Binomial Checkpointing for Arbitrary Programs with No User Annotation*

Jeffrey Mark Siskind† Barak A. Pearlmutter‡

April 2016

Heretofore, automatic checkpointing at procedure-call boundaries [1], to reduce the space complexity of reverse mode, has been provided by systems like TAPENADE [2]. However, binomial checkpointing, or treeverse [3], has only been provided in AD systems in special cases, e.g., through user-provided pragmas on DO loops in TAPENADE, or as the nested taping mechanism in ADOL-C for time integration processes, which requires that user code be refactored. We present a framework for applying binomial checkpointing to arbitrary code with no special annotation or refactoring required. This is accomplished by applying binomial checkpointing directly to a program trace. This trace is produced by a general-purpose checkpointing mechanism that is orthogonal to AD.

Consider the code fragment in Listing 1. This example, \( y = f(x) \), while contrived, is a simple caricature of a situation that arises commonly in practice, e.g., in adaptive grid methods. Here, the duration of the inner loop varies wildly as some function \( l(x,i) \) of the input and the outer loop index, perhaps \( 2^{l(x,i)} - l(x,(1007)^{x}) \mod n \), that is small on most iterations of the outer loop but \( O(n) \) on a few iterations. Thus the optimality of the binomial schedule is violated. The issue is that the optimality of the binomial schedule holds at the level of primitive atomic computations but this is not reflected in the static syntactic structure of the source code. Often, the user is unaware or even unconcerned with the micro-level structure of atomic computations and does not wish to break the modularity of the source code to expose such. Yet the user may still wish to reap the benefits of an optimal binomial checkpointing schedule [4]. Moreover, the relative duration of different paths through a program may vary from loop iteration to loop iteration in a fashion that is data dependent, as shown by the above example, and not even statically determinable. We present an implementation strategy for checkpointing that does not require user placement of checkpoints and does not constrain checkpoints to subroutine boundaries, DO loops, or other syntactic program constructs. Instead, it can automatically and dynamically introduce a checkpoint at an arbitrary point in the computation that need not correspond to a syntactic program unit.

We have previously introduced VLAδ, a pure functional language with builtin AD operators for both forward and reverse mode. Here, we adopt slight variants of these operators with the following signatures.

\[ \mathcal{F} : f \times \hat{x} \mapsto y \hat{y} \quad \mathcal{F} : f \times y \mapsto y \hat{x} \]

The \( \mathcal{F} \) operator calls a function \( f \) on a primal \( x \) with a tangent \( \hat{x} \) to yield a primal \( y \) and a tangent \( \hat{y} \). The \( \mathcal{F} \) operator calls a function \( f \) on a primal \( x \) with a cotangent \( \hat{y} \) to yield a primal \( y \) and a cotangent \( \hat{x} \). Here, we restrict ourselves to the case where (co)tangents are ground data values, i.e., reals and (arbitrary) data structures containing reals and other scalar values, but not functions (i.e., closures). For our purposes, the crucial aspect of the design is that the AD operators are provided within the language, since these provide the portal to the checkpointing mechanism.

In previous work, we introduced StalinV, a highly optimizing compiler for VLAδ. Here, we formulate a simple evaluator (interpreter) for VLAδ (Fig. 1) and extend such to perform binomial checkpointing. The operators \( \circ \) and \( \bullet \) range over the unary and binary basis functions respectively. This evaluator is written in what is known in the programming-language community as direct style, where functions (in this case \( \mathcal{E} \), denoting ‘eval’, \( \mathcal{A} \), denoting ‘apply’,

---

*Extended abstract presented at the AD 2016 Conference, Sep 2016, Oxford UK.
†Corresponding Author, School of Electrical and Computer Engineering, Purdue University, qobi@purdue.edu
‡Dept of Computer Science, National University of Ireland Maynooth, barak@pearlmutter.net

Listing 1: FORTRAN example

```fortran
subroutine f(x, y)
  n = 100003
  y = x
  c$ad binomial-ckp n+1 30 1
do i = 1, n
  m = l(x, i)
do j = 1, m
y = y*y
end do
end
```

