration proportional to** $|O_t^r|$, as well as some temporal overhead, proportional to $|O_t^r|$, due to the extra push/pop.

2.1 *Aliasing by Pointers, References, and Parameter Passing*

Aliasing is difficult to detect even when all the code involved is exposed to the tool. Even

more when differentiating a library for example, the calling context is not known and analysis tools generally assume that this external context will not introduce aliasing by passing actual arguments with overlapping storage locations to distinct formal arguments of library primitives. We therefore **recommend that the code to be differentiated is not used in a calling context that introduces aliasing**, and in the sequel we assume this is the case.

Pointers are an obvious source, but not the only source, of aliasing. In almost all numerical models aliasing is unavoidably introduced via array subscripts. The question whether for example the C expressions `array[i]` and `array[j]` refer to the same memory location is generally undecidable at compile time for runtime-computed indices `i` and `j`. Array section analysis [3] may be a partial remedy, but it is currently not implemented in AD tools and is not useful for additional levels of indirection as in `array[indexTable[i]]` often found in models with unstructured meshes.

An advantage of the variable-length array concept in Fortran is that a dereference `a(i)` has to remain within the storage associated with a plain (i.e., not a `pointer`, `allocatable`, `equivalence`, formal parameter, or blank `common`) array variable `a`. It cannot itself alias a reference `b(j)` to another array `b`. A plain variable can be aliased by another variable only by explicitly being qualified as `target`. Because C++ lacks a built-in variable-length array type, one has to rely on pointers and pointer arithmetic. Making the pointer constant via an instantiation such as

double ∗ **const** a = (**double** ∗ **const**)malloc(n∗**sizeof**(**double**));

prevents the reassignment of `a` but does not prevent its address value from being assigned to another pointer. In Fortran, while an `allocatable` can be reallocated, it cannot have its address value assigned to a pointer variable unless also declared to be a `target`. Consequently, no exclusion of aliasing based on syntax can be made for C/C++[2]. Instead, one has to use a pointer analysis that recognizes the instantiation of the `const` pointer with as-of-yet unaliased heap memory. If `a`'s value is assigned to another pointer variable the door to aliasing is open.

To be conservatively correct, data flow analysis must propagate properties to all variables that might alias each other, thus overestimating most sets of variables of interest. Consequently optimization of the differentiated code is less powerful and retains more unnecessary code; see also Sec. 4.1. If pointers are used to reference active data (with computed addresses), their values become part of the trajectory restoration. Consequently, **a high number of (active) pointer values and their possible aliases being overwritten or going out of scope increases the overall spatial and temporal overhead**. A related issue is the validity of a restored pointer value when it refers to potentially deallocated dynamic memory; see Sec. 3.3.

A more benign form of aliasing occurs when the aliasing entity has a scope that is contained in the scope of the aliased entity and the aliased entity is a stack variable or a heap variable that does not go out of scope during the adjoint computation. The typical cases are aliasing between formal and actual parameters in subroutine calls and C++ reference variables directly referring to stack variables. Here, no explicit storing and restoring of a pointer value (virtual address) is required. Instead, one can rely on semantically correct reinstantiation of the aliasing entity (reference variable) during the reverse sweep by referring to the same syntactic representation of the aliased entity. The difference caused by not instantiating with a stack variable is shown in Fig. 3, where the instantiation of `dr2` as on line 4 requires restoring the value of pointer `dp` (imagine

[2]Fixed-size arrays such as `double a[3],b[4]` are more straightforward.

```
1  double d(2.3),*dp(0);
2  double &dr1=d; // simple
3  dp=&d; // "computed" address
4  double &dr2=*dp; //not simple
```

Figure 3. Code snippet for C++ reference variables aliasing a static address and a computed address.

something complicated for line 3) while the instantiation of `dr1` on line 2 does not; see also Sec. 3.3. The conclusion is to **avoid pointers whenever possible**. A programming style that uses data structures as global or stack variables at the top level and passes only references to these variables aligns the scope of the data structures with the syntactic scopes of blocks and subroutines, thereby reducing the need for pointers.

