The Data-Flow Equations of Checkpointing in reverse Automatic Differentiation

Benjamin Dauvergne[1] and Laurent Hascoët[1]

INRIA Sophia-Antipolis, TROPICS team,
2004 Route des lucioles, BP 93, 06902 Sophia-Antipolis, France

Abstract. Checkpointing is a technique to reduce the memory consumption of adjoint programs produced by reverse Automatic Differentiation. However, checkpointing also uses a non-negligible memory space for the so-called "snapshots". We analyze the data-flow of checkpointing, yielding a precise characterization of all possible memory-optimal options for snapshots. This characterization is formally derived from the structure of checkpoints and from classical data-flow equations. In particular, we select two very different options and study their behavior on a number of real codes. Although no option is uniformly better, the so-called "lazy-snapshot" option appears preferable in general.

1 Introduction

Mathematical derivatives are a key ingredient in Scientific Computation. In particular, gradients are essential in optimization and inverse problems. The reverse mode of Automatic Differentiation (AD) is probably one of the most convenient and efficient ways to obtain gradients. However, it still offers room for improvement regarding efficiency, and more importantly it suffers from a high memory consumption.

This memory consumption is inherent to the nature of gradient computation. For instance the other highly efficient way to obtain gradients is to solve the "adjoint equations", which is also known to consume memory space. However, solving the adjoint equations usually implies a great deal of hand-coding, which makes this approach less convenient.

Being a software transformation technique, reverse AD can and must take advantage from software analysis and compiler technology [1] to minimize these efficiency problems. In this paper, we will analyze "checkpointing", an AD technique to trade repeated computation for memory consumption, with the tools of compiler data-flow analysis.

Checkpointing offers a range of fine-grain options that affect the efficiency of the resulting differentiated code. Our goal is to formalize these options and to find which ones are optimal. This study is part of a general effort to formalize all the compiler techniques useful to reverse AD, so that AD tools can make the right choices using a firmly established basis.

2 Reverse Automatic Differentiation

Automatic Differentiation is a program transformation technique. Given a program P that computes a differentiable function F, an AD tool creates a new program that computes some derivatives of F. Based on the chain rule of calculus, AD introduces into P new "derivative" statements, each one corresponding to one original statement of P. In particular, reverse AD creates a program $\overline{P}$ that computes gradients. In $\overline{P}$, the derivative statements corresponding to the original statements are executed in *reverse order* compared to P. The derivative statements use some of the values used by their original statement, and therefore the original statements must be executed in a preliminary "forward sweep" $\overrightarrow{P}$, which produces the original values that are used by the derivative statements forming the "backward sweep" $\overleftarrow{P}$. This is illustrated by fig. 1, in which we have readily split P in three successive parts, U upstream, C in the center, and D downstream. In our context, original values are made available to the backward sweep through `PUSH` and `POP` routines, using a stack that we call the "tape". Not

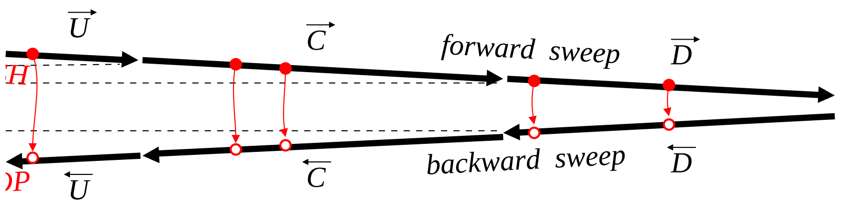

Fig. 1. Basic structure of reverse differentiated programs

all original values are required in the backward sweep. Because of the nature of differentiation, values that are used only "linearly" are not required. The "To Be Restored" (TBR) analysis finds the *Req* set of these required values. This *Req* set evolves as the forward sweep advances. For example on fig. 1, TBR analysis of U finds the variable values required by $\overleftarrow{U}$ (i.e. actually $\mathbf{use}(\overleftarrow{U})$), which must be preserved between the end of $\overrightarrow{U}$ and the beginning of $\overleftarrow{U}$. In other words, $\overline{C;D} \doteqdot \overrightarrow{C};\overrightarrow{D};\overleftarrow{D};\overleftarrow{C}$ must be built in such a way that:

$$\mathbf{out}(\overline{C;D}) \cap \mathbf{use}(\overleftarrow{U}) = \emptyset \ .$$

