Adjoint Data-Flow analyses applied to checkpointing

Benjamin Dauvergne

TROPICS Project, INRIA Sophia-Antipolis

Why checkpoints?

- Instead of recording the tape of the executions, you want to reexecute some part of your code.
- To do this you need to restore the variables used by this part to the value they carried at the time of the first execution.
- $\mathbf{Used}$ here means read before written, it's a classical data flow definition, like $\mathbf{Def}$

How can we do it?

- By hand :
  we know the code, we know that there is something called the $state$ and this state is read and written at each iteration. We just save it on the stack.
- Automatically: when you write a source to source AD tool you don't know what the input code is doing, so you need data flow analysis to find out those used variables and if you really need to save them.

Problem: We don't have so good data flow informations especially concerning arrays.

What should we save?

Data flow notation from a previous paper of L. Hascoet and M. Araya.

$$P := [I_1, ..., I_n]$$

$$\overline{P} := \emptyset \vdash \overline{P}$$

$$\mathbf{TBR} \vdash \overline{I;D} \quad = \left| \begin{array}{l} \mathbf{PUSH}(\mathbf{Def}(I) \cap (\mathbf{TBR} \cup \mathbf{Use}(I'))) \\ I \\ (\mathbf{TBR} \cup \mathbf{Use}(I')) \setminus \mathbf{Def}(I) \vdash \overline{D} \\ \mathbf{POP}(\mathbf{Def}(I) \cap (\mathbf{TBR} \cup \mathbf{Use}(I'))) \\ I' \end{array} \right.$$

- P is the program, I_i are its instructions, I' is the adjoint code associated with a simple intruction and $\mathbf{TBR} \vdash \overline{P}$ is the generated adjoint for a program P given a contextual save set $\mathbf{TBR}$.
- No dead code removal like in this previous paper because this is not the point.

What should we save?

Data flow notation from a previous paper of L. Hascoet and M. Araya.

$$P := [I_1, ..., I_n]$$

$$\overline{P} := \emptyset \vdash \overline{P}$$

$$\mathbf{TBR} \vdash \overline{I;D} \quad = \quad \left| \begin{array}{l} \mathbf{PUSH}(\mathbf{Def}(I) \cap (\mathbf{TBR} \cup \mathbf{Use}(I'))) \\ I \\ (\mathbf{TBR} \cup \mathbf{Use}(I')) \setminus \mathbf{Def}(I) \vdash \overline{D} \\ \mathbf{POP}(\mathbf{Def}(I) \cap (\mathbf{TBR} \cup \mathbf{Use}(I'))) \\ I' \end{array} \right.$$

- This defi nition is recursive on the rest of the instruction list of P, and could be applied without much modifi cation to checkpoints.
- Checkpoints are juste big instructions

The checkpointing case

My goal is to extend this scheme to checkpoints.

Take some big part C and checkpoint it:
I' now becomes $\mathbf{TBR} \vdash \overline{C}$.

$$\mathbf{TBR} \vdash \overline{C;D} \quad = \left|\begin{array}{l} \mathbf{PUSH}\left(\mathbf{Def}(C) \cap (\mathbf{TBR} \cup \mathbf{Use}\left(\overline{C}\right))\right) \\ C \\ (\mathbf{TBR} \cup \mathbf{Use}\left(\overline{C}\right)) \backslash \mathbf{Def}(C) \vdash \overline{D} \\ \mathbf{POP}\left(\mathbf{Def}(C) \cap (\mathbf{TBR} \cup \mathbf{Use}\left(\overline{C}\right))\right) \\ \mathbf{TBR} \vdash \overline{C} \end{array}\right.$$

The checkpointing case

My goal is to extend this scheme to checkpoints.