Another old-style but commonly used Fortran idiom is the passing of arrays by *storage association* (think pointer), as illustrated in Fig. 4. The array in the call always appears

```
1  subroutine foo(a)
2    real :: a(5)
3    a=4.2
4  end subroutine
```

```
1  subroutine bar(b)
2    real :: b(2,2,2)
3    b=42
4  end subroutine
```

```
1  program p
2    real :: x(2,4)
3    x=reshape((/1,2,3,4,5,6,7,8/),(/2,4/))
4    call foo(x(1,2)) ! columns 2 - 4
5    call bar(x(1,1)) ! whole array
6  end program
```

Figure 4. Examples of Fortran parameter passing by storage association

as a scalar, but a pointer is passed and the (remaining) array implicitly reshaped. In particular, the syntax on the caller side provides no indication about the dimension of the array received on the callee side. This makes it hard to determine whether all values of an array passed to another routine are completely overwritten by that routine leading to overestimates in the data flow analysis for trajectory restoration. In our example, we can see by inspection that the call to `bar` overwrites all of `x`, but the call to `foo` overwrites only columns 2 and 3 and the first element of column 4. Only the introduction of interfaces and the enhanced Fortran array syntax allows for an unambiguous representation of array slices and a tighter data-flow analysis and compiler optimization. We therefore **recommend the use of interfaces and enhanced array syntax over parameter passing by storage association**.

In Fortran, associating dynamically allocated memory to variables is permitted only when they are qualified as either `pointer` or `allocatable`. An important difference between the two is that a Fortran pointer variable can also point to other program variables (that have been qualified as `target`, thereby introducing aliasing) while an allocatable variable cannot. A simple rule of thumb is to **avoid the use of `pointer` variables in Fortran where possible in favor of `allocatable` variables**. Practical experience with modeling codes has shown that there are many scenarios where pointer

variables could be replaced by allocatables, not just benefitting the transformation but also compiler optimization. A very precise pointer analysis may help the transformation but its results will still be subject to conservative assumptions for problems that are undecidable at compile time.

2.2 *Multiple Types and Single Storage*

Legacy codes often use language idioms that save memory by allowing otherwise unrelated variables to share some storage space. This originates from early hardware where memory was scarce and available only through static allocation. Examples are `union` in C/C++ and `equivalence` in Fortran; see Fig. 5. One storage space can represent

```
1  real :: a(3)
2  integer :: i(3)
3  equivalence(a,i)
```

```
1  union {
2    int i;
3    double d;
4  } u;
```

Figure 5. Code for Fortran `equivalence` (top), and C/C++ `union` (bottom)

one of a prescribed set of data types at any one time. The selection of the intended type is done by the programmer, for instance by encapsulating a discriminator for `union` types, but in practice no compile time analysis can determine which type a given storage space use is supposed to represent. Assuming type consistency can be enormously helpful for activity analysis by filtering out right away all program variables that are not of floating-point type. Type consistent use means that the `equivalence` statement could be removed or the `union` turned into a `struct` and the program remains semantically correct. However no static analysis can guarantee type consistency in general. For conservative correctness, data dependence analysis therefore has to propagate across all references to the given (shared) storage space and will likely enlarge the overestimate. In many cases other solutions exist that provide type consistency for the numerical data. Therefore **we recommend avoiding `union` and `equivalence`**.

We point out that C++ class hierarchies using virtual functions (or explicit casting) from base class pointers also fall into this category. Assume that one implementation of a given virtual function changes data, e.g. in Fig. 6, `foo` in class `CC1`. Then the conservative assumption is that a call to any of these virtual functions via a base class pointer may change the union of all data changed by any implementation. Given that virtual functions are one of the basic object oriented techniques it would be hard to recommend their avoidance. Rather, we recommend to encapsulate logic that does not modify data into a separate hierarchy of virtual functions that then could have the `const` qualifier, thereby enabling compiler optimizations. Again, one may say this is just a tenet of good design but the consequences of not adhering to it can be more pronounced for the generated adjoint code than is the case for compiler optimization in general.

2.3 *Big Work Array*

The usage pattern of packing all data into a single, large work array is typically justified by the desire to control the memory use for data or to pass around all needed data via

```
1  class BC {
2  public: virtual float foo(float x) = 0 ;
3  };
4
5  class CC1 : public BC {
6    float v = 1.0 ;
7    float foo(float x) {v = v*x; return v*x;}
8  };
9
10 class CC2 : public BC {
11   float foo(float x) {return 1.5*x;}
12 };
13
14 void func(BC *ob, float *x) {
15   *x = ob->foo(*x); //turns *x active
```

Figure 6. C++ code snippet with activity overestimation due to a virtual function

```
1  subroutine foo (r)
2    real :: r(3)
3    do i=1,3
4       r(i)=i*1.0
5    end do
6  end subroutine
7
8  program p
9    common /c/ c1, c2, c3
10   real :: r(1)
11   equivalence (c1,r)
12   call foo(r)
13   print *,c1,c2,c3
14 end
```

Figure 7. Dummy array equivalenced to common block.

one syntactic entity. The lack of derived types in the old Fortran standards has certainly contributed to the pattern's proliferation. Numerical values representing different physical data would be accessed via certain agreed-upon offsets. Should activity analysis detect certain values in the work array to be active, then by the same rationale explained in Sec. 2.1 (Pointer Aliasing) the entire work array would be made active.