To this end, each time a required value is going to be overwritten by a statement, it is `PUSH`'ed beforehand, and it is `POP`'ped just before the derivative of this statement.

Although somewhat complex, reverse AD can be easily applied by an automatic tool, and has enormous advantages regarding the number of computation steps needed to obtain the gradient. See [4, chapter 3] for a thorough discussion on its merits, and [2, 6] for a complete description of the TBR mechanism.

In [5], we studied the data-flow properties of reverse differentiated programs, in the basic case of fig. 1, i.e. with no checkpointing. We formalized the structure of these programs and derived specialized data-flow equations for the “adjoint liveness” analysis, which finds original statements that are useless in the differentiated program, and for the TBR analysis. In this paper, we will focus on the consequences of introducing checkpointing. In this respect this paper, although stand-alone, is a continuation of [5].

3 The equations of Checkpointing Snapshots

Checkpointing reduces the peak memory consumption by duplicating a “checkpointed” code fragment, e.g. C on fig. 2. This allows the tape consumed by $\overline{D} \doteqdot \overrightarrow{D}; \overleftarrow{D}$ to be freed before execution of $\overline{C} \doteqdot \overrightarrow{C}; \overleftarrow{C}$. The peak memory consumption for $\overline{C;D}$ is thus reduced to the maximum of the peak for $\overline{C}$ and the peak for $\overline{D}$.

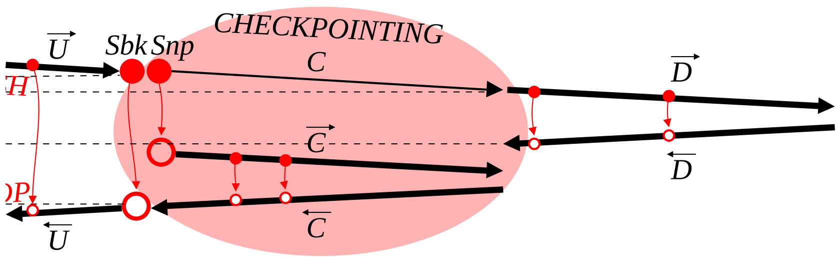

Fig. 2. Checkpointing in reverse AD

However, duplicate execution of C requires that “enough” variables (the “snapshot” *Snp*) are preserved to restore the context of execution. This also uses memory space, although less than the tape for $\overrightarrow{C}$. To not lose the benefit of checkpointing, it is therefore essential we keep the snapshot small.

The coupling between $\overline{D}$ and $\overline{C}$ makes this trade-off tricky. A larger snapshot can mean smaller tapes, and conversely; intuitively, if the snapshot saves extra variables, then these variables vanish from the *Req* set before $\overline{D}$, and thus the tape used by $\overline{D}$ may turn out smaller. Therefore, unlike what happens for the simpler case with no checkpoints, there is no unique best choice for these sets. There are several “optimal” choices, among which none is better nor worse than the others. Our goal is to establish the constraints that define and link the “snapshot” and “tape” sets, yielding necessary set equations that will characterize all the optimal choices. The final decision depends on each particular case. For our AD tool TAPENADE, we settled on a trade-off (*cf* sec. 4) that our benchmarks indicated as a mean best choice.