Take some big part C and checkpoint it:
I' now becomes $\mathbf{TBR} \vdash \overline{C}$.

$$\mathbf{TBR} \vdash \overline{C;D} \quad = \quad \left| \begin{array}{l} \mathbf{PUSH}\left(\mathbf{Def}(C) \cap (\mathbf{TBR} \cup \mathbf{Use}\left(\overline{C}\right))\right) \\ C \\ (\mathbf{TBR} \cup \mathbf{Use}\left(\overline{C}\right)) \setminus \mathbf{Def}(C) \vdash \overline{D} \\ \mathbf{POP}\left(\mathbf{Def}(C) \cap (\mathbf{TBR} \cup \mathbf{Use}\left(\overline{C}\right))\right) \\ \mathbf{TBR} \vdash \overline{C} \end{array} \right.$$

We'll call this scheme «store later» because it postpones the save of $\mathbf{Use}\left(\overline{C}\right)$ in the downstream.

The checkpointing case 2

But you could also do this

$$\mathbf{TBR} \vdash \overline{I;D} \quad = \left| \begin{array}{l} \mathbf{PUSH}\,(\mathbf{Def}(C) \cap \mathbf{TBR})) \\ \mathbf{PUSH}\,\left(\mathbf{Def}(C) \cap \mathbf{Use}\,(\emptyset \vdash \overline{C}))\right) \\ C \\ (\mathbf{TBR} \cup \mathbf{Use}\,(\overline{C}))\backslash \mathbf{Def}(C) \vdash \overline{D} \\ \mathbf{POP}\,\left(\mathbf{Def}(C) \cap \mathbf{Use}\,(\emptyset \vdash \overline{C}))\right) \\ \emptyset \vdash \overline{C} \\ \mathbf{POP}\,(\mathbf{Def}(C) \cap \mathbf{TBR})) \end{array} \right.$$

Current tapenade checkpointing scheme

Here is another scheme: «store early».
We store all variables in $\mathbf{Use}\left(\overline{C}\right)$ that may be killed later.

$$\mathbf{TBR} \vdash \overline{I;D} \quad = \left| \begin{array}{l} \mathbf{PUSH}\left(\mathbf{Def}(C) \cap \mathbf{TBR}\right)^{set1} \\ \mathbf{PUSH}\left(\mathbf{Def}(C;D) \cap \mathbf{Use}\left(\overline{C}\right)\right)^{set2} \\ C \\ \left(\mathbf{TBR} \cup \mathbf{Use}\left(\overline{C}\right)\right) \setminus \left(set1 \cup set2\right) \vdash \overline{D} \\ \mathbf{POP}\left(set2\right) \\ \emptyset \vdash \overline{C} \\ \mathbf{POP}\left(set1\right) \end{array} \right.$$

A code where store early is bad

$$\text{Loop}\;\left|\;\begin{array}{l} proc_1(\text{Use } state, \text{Def } A) \\ proc_2(\text{Use } state, \text{Def } B) \\ proc_3(\text{Use } state, \text{Def } C) \\ proc_4(\text{Use } ABC, \text{Def } state) \end{array}\right.$$

Let's try to differentiate it checkpointing each proc.

A code where store early is bad

```
Loop | $\mathbf{PUSH}(state)$
     | $proc_1(\text{Use } state, \text{Def } A)$
     | $\mathbf{PUSH}(state)$
     | $proc_2(\text{Use } state, \text{Def } B)$
     | $\mathbf{PUSH}(state)$
     | $proc_3(\text{Use } state, \text{Def } C)$
     | $\mathbf{PUSH}(A,B,C)$
     | $proc_4(\text{Use } ABC, \text{Def } state)$
```

It’s not really good, each time we save *state*, we save the same values. Let’s try to use the «store later» checkpointing scheme.