A Fortran feature expressly supporting such a pattern is the *blank common block*. The Fortran standard prescribes no restrictions regarding type or size consistency between references to the blank common block. Consequently, there are usage scenarios for which a correct adjoint source transformation would require propagation of activity through variables of a non-differentiable type, when they are bitwise identical views of storage space also accessed by active variables. Another frequent Fortran implementation of the pattern is the use of an equivalence to the beginning of a (named) common block, for example, for initialization as illustrated in Fig. 7. Because of the equivalence, all elements of the common block `c` become active if any of its elements are detected to be active. **We recommend avoiding the "big work array" pattern**.

2.4 *File I/O, Interfaces, Casting, and Obfuscated Dependencies*

Data dependence analysis within compilers (and source transformation tools) relies on following program variables and the values they carry. Values can, however, also be transferred by other means such as writing to and subsequent reading from files. Another example is string-based interfaces where model components exchange values via large dictionaries whose keys are agreed-upon strings. These cases can be seen as an extension of the big work array pattern. Data is exchanged via some container, and the access is done with a key (such as file names, interface strings, offsets). Frequently such containers are modeled as something equivalent to `void` pointers, and values have to be cast to the correct data type. For modular numerical models an exchange of data by the above means is not uncommon. In general, the automatic tracing of data dependencies described through a lookup of values via key is impractical because keys can be computed at run time and casting removes any hints on type matching the analysis could otherwise use to narrow the dependencies. The conservative fallback solution is therefore to assume that any data read from any container (file or interface container) depends on any data written to the container, causing an obvious but unavoidable overestimation.

Therefore, **avoid unstructured interfaces, casting and transfer of intermediate data via files** if possible. The effort required to replace such interfaces and file-based data transfers may sometimes render this recommendation impractical. Since the analysis expects dependencies that are traceable in terms of program variables, one can temporarily replace the file I/O with stub methods that exchange data via assignment to stubbed global variables. In the differentiated code these stubs can then be reverted to the original file I/O (with adjustments to carry along the derivative data). Similarly for interfaces, a user-provided code modification (possibly automated) could extract the agreed-upon keys, build a proper structure with appropriate types to avoid casting, and replace the key-based lookup methods with direct access to the structure elements. Accomplishing the above is easier when there are no computed keys. **The less structure the interface exhibits, the harder an exact analysis of the data dependencies becomes**.

As AD tools progressively become able to analyze MPI-parallel source, the obfuscated dependency problem extends to values communicated across processes. The abstract interpretation mechanism used by static analysis to propagate data-flow properties will be accurate only if matching send/receive pairs can be identified. Otherwise, the conservative fallback is that one active send may make all received values active. Message-passing libraries generally provide ways (communicators, tags...) that can help detect non-matching send/receive pairs, therefore reducing overapproximation in data-flow analysis. Therefore, we recommend to **use message-passing primitives so as to maximize static separation of groups of sending and receiving endpoints**.

3. Impact on Reversal of Control Flow and Data Flow

In contrast to the impact on the code analysis, the use of some features may prevent semantically correct control- and data-flow reversal altogether.

3.1 *"Not-So-Structured" Control Flow*

Compiler builders have been strongly recommending structured control flow for many years, because it permits better code optimization. Old-style Fortran constructs, such as an alternate `entry` into a subroutine, are discouraged. Generally, every sort of alternative

entry into a control structure (e.g., jumps into the body of a conditional branch or, even worse, jumps into a loop body) is either prohibited or permitted only under specific circumstances, among other reasons because of the difficulty in orchestrating variable instantiation in a meaningful fashion.

The situation is different for alternate exits from control structures, such as loop exits,

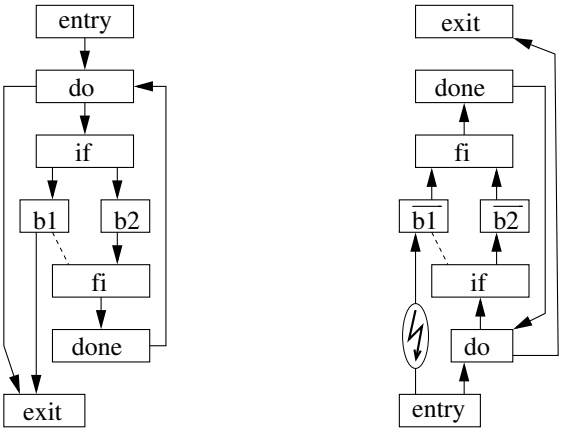

Figure 8. Control-flow graph with early return (left) and its naive, syntactically incorrect reverse (right).

which are still considered an acceptable practice. However, alternate exits can degrade loop data-dependence analysis and prevent automatic loop parallelization. Modern languages encourage use of an exception mechanism mostly for a graceful error recovery, but exception handling requires the ability to unwind the stack through more paths than during regular execution and thereby can block several compiler optimizations.