Four unknown sets of variables: Let's write the differentiated program depicted in fig. 2 as:

$$\begin{aligned} \mathit{Req} \vdash \overline{C;D} = {} & \texttt{PUSH}(\mathit{Sbk}); \\ & \texttt{PUSH}(\mathit{Snp}); \\ & C; \\ & \mathit{Req}_D \vdash \overline{D}; \\ & \texttt{POP}(\mathit{Snp}); \\ & \mathit{Req}_C \vdash \overline{C}; \\ & \texttt{POP}(\mathit{Sbk}); \end{aligned}$$

where *Req* is the incoming "required set" of variables that must be preserved across execution of $\mathit{Req} \vdash \overline{C;D}$, because they are required by the derivative instructions $\overleftarrow{U}$ of the upstream part U. In other words *Req* is a requirement imposed on $\overline{C;D}$ by the upstream context U. On the other hand, Req_D and Req_C are the sets of variables that $\overline{C}$ and $\overline{D}$ will be required to preserve, respectively. We have several choices at hand for Req_D and Req_C, depending on *Req* and on our choice for the snapshot. Thus Req_D and Req_C are our unknowns, to be determined together with the snapshot.

Let's now examine the snapshot itself. Due to the stack structure, there are two places where variables may be popped from the stack and restored: before the duplicated run $\overline{C}$, and after it i.e. before running $\overleftarrow{U}$. Therefore we introduce two snapshot sets *Snp* and *Sbk*.

- Variables in *Snp* are restored just before running $\overline{C}$, thus ensuring that their value is the same for both executions of C. *Snp* corresponds to the usual definition of the snapshot.
- Variables in *Sbk* "*backward snapshot*" are restored just before running $\overleftarrow{U}$. This ensures that, whatever happens to these variables during $\overline{C;D}$, their value is preserved when going back into $\overleftarrow{U}$. We view *Sbk* as an hybrid of snapshot and tape.

In total, we have four "unknown" sets to choose: Req_D, Req_C, *Sbk* and *Snp*. Those sets must respect constraints parameterized upon *Req*, Req_D, Req_C, *Sbk*, *Snp*, and upon the fixed **use** and **out** data-flow sets for the code fragments C, D, $\overline{C}$, and $\overline{D}$. Essentially, these constraints will guarantee that checkpointing actually preserves the computed derivatives.

Two necessary and sufficient conditions: Fig. 1 shows the differentiated program in the reference case with no checkpointing. This reference program is assumed correct. All we need to guarantee is that the result of the differentiated program, i.e. the derivatives, remain the same when checkpointing is done. This can be easily formulated in terms of data-flow sets. We observe that the order of the backward sweeps is not modified by checkpointing. Therefore the derivatives are preserved if and only if the original, non-differentiated variables that are used during the backward sweeps hold the same values. In other words, the snapshot and the tape must preserve the **use** set of $\overline{C}$ between time t_1 and t_3 i.e.

$$\mathbf{out}\begin{pmatrix}\texttt{PUSH}(\mathit{Sbk});\\ \texttt{PUSH}(\mathit{Snp});\\ C;\\ \mathit{Req}_D \vdash \overline{D};\\ \texttt{POP}(\mathit{Snp});\end{pmatrix} \bigcap \mathbf{use}(\overline{C}) = \emptyset \tag{1}$$

and the **use** set of $\overleftarrow{U}$, which is *Req* by definition, between time t_1 and t_4 i.e.

$$\mathbf{out}\begin{pmatrix}\texttt{PUSH}(\mathit{Sbk});\\ \texttt{PUSH}(\mathit{Snp});\\ C;\\ \mathit{Req}_D \vdash \overline{D};\\ \texttt{POP}(\mathit{Snp});\\ \mathit{Req}_C \vdash \overline{C};\\ \texttt{POP}(\mathit{Sbk});\end{pmatrix} \bigcap \mathit{Req} = \emptyset \tag{2}$$

The rest is purely mechanical. Classically, the **out** set of a sequence of code fragments is the union of the **out** sets of each fragment, i.e.

