Adjoint Data-Flow analyses applied to checkpointing - Tradeoff between snapshots and TBR

Benjamin Dauvergne

Tropics Project, INRIA Sophia-Antipolis

Why checkpoints?

- Instead of recording the tape of the execution, 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 is a classical data flow analysis notation, like $\mathbf{Def}$.

$$\mathbf{Use}(I_1,...,I_n) = \mathbf{Use}(I_1) \cup (\mathbf{Use}(I_2,...,I_n) \setminus \mathbf{Def}(I_1))$$

.

Usual way of doing checkpoints

- By hand :
  we know the code, we know that there is something called the $state$ and it is read and written between checkpoints. We create a procedure which saves it on the tape and we provide it to the AD tool.

- 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 they will be overwritten.

What should we save?

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

$$X = [I_1, \ldots, I_n] \text{ a sequence of instructions}$$

$$\text{adjoint program of } X = \emptyset \vdash \overline{X} \text{ where}$$

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

- I' is the adjoint code associated with a single intruction. When you differentiate you have a context: save set $\mathbf{TBR}$.

The TBR - Snapshot trade off

Bigger TBR

$$
\begin{aligned}
\mathbf{TBR} \vdash \overline{C;D} \quad = \quad & \mathbf{PUSH}\,(\mathbf{Def}(C) \cap \mathbf{TBR}) \\
& \mathbf{PUSH}\left(\mathbf{Def}(C) \cap \mathbf{Use}\left(\overline{C}\right)\right) \\
& C \\
& \left(\mathbf{TBR} \cup \mathbf{Use}\left(\overline{C}\right)\right) \setminus \mathbf{Def}(C) \vdash \overline{D} \\
& \mathbf{POP}\left(\mathbf{Def}(C) \cap \mathbf{Use}\left(\overline{C}\right)\right) \\
& \emptyset \vdash \overline{C} \\
& \mathbf{POP}\,(\mathbf{Def}(C) \cap \mathbf{TBR}))
\end{aligned}
$$

Bigger Snapshot

$$
\begin{aligned}
\mathbf{TBR} \vdash \overline{C;D} \quad = \quad & \mathbf{PUSH}\,(\mathbf{Def}(C) \cap \mathbf{TBR}) \\
& \mathbf{PUSH}\left(\mathbf{Def}(C;D) \cap \mathbf{Use}\left(\overline{C}\right)\right) \\
& C \\
& \mathbf{TBR} \setminus (\mathbf{Def}(C) \cup \mathsf{Snap}) \vdash \overline{D} \\
& \mathbf{POP}\left(\mathbf{Def}(C;D) \cap \mathbf{Use}\left(\overline{C}\right)\right) \\
& \emptyset \vdash \overline{C} \\
& \mathbf{POP}\,(\mathbf{Def}(C) \cap \mathbf{TBR})
\end{aligned}
$$

A code where «big snapshots» are bad

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

In Tapenade we checkpoint all calls so this example is interesting.

A code where «big snapshots» are bad

The forward sweep of preceding code using «big snapshots».

Loop

$$\begin{array}{l|l}
\text{Loop} & \textbf{PUSH}(state) \\
 & proc_1(\text{Use } state, \text{Def } A) \\
 & \textbf{PUSH}(state) \\
 & proc_2(\text{Use } state, \text{Def } B) \\
 & \textbf{PUSH}(state) \\
 & proc_3(\text{Use } state, \text{Def } C) \\
 & \textbf{PUSH}(A, B, C) \\
 & proc_4(\text{Use } ABC, \text{Def } state)
\end{array}$$

It's not really good, each time we save $state$, we save the same values.

A code where «big snapshots» are bad

The forward sweep of preceding code using «big TBR».

Loop

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

Now we are able to remove redundant **PUSH**.

A code where « big TBR » is bad

$proc_1(use = \text{array}A)$

a gather/scatter loop on A

- The forward sweep of preceding code using «big TBR»:

  $proc_1(use = \text{array}A)$

  a gather/scatter loop on A full of $\mathbf{PUSH}(A(i))$

  $\#\mathbf{PUSH} > \mathbf{sizeof}(A)$.

- The forward sweep of preceding code using «big snapshots»:

  $\mathbf{PUSH}(A)$

  $proc_1(use = \text{array}A)$

  a gather/scatter loop on A with less $\mathbf{PUSH}$

Numerical results

On one of our test code using the « big snapshots » scheme:

```
Time of  original  function:    2.26999962300062
Time of tangent AD function:    7.00000000000000
Time of reverse AD function:    25.4899978637695
Max Stack size: 15876 blocks of 16384 bytes
```

with a always « big TBR » scheme :

```
Time of  original  function:    2.28999943226576
Time of tangent AD function:    7.09000152587891
Time of reverse AD function:    22.7300049591064
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

- It is important to look at how you compute your snapshots.
- «big TBR» is the scheme which gives the better result in general.
- If a static analysis can infer that an array is going to be completely written once or more just after, «big snapshots» seems to be appropriate.

Further work

- Find more, easily detectable code patterns, where one or the other scheme is better.
- How could flow dependant data flow informations help us ? i.e specialization at run-time or using profiling.
- Array region analysis.
- The placement of checkpoints in big callgraphs/flowgraphs.