Control flow reversal implemented in a naive way amounts to relabeling the control flow vertices (entry↔exit, do↔done, if↔fi, etc.), inverting the edge direction, and adjusting loop control edges as shown in Fig. 8. The early return from branch b1 to the exit (left) implies, however, in the reversed control flow graph (right) a possibly forbidden jump from the new entry into the nested control flow structure to $\overline{\text{b1}}$ (*irreducible* loop). The same logic applies also to the common loop control statements such as `break`. Symmetrically well-structured control flow[3] can be reversed while retaining the control flow structures and the efficiency they offer. Otherwise a fallback solution is to record the sequence of basic blocks executed at run time and replay this sequence in reverse, switching on the recorded block identifier to decide on the adjoint basic block. This fallback solution is in general possible only for Fortran, because this "flattening" approach cannot replicate the nested scopes possible in C/C++ control flow graphs when these nested scopes contain variable declarations[4]. This fallback solution obviously removes any chance for loop-based compiler optimization of the adjoint code and also incurs a **spatial overhead proportional to the loop iteration count**. This same fallback solution can be applied to all unstructured constructs; however, the now obvious recommendation is to **avoid unstructured control flow**.

Exception handling is more complicated than unstructured control flow because it can involve jumps between subroutines. Reversing such jumps by restoring a call stack is

[3]For simplicity imagine that all control paths enter a loop body through its enclosing "do" node and exit through its "done" node and all control paths enter a branch body through its enclosing "if" node and exit through its "fi" node.

[4]That is, the same reason that prevents the jump into the control flow structure in the first place also prevents this work around.

theoretically possible but practically very costly. Because of the implied performance degradation, numerical models rarely use exception for anything other than a graceful exit from error conditions. Hence, no practical efforts have been undertaken to implement reversal of exceptions, and **we recommend avoiding the use of exceptions for nonterminal error handling**. C++ exception handling cleanly unwinds the stack of the `try` block, and as part of good coding practice the logic in the corresponding `catch` block should explicitly revert all sideffects not local to the `try` block but incurred by it prior to the exit or the exception being (re)thrown. If that could be asserted by the user, the control flow portion of the trajectory restoration would need only to note an exception upon which the reverse sweep can simply skip both the `catch` and the `try` block.

3.2 *Temporaries for Arrays*

Reverse differentiation must split out arguments of procedure calls that are not simple variable references, but instead are complex expressions of other function calls. For instance `foo(bar(x));` must be changed to `t=bar(x); foo(t);`, thus introducing a temporary variable `t`. For scalar temporaries one simply has to determine the appropriate type, but for Fortran array variables one also has to determine the shape (i.e., the array dimensions). In Fortran a stack array variable has to have dimensions precomputed upon entry of the scope (subroutine, module), or else the array has to be allocated on the heap with the overhead this incurs. An example is given in Fig. 9 (top) where the shape changes (the size increases) in each loop iteration. Preallocating temporaries matching

```
1  do i=42,j,-1
2    a(i:)=sin(sqrt(a(i:)))
3  end do
```

```
1  do i=42,j,-1
2    allocate(t(size(a(i:))))
3    t=sqrt(a(i:))
4    a(i:)=sin(t)
5    deallocate(t)
6  end do
```

Figure 9. Code snippet for loop with implicit variable shape temporary (top) and rewritten with a heap variable according to the suggestions made (bottom)

the full size of the involved arrays may prove inefficient if the arrays are huge but the slices in the array operations small. Alternatively the AD transformation could turn the array operations into explicit loops on scalar entities as some compilers internally do. Consequently, **we recommend preferring explicit loops with scalar operations over array operations on variable shapes**.

Even when no array operations are involved (as in C or C++), temporaries may need to be created for array arguments to subroutines. Suppose a subroutine may be called with an active array argument and therefore its adjoint subroutine must pass derivative values. If in some call context, the actual array argument is passive, one has to create and pass a dummy array as a placeholder for the derivative values expected in the interface. If the original array argument is a pointer (in C/C++) or an assumed size array (in Fortran) (see also Sec. 2.1 and Sec. 4.1), the correct size of that placeholder is difficult to determine. Therefore **we recommend avoiding assumed-size arrays**.