$$\mathbf{out}(A;B) = \mathbf{out}(A) \cup \mathbf{out}(B) \ \ ,$$

except in the very special case of the PUSH/POP pair, which remove their argument from the stack, i.e.

$$\mathbf{out}(\texttt{PUSH}(v);A;\texttt{POP}(v)) = \mathbf{out}(A) \setminus \{v\} \ \ .$$

Also, the mechanism of reverse AD ensures that the variables in the required context are actually preserved, so that:

$$\mathbf{out}(\mathit{Req} \vdash \overline{A}) = \mathbf{out}(\overline{A}) \setminus \mathit{Req} \ \ .$$

Therefore, equation (1) becomes:

$$\big(\mathbf{out}(C) \cup (\mathbf{out}(\overline{D}) \setminus \mathit{Req}_D)\big) \setminus \mathit{Snp} \bigcap \mathbf{use}(\overline{C}) = \emptyset \tag{3}$$

and equation (2) becomes:

$$\Big(\big(\mathbf{out}(C) \cup (\mathbf{out}(\overline{D}) \setminus \mathit{Req}_D)\big) \setminus \mathit{Snp} \cup \big(\mathbf{out}(\overline{C}) \setminus \mathit{Req}_C\big)\Big) \setminus \mathit{Sbk} \bigcap \mathit{Req} = \emptyset \ \ . \tag{4}$$

From (3) and (4), we obtain necessary and sufficient conditions on *Sbk*, *Snp*, Req_D and Req_C. These conditions ensure that the differentiated programs with and without checkpointing return the same derivatives:

$$\begin{aligned}
\mathit{Sbk} &\supseteq \big((\mathbf{out}(C) \cup (\mathbf{out}(\overline{D}) \setminus \mathit{Req}_D)) \setminus \mathit{Snp} \\
&\qquad\qquad\qquad \cup\ (\mathbf{out}(\overline{C}) \setminus \mathit{Req}_C)\big) \cap \mathit{Req} \\
\mathit{Snp} &\supseteq \big(\mathbf{out}(C) \cup (\mathbf{out}(\overline{D}) \setminus \mathit{Req}_D)\big) \cap \big(\mathbf{use}(\overline{C}) \cup (\mathit{Req} \setminus \mathit{Sbk})\big) \\
\mathit{Req}_D &\supseteq (\mathbf{out}(\overline{D}) \setminus \mathit{Snp}) \cap \big(\mathbf{use}(\overline{C}) \cup (\mathit{Req} \setminus \mathit{Sbk})\big) \\
\mathit{Req}_C &\supseteq (\mathbf{out}(\overline{C}) \setminus \mathit{Sbk}) \cap \mathit{Req}
\end{aligned}$$

Notice the cycles in these inequations. If we add a variable into *Snp*, we may be allowed to remove it from Req_D, and vice versa: as we said, there is no unique best solution. Let's look for the minimal solutions, i.e. the solutions to the equations we obtain by replacing the "$\supseteq$" sign by a simple "=".

Solving for the unknown sets: Manipulation of these equations is tedious and error-prone. Therefore, we have been using a symbolic computation system (e.g. Maple [8]). Basically, we have inlined the equation of, say, *Snp* into the other equations, and so on until we obtained fixed point equations with a single unknown X of the form

$$X = A \cup (X \cap B) \quad ,$$

whose solutions are of the form "A plus some subset of B". The solutions are expressed in terms of the following sets:

$$\begin{aligned}
Snp_0 &= \mathbf{out}(C) \cap (\mathbf{use}(\overline{C}) \cup (Req \setminus \mathbf{out}(\overline{C}))) \\
Opt_1 &= Req \cap \mathbf{out}(\overline{C}) \cap \mathbf{use}(\overline{C}) \\
Opt_2 &= Req \cap \mathbf{out}(\overline{C}) \setminus \mathbf{use}(\overline{C}) \\
Opt_3 &= \mathbf{out}(\overline{D}) \cap (\mathbf{use}(\overline{C}) \cup Req) \setminus \mathbf{out}(C)
\end{aligned} \tag{5}$$