A code where store early is bad

Loop

$$\begin{array}{|l}
\mathbf{PUSH}(A) \\
function_1(\mathbf{Use}() = state, \mathbf{Def}() = A) \\
\mathbf{PUSH}(B) \\
function_2(\mathbf{Use}() = state, \mathbf{Def}() = B) \\
\mathbf{PUSH}(C) \\
function_3(\mathbf{Use}() = state, \mathbf{Def}() = C) \\
\mathbf{PUSH}(state) \\
function_4(\mathbf{Use}() = A, B, C, \mathbf{Def}() = state = state)
\end{array}$$

Yeah, it’s better.

A code where store later is bad

Look at this code:

$$Loop\left|\begin{array}{l} function_1(use = \text{array}A) \\ do\,i = 1, n A(i) = 0.0 \text{\# Re-init } A \\ do\,k = 1, m A(indir(k)) = A(indir(k)) + \ldots \text{\# Gather loop} \end{array}\right.$$

Suppose $m >> n$ if we apply a «store later» scheme to A we'll store too much cells of A.

$$Loop\left|\begin{array}{l} function_1(use = \text{array}A) \\ do\,i = 1, n\,\mathbf{PUSH}(A(i))A(i) = 0.0 \text{\# Re-init } A \\ do\,k = 1, m\,\mathbf{PUSH}(A(indir(k)))A(indir(k)) = A(indir(k)) + \ldots \text{\# Gather loop} \end{array}\right.$$

If instead we used «store early»:

$$Loop\left|\begin{array}{l} \mathbf{PUSH}(A) \\ function_1(use = \text{array}A) \\ do\,i = 1, n A(i) = 0.0 \text{\# Re-init } A \\ do\,k = 1, m A(indir(k)) = A(indir(k)) + \ldots \text{\# Gather loop} \end{array}\right.$$

So there is a choice to make...

- We need rules or heuristics for choosing one of the scheme for each variable at each checkpoint site (in Tapenade they are call site).
- The «store later» seems the more natural scheme.
- The «store early» scheme should be used if we can infer an array is going to be completely written once or more.

Numerical results

On one of our test code using the current Tapenade scheme i was getting those resources utilisation:

```
VALIDATION TESTS FOR selmin-uns2d:
Time of  original  function:    2.269999962300062
Time of tangent AD function:    7.000000000000000
Time of reverse AD function:    25.48999786376953
Max Stack size: 15876 blocks of 16384 bytes
```

with a always «store later» scheme, i got those results:

```
VALIDATION TESTS FOR selmin-uns2d:
Time of  original  function:    2.289999943226576
Time of tangent AD function:    7.090000152587891
Time of reverse AD function:    22.73000049591064
Max Stack size: 11815 blocks of 16384 bytes
```

It's a 26% gain in terms of memory and a 11% gain on cpu, without even knowing the code.

Conclusion

- We still need better data flow information on our codes.
- We must use this information to choose the good structure for our adjoint codes, like where to put checkpoints and how to do them.

Other kind of work i'm working on..

- Reversing the data fbw instructions:
  - Easy if the new value is a stricly affi ne function of the old value:

Forward	Reverse
`a[i] += b*c+d...`	`a[i] -= b*c+d...`

  - Harder:

for doing something like this you

Forward	Reverse

```
do i = 1,m
  v1 = face(1)(i)
  v2 = face(1)(i)
  v3 = face(1)(i)
  x(1) = vertex(1)(v1)
  y(1) = vertex(2)(v1)
  z(1) = vertex(3)(v1)
  |...|
  z(3) = vertex(3)(v3)
  function1(x,y,z,...)
enddo
```
	```
	do i = m, 2
	 function1_bar()
	  v1 = face(1)(i-1)
	  v2 = face(1)(i-1)
	  v3 = face(1)(i-1)
	  x(1) = vertex(1)(v1)
	  y(1) = vertex(2)(v1)
	  z(1) = vertex(3)(v1)
	  |...|
	  z(3) = vertex(3)(v3)
	enddo
	if (m > 0)
	  function1_bar()
	endif
	```

need iteration aware data fbw analysis, you fi nd them in parallelizing compiler like Open64 or next version of gcc.