3.3 *Dynamic Memory*

When the value of a reference expression must be stored in the trajectory and restored later and this expression uses pointers or array indices, these pointer values and indices must also be available when the expression is restored and, if overwritten, must become part of the trajectory restoration. A problem arises when the memory the pointer points to is deallocated, as shown in Fig. 10. Because the derivative computation proceeds

```
1  module m
2   real, allocatable, target :: x(:)
3   contains
4
5   subroutine bar(y)
6    real :: y(:)
7    real, pointer :: p
8    integer :: i=2
9    allocate(x(3))
10   x=y
11   p=>x(i+1) !compute addr.
12   p=p*2 !more uses etc.
13   y=x
14   deallocate(x)
15  end subroutine
16 end module
```

Figure 10. Local deallocation of dynamic memory.

backwards, it needs to restore the pointer value of `p`; but the array `x` was deallocated. Even though the adjoint code will reallocate `x`, it is unlikely to reside at the same spot in memory; therefore, pushing the pointer value of `p` when it goes out of scope is useless. To restore a semantically correct pointer value, one needs to recompute the address from a possibly changed base and increment by a stored offset. In general there is no guarantee that a simple syntactic expression for the changed base pointer is available at the location where the adjoint code needs to recompute the address. A pointer is considered merely assigned (i.e., not computed) only for the initial association of "unnamed" dynamic memory as, for instance, in an `allocate` call in Fortran or the assignment of the direct return value of `malloc/new` in C/C++. In our example in Fig. 10 the right-hand side of line 11 does provide a syntactic representation of the base pointer; but if instead of the assignment the statement in line 11 was a subroutine call setting the value of `p`, then no "simple" syntactic expression is available. One couldn't just move that hypothetical call out of proper reverse order to precede the adjoint of line 12 where the pointer value is required, because that call may have side effects or may be computationally expensive, e.g., when the pointer is assigned as a result of a search in data.

Instantiation of arrays from dynamic memory as discussed in Sec. 2.1 is benign as long as the address of the array or any of its elements is not input to another address computation. In that case, no dereference is ever done via an absolute pointer value that would have to be stored. It is thus sufficient to store the offsets (indices). The change in the base pointer (between the forward and the reverse sweep) remains inconsequential as long as the restrictions with respect to a checkpointing scheme, see Sec. 5 are obeyed.

Finally, absolute pointer values retain validity for trajectory restoration if base pointers do not change. The latter can be ensured if no dynamic memory is deallocated before the reverse sweep starts, which implicitly also prohibits reallocation to a new base address. Therefore, **if pointers are assigned computed addresses to dynamic memory**

then an adjoint transformation requires using dynamic memory management without deallocation or reallocation. Such a scheme can be beneficial for the original model already because it minimizes the number of costly operating system calls for memory allocation. Final deallocation preempting the operating system cleanup upon exiting the process (that is, after the reverse sweep has completed) is of course permitted.

3.4 *Message-passing communication*

Data-flow reversal of programs that use Message-passing parallelism appears straightforward on simple, textbook-like usage patterns. Send/receive pairs are simply exchanged. Collective communications are benign in general. More interestingly, a non-blocking pair isend/wait becomes a non-blocking irecv/wait, but as the execution order is reversed the irecv occupies the *adjoint* location of the wait, and the adjoint wait occupies the *adjoint* location of the isend. The same logic applies symmetrically to a non-blocking pair irecv/wait.

However, a program can obfuscate the link between a non-blocking call and its wait. The connection is done through the request identifier, which can be passed and copied in ways intractable by static analysis. The only general answer is a specific library that encapsulates the message-passing library, and that manages all message-passing decisions, *dynamically* during the forward and the reverse sweep. This involves sophisticated bookkeeping, putting an extra temporal overhead on the already costly message-passing calls.

Although unavoidable in general, the use of this bookkeeping library can be avoided for simple usage patterns, automatically applying transformations in terms of the original message-passing library instead. The key and missing prerequisite is of course automated detection of said simple patterns. In anticipation of AD tools detecting these favorable patterns in the future, **avoid if possible to move the communication request from one variable to another before it is used by its wait**. Also **avoid placing the non-blocking communication call and its wait in separate subroutines**.