For each partition of Opt_1 in two sets Opt_1^+ and Opt_1^-, and similarly for Opt_2 and Opt_3, the following is a minimal solution of our problem:

$$\begin{aligned}
Sbk &= Opt_1^+ \cup Opt_2^+ \\
Snp &= Snp_0 \cup Opt_2^- \cup Opt_3^+ \\
Req_D &= \phantom{Snp_0 \cup Opt_2^- \cup{}} Opt_3^- \\
Req_C &= Opt_1^- \cup Opt_2^-
\end{aligned} \tag{6}$$

and there are no other optimal solutions. Furthermore, $Opt_1 \subseteq Snp_0$, and the sets Snp_0, Opt_2, and Opt_3 are disjoint.

4 Discussion and Experimental Results

The final decision for sets *Sbk*, *Snp*, Req_D, and Req_C depends on each particular context. No strategy is systematically best. We looked at two options.

We examined first the option that was implemented until recently in our AD tool TAPENADE [7]. We call it "*eager snapshots*". This option stores enough variables in the snapshots to reduce the sets Req_D and Req_C as much as possible, therefore reducing the number of subsequent PUSH/POP in $\overline{D}$ and $\overline{C}$. Equations (6) show that we can even make these sets empty, but experiments showed that making Req_D empty can cost too much memory space in some cases.

As always, the problem behind this is undecidability of array indexing: since we can't always tell whether two array indexes designate the same element or not, the "eager snapshot" strategy may end up storing an entire array whereas only one array element was actually concerned.

Therefore "eager snapshot" chooses Opt_1^- and Opt_2^- empty but

$$\begin{aligned} Opt_3^+ &= \mathbf{out}(\overline{D}) \cap (\mathbf{use}(\overline{C}) \setminus Req) \setminus \mathbf{out}(C) \\ Opt_3^- &= \mathbf{out}(\overline{D}) \cap Req \setminus \mathbf{out}(C) \end{aligned}$$

which gives:

$$\begin{aligned} Sbk &= Req \cap \mathbf{out}(\overline{C}) \\ Snp &= (\mathbf{out}(C) \cap (\mathbf{use}(\overline{C}) \cup Req \setminus \mathbf{out}(\overline{C}))) \cup \\ &\quad\ (\mathbf{out}(\overline{D}) \cap \mathbf{use}(\overline{C}) \setminus Req \setminus \mathbf{out}(C)) \\ Req_D &= \mathbf{out}(\overline{D}) \cap Req \setminus \mathbf{out}(C) \\ Req_C &= \emptyset \end{aligned} \tag{7}$$

Notice that intersection between *Sbk* and *Snp* is nonempty, and requires a special stack mechanism to avoid duplicate storage space.

We examined another option that is to keep the snapshot as small as possible, therefore leaving most of the storage work to the TBR mechanism inside $\overline{D}$ and $\overline{C}$. We call it "*lazy snapshots*", and it is now the default strategy in TAPENADE. Underlying is the idea that the TBR mechanism is efficient on arrays because when an array element is overwritten by a statement, only this element is saved.

Therefore, "lazy snapshot" chooses all Opt_1^+, Opt_2^+, and Opt_3^+ empty, yielding:

$$\begin{aligned} Sbk &= \emptyset \\ Snp &= \mathbf{out}(C) \cap (Req \cup \mathbf{use}(\overline{C})) \\ Req_D &= \mathbf{out}(\overline{D}) \cap (Req \cup \mathbf{use}(\overline{C})) \setminus \mathbf{out}(C) \\ Req_C &= \mathbf{out}(\overline{C}) \cap Req \end{aligned} \tag{8}$$

We ran TAPENADE on our validation application suite, for each of the two options. The results are shown on table 1. We observe that lazy snapshots perform better in general. Actually, we could show the potential advantage of eager snapshots only on a hand-written example, where the checkpointed part C repeatedly overwrites elements of an array in *Req*, making TBR mechanism more expensive than a global snapshot of the array. On real applications, however, this case is rare and lazy snapshots work better.