4. Impact of Data Representation on Checkpointing

In Sec. 1 we explained the need for checkpointing. The code analysis provides the set of program variables that need to be stored and restored to enable recomputation from a given checkpoint. Exactly which values are associated with a program variable and how the code transformation orchestrates the storage are obvious only in the case of scalar variables of the built-in plain data types. As soon as arrays, structures, pointers, and references are considered for checkpointing, a set of problems has to be solved that is associated in other contexts with the concept of *serialization*. Practical experience shows that **the main efficiency penalty arises from overestimating the checkpoints leading to spatial overhead**, thereby reducing the number of checkpoints that can be stored, which then **may require additional forward computations as a temporal overhead** in the tradeoff. In the following we consider scenarios that affect adjoint computations most frequently.

4.1 *Arrays Represented by Pointers*

Built-in Fortran arrays with the exception of assumed-size arrays allow one to query shape information of arrays and array slices anywhere in the code; see also Sec. 2.1 and Sec. 3.2. A typical pattern in C and C++ is the passing of arrays via pointers; multi-dimensional arrays are represented by arrays of pointers. There, dimension information is not syntactically coupled to the array variables and cannot be queried. To generate code for writing checkpoints one must know whether a given pointer points to a single scalar value or else how many values are semantically associated with it. Consequently, no compile time algorithm currently achieves the goal for the general case.

The variable-length arrays provided in the C99 standard are unfortunately not supported in C++. Instead, C++ programmers often use container classes like `std::vector`. Alas, no standardized C++ solution exists for multidimensional arrays, and existing container classes incur a considerable performance penalty as long as the dereference syntax follows the idiom of pointer dereferencing as in `double **a; ... a[i][j]=1.0;`. The root cause is the successive application of `[]` operators in a chain requiring the creation of temporary objects. Other approaches that avoid the `[]` operator chain can achieve competitive performance, but without a standardized implementation the recognition and exploitation of specific containers in code analysis and source transformation tools require considerable implementation effort for each set of such containers.

Because in **C and C++ codes** pointers representing arrays are difficult to avoid, the most appropriate solution is to **hand-write checkpointing routines guided by side-effect analysis results**. While the problem does not exist for fixed and assumed-shape Fortran arrays, the notable exception is again the assumed-size array. Thus, for the ability to checkpoint we recommend strongly to **avoid Fortran assumed-size arrays**. As a fallback solution it is possible to implement **runtime bookkeeping** of dimension information associated with array pointers, but this **is computationally expensive to the point where it becomes prohibitive**.

4.2 *References versus Instances and Recursive Structures*

A problem well known from serialization is to decide whether a given value referenced, for instance, by some pointer has already been written to the checkpoint (think serialized) via some other pointer (or reference). For example, if one has to checkpoint data represented as a linked list, one must follow all the pointers to checkpoint all the data. However, if other data structures had pointers to the elements in that same list, one would prefer to not checkpoint these list elements again. A simple example illustrating the severity of the problem is a circular linked list. Neither Fortran nor C/C++ provide syntactic features that help decide the question. Future solutions to this issue may combine storage of individual values with direct mapping of some parts of the memory (e.g. `memcpy` in C). As mentioned earlier, this requires inserting runtime bookkeeping to keep track of relative virtual addresses. In the current state of the art AD tools will probably resort, as in Sec. 4.1, to asking the end-user to **hand-write checkpointing routines guided by side-effect analysis results**. The logic implemented for Python in `copy.deepcopy` and in serialization by `pickle` illustrates the concept and the performance penalty of automatic bookkeeping for recursive structures.

5. Syntactic Scopes, Checkpointing, and Resources

A piece of source code (e.g., a subroutine) is called *non-reentrant* when re-executing it produces a different behavior, because it affects some property of the machine state that cannot be stored and restored simply as a variable value. Consequently, *non-reentrant* source code cannot be checkpointed. In Sec. 1 we explained the repeated forward computation from checkpoints. In practice, recomputations of sections are delimited by syntactic scopes (loop bodies, subroutines). Allocation and release of program resources (dynamic memory, file handles, mutexes, MPI communicators, etc.) may span multiple syntactic scopes. Consequently, a recomputation scheme unaware of a resource lifespan may erroneously attempt multiple allocations or releases of the same resource (i.e., reenter non-reentrant code). For instance checkpointing a subroutine that allocates dynamic

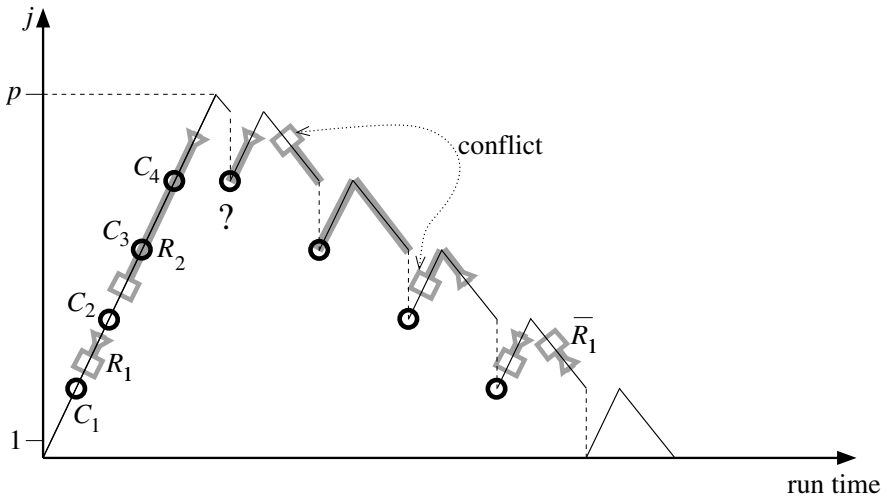

Figure 11. The scope for resource R_1 is benign while the scope for resource R_2 straddles checkpoints and causes problems.

memory to some variable but does not release it means on reentry just resetting the pointer value and leaking memory. This situation is illustrated in Fig. 11. The resource R_1 is acquired and released between checkpoints C_1 and C_2 and the same can be done for the recomputation started from C_1. For the corresponding reverse sweep the adjoint resource $\overline{R}_1$ is acquired where the forward sweep releases it and vice versa. Note that it depends on the kind of resource whether or not R is distinct from $\overline{R}$. For example, values being read from some data file would imply opening and closing the same data file in the forward and the reverse sweep while dynamic memory allocated for an array does not have to occupy the same heap location in the two sweeps. Conversely, the resource R_2 straddles checkpoints C_3 and C_4. For the forward sweep started from C_4 there is no logic that acquires R_2. One could argue that the acquisition could be made part of the logic that restores state from the checkpoint. Still, one would have to deal with the potentially conflicting acquisitions shown for the reverse sweep following the restoration from C_4 and the forward sweep following the restoration from C_2. A generic solution (not currently implemented in any AD tool) to this problem requires some bookkeeping logic that relates the resource scope to the checkpointing scheme to ensure proper action upon checkpoint restoration and proper action upon passing the acquisition and release points in the forward and reverse sweeps. Because dynamic memory is one of the resources to be handled, portions of this logic would resemble a garbage collection algorithm which

gives some indication of the complexity of its implementation.

Also in this category fall message passing communications in parallel programs where the resource is the communication channel and hence a checkpoint may not be taken while a message has been sent but not yet received. In short, in order to avoid the non-reentrant situation, any portion of the model source code to be reexecuted in a checkpointing scheme must **always contain none or both ends of any resource lifespan such as file open/close, memory allocate/deallocate, and message send/receive**.

For the last problem scenario, we consider checkpointing data accessible via variables that are not in the scope of the location where the checkpoint is taken. That applies to all global variables not visible at this point. The transformation algorithm has to inject the missing declarations and resolve potential name clashes. It also applies to remanent local variables given in the examples in Fig. 12 as `s`. The notable difference

```
1 subroutine foo(x)
2   real, save :: s=1.0
3   s=s*2; x=s-1
4 end subroutine
```

```
1 void foo(int& x) {
2   static int s=x;
3   s=s*2; x=s-1;
4 }
```

Figure 12. Local remanent variables in Fortran (top) and C++ (bottom).

between the Fortran and C++ variants is that in Fortran the initialization expression for s has to evaluate to a constant at compile time. This enables the source transformation tool to promote that variable to a remanent variable of global scope. In C++ this is not so simple because the one-time initialization value can be computed. Multiple call sites to `foo` in the source code imply that at compile time no call may provably be designated the *first* call. To emulate the local scope remanent variable semantic with a global scope remanent variable, explicit runtime bookkeeping needs to be generated to ensure the one-time initialization with the correct value. However, while current AD tool implementations can identify the variables that have to be moved, they do not yet have the capabilities to automatically perform the moves. Until they do, we recommend for C++ to manually **move remanent local declarations to a scope enclosing all checkpointing locations**.