Whatever the option chosen, equations (6) naturally capture all interactions between successive snapshots. For example, if several successive snapshots all use an array `A`, and only the last snapshot overwrites `A`, it is well known that `A` must be saved only in the last snapshot. However, when an AD tool does not rely on a formalization of checkpointing such as the one we introduce here, it may very well happen that `A` is stored by all the snapshots.

5 Conclusion

We have formalized the checkpointing technique in the context of reverse AD by program transformation. Checkpointing relies on saving a number of variables and several options are available regarding which variables are saved and

Code	*Domain*	*Run-time*	*Eager (7)*	*Lazy (8)*
OPA	oceanography	780 s	480 Mb	479 Mb
STICS	agronomy	35 s	229 Mb	229 Mb
UNS2D	CFD	23 s	248 Mb	185 Mb
SAIL	agronomy	17 s	1.6 Mb	1.5 Mb
THYC	thermodynamics	12 s	33.7 Mb	18.3 Mb
LIDAR	optics	10 s	14.6 Mb	14.6 Mb
CURVE	shape optim	2.7 s	1.44 Mb	0.59 Mb
SONIC	CFD	0.2 s	3.55 Mb	2.02 Mb
Contrived example		0.1 s	8.20 Mb	11.72 Mb

Table 1. Comparison of the eager and lazy snapshot approaches on a number of small to large applications

when. Regarding memory consumption, no option is strictly better than all others: instead, there are a number of optimal options which can turn out to be the best on some source program configuration. Specializing standard data-flow equations for the particular structure of checkpoints, we obtain a precise description of all these possible optimal options. We thus have a tool for checking that a given definition of the sets involved in checkpointing is either optimal or is still missing some potential improvement. This gives us safer and more reliable implementation in AD tools.

Finding this precise description involves tedious manipulation of set equations. Therefore we used the help of a symbolic computation system for the "mechanical" part of this work.

We selected two possible optimal options and implemented them in the AD tool TAPENADE. Experience shows that the option called "lazy snapshots" performs better on most cases.

However, we believe that for reverse AD of a given application code, the option chosen need not be identical for all checkpoints. This formal description of all the possible options allows us to look for the best option for each individual checkpoint, based on static properties at this particular code location. In this regard, we used symbolic computation again and came up with a very pleasant property: for a given checkpoint, whatever the optimal option chosen for the snapshot, the **out** set of this piece of code turns out to be always the same:

$$\mathbf{out}(\overline{C;D}) = \big(\mathbf{out}(\overline{C}) \cup ((\mathbf{out}(\overline{D}) \cup \mathbf{out}(C)) \setminus \mathbf{use}(\overline{C}))\big) \setminus \mathit{Req}\ ,$$

and this **out** set is what is used in the optimal choices of the other checkpoints around. Therefore the choice of the optimal option is local to each checkpoint.

One of the current big challenges of reverse AD is to find the best possible architecture of nested checkpoints for any given program. Such an optimal architecture has been found for programs with a simple enough iterative structure [3]. For arbitrary programs, the question becomes so complex that heuristics will be needed. In any case the precise description given here for the memory cost of each checkpoint is essential for the optimal checkpointing problem.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. C. Faure and U. Naumann. Minimizing the tape size. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 34, pages 293–298. Springer, New York, NY, 2001.
3. Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.
4. Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.
5. L. Hascoët and M. Araya-Polo. The adjoint data-flow analyses: Formalization, properties, and applications. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.
6. L. Hascoët, U. Naumann, and V. Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2004.
7. L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical report 0300, INRIA, 2004. http://www.inria.fr/rrrt/rt-0300.html.
8. Darren Redfern. *The Maple handbook, Maple V, release 4*. Springer, 1996.