6. Summary

In this paper we briefly describe a collection of commonly used programming features, idioms, and patterns that impact the efficiency of code generated by source transformation for the computation of adjoints. With the benefit of the reader by now having a better understanding of cause and effect, we will wrap our recommendations up as **things to try if your adjoint code is too slow or unwieldy**. Some of these are mentioned in manuals of the various AD tools. To our knowledge, however, this paper is the first reasonably comprehensive description of the relevant issues:

- Don't call the code to be differentiated in a context that introduces aliasing (Sec. 2.1)
- Avoid pointers whenever possible (Sec. 2.1)

- Reduce the number of (active) pointer values (including their aliases) being overwritten or going out of scope (Sec. 2.1)
- In Fortran, avoid the use of `pointer` variables where possible in favor of `allocatable` variables (Sec. 2.1)
- If pointers are assigned computed addresses to dynamic memory then try and move allocation, deallocation and reallocation outside the differentiated code portion (Sec. 3.3)
- Use interfaces and enhanced array syntax rather than parameter passing by storage association (Sec. 2.1)
- Avoid `union` and `equivalence` (Sec. 2.2)
- Avoid the "big work array" pattern (Sec. 2.3)
- Avoid unstructured interfaces, casting, and transferring intermediate data via files (Sec. 2.4)
- Expose as much as possible data structure in interfaces (Sec. 2.4)
- Avoid Fortran assumed-size arrays (Sec. 3.2, Sec. 4.1)
- Avoid unstructured control flow (Sec. 3.1)
- Avoid the use of exceptions for nonterminal error handling (Sec. 3.1)
- Prefer explicit loops with scalar operations over array operations on variable shapes (Sec. 3.2)
- Use message-passing primitives so as to maximize static separation of groups of sending and receiving endpoints (Sec. 2.4)
- Avoid moving the communication request from one variable to another before it is used by its wait (Sec. 3.4)
- Avoid placing the non-blocking communication call and its wait in separate subroutines (Sec. 3.4)
- Design your code so that hand-writing checkpointing routines is easy, guided by side-effect analysis results (Sec. 4.1)
- Design your code such that checkpointed portions always contain none or both ends of any resource lifespan such as file open/close, memory allocate/deallocate, and message send/receive (Sec. 5)
- Move remanent local declarations to a scope enclosing all checkpointing locations (Sec. 5)

We aim less at presenting formally exact descriptions and more at being easily understood by AD tool users who are not experts in compiler technology or source code analysis. Undesirable effects can arise from many areas of a program design such as interfaces, data structures, and control flow. We present the reasons for the effects (overhead in the generated adjoint code) with respect to the causes (use of certain patterns, language features etc.). Where possible we also quantify (with a crude guess) the overhead in relation to certain underlying analysis overestimates. Our objective is to highlight the principal issues and give *plausible* explanations for recommendations designed to help users of AD tools improve the performance of the transformed source code. The scope of this paper does not include an *exact* characterization of the conditions when certain programming features and idioms have undesirable effects. The conditions depend not only on the circumstances in the source code but also on choices in the analyses implementations, which are hard to formalize and even harder to verify for the user of AD tools. We have, however, explained how analyses overestimates and efficiency penalties grow monotonely. Any overestimate reduction, even in the worst case, cannot be detrimental to the generated code. With a few noted exceptions all the changes that benefit the efficiency of the adjoint code can also be expected to aid the compiler optimization of the original source code.

Acknowledgement(s)

This work was supported by the U.S. Department of Energy, under contract DE-AC02-06CH11357.

References

[1] AD community website, `http://www.autodiff.org`.

[2] C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, and J. Utke (eds.), *Advances in Automatic Differentiation*, Lecture Notes in Computational Science and Engineering, Vol. 64, Springer, Berlin, 2008.

[3] B. Creusillet and F. Irigoin, *Interprocedural array region analyses*, International Journal of Parallel Programming 24 (1996), pp. 513–546.

[4] B. Dauvergne and L. Hascoët, *The Data-Flow Equations of Checkpointing in Reverse Automatic Differentiation*, in *Computational Science – ICCS 2006*, Lecture Notes in Computer Science, Vol. 3994, Springer, Heidelberg, 2006, pp. 566–573.

[5] S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther (eds.), *Recent Advances in Algorithmic Differentiation*, Lecture Notes in Computational Science and Engineering, Vol. 87, Springer, Berlin, 2012.

[6] A. Griewank and A. Walther, *Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation*, ACM Transactions on Mathematical Software 26 (2000), pp. 19–45, Available at `http://doi.acm.org/10.1145/347837.347846`, also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[7] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., no. 105 in Other Titles in Applied Mathematics, SIAM, Philadelphia, PA, 2008, Available at `http://www.ec-securehost.com/SIAM/OT105.html`.

[8] MPI, `http://www.mcs.anl.gov/mpi`, the Message Passing Interface Standard.

[9] M. Strout, B. Kreaseck, and P. Hovland, *Data-Flow Analysis for MPI Programs*, in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2006, pp. 175 –184.

[10] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann, *Toward Adjoinable MPI*, in *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering, PDSEC-09*, Rome, Italy, 2009.