Figure 1: Direct-style evaluator for VLAδ.
and the implementations of $\overrightarrow{J}$ and $\overleftarrow{J}$ in the host) take inputs as function-call arguments and yield outputs as function-call return values [5]. AD is performed by overloading the basis functions in the host, in a fashion similar to FADBAD++ [6], $x \times x$ denotes recursively bundling a data structure containing primals with a data structure containing tangents, or alternatively recursively unbundling such when used as a binder, and $y \triangleq \overleftarrow{y}$ denotes running the reverse sweep on the tape $y$ with the output cotangent $\overleftarrow{y}$, or alternatively extracting the primal $y$ and input cotangent $\overrightarrow{x}$ from the tape when used as a binder $y \triangleq \overrightarrow{\times}$.

We introduce a new AD operator $\tilde{J}$ to perform binomial checkpointing. The crucial aspect of the design is that the signature (and semantics) of $\tilde{J}$ is identical to $\overrightarrow{J}$; they are completely interchangeable, differing only in the space/time complexity tradeoffs. This means that code need not be modified to switch back and forth between ordinary reverse mode and binomial checkpointing, save interchanging calls to $\overrightarrow{J}$ and $\tilde{J}$.

Conceptually, the behavior of $\tilde{J}$ is shown in Fig. 2. In this inductive definition, a function $f$ is split into the composition of two functions $g$ and $h$ in step 1, the checkpoint $u$ is computed by applying $g$ to the input $x$ in step 2, and the cotangent is computed by recursively applying $\tilde{J}$ to $h$ and $g$ in steps 3 and 4. This divide-and-conquer behavior is terminated in a base case, when the function $f$ is small, at which point the cotangent is computed with $\overrightarrow{J}$, in step 0. If step 1 splits a function $f$ into two functions $g$ and $h$ that take the same number of computational steps, the recursive divide-and-conquer process yields the logarithmic asymptotic space/time complexity of binomial checkpointing.

The central difficulty in implementing the above is performing step 1, namely splitting a function $f$ into two functions $g$ and $h$, ideally ones that take the same number of computational steps. A sophisticated user can manually rewrite a subroutine $f$ into two subroutines $g$ and $h$. A sufficiently powerful compiler or source transformation tool might also be able to, with access to nonlocal program text. But an overloading system, with access only to local information, would not be able to.

We solve this problem by providing an interface to a general-purpose checkpointing mechanism orthogonal to AD.

$$\text{PRIMOPS } f \ x \to \ (y,n) \quad \text{Return } y = f(x) \text{ along with the number } n \text{ of steps needed to compute } y.$$

$$\text{CHECKPOINT } f \ x \ n \to u \quad \text{Run the first } n \text{ steps of the computation of } f(x) \text{ and return a checkpoint } u.$$

$$\text{RESUME } u \to y \quad \text{If } u = (\text{CHECKPOINT } f \ x \ n), \text{ return } y = f(x).$$

This interface allows (a) determining the number of steps of a computation, (b) interrupting a computation after a specified number of steps, usually half the number of steps determined by the mechanism in (a), and (c) resuming an interrupted computation to completion. A variety of implementation strategies for this interface are possible. We present one in detail momentarily and briefly discuss others below.

Irrespective of how one implements the general-purpose checkpointing interface, one can use it to implement $\tilde{J}$ as shown in Fig. 3. The function $f$ is split into the composition of two functions $g$ and $h$ by taking $g$ as $\lambda x.\text{CHECKPOINT } f \ x \ n$, where $n$ is half the number of steps determined by $\text{PRIMOPS } f \ x$, and $h$ as $\lambda u.\text{RESUME } u$.

One way of implementing the general-purpose checkpointing interface is to convert the evaluator from direct style to continuation-passing style (CPS, [7]), where functions (in this case $\mathcal{E}$, $\mathcal{A}$, $\overrightarrow{J}$, and $\overleftarrow{J}$ in the host) take an additional continuation input $k$ and instead of yielding outputs via function-call return, do so by calling the continuation with said output as arguments (Fig. 5). In such a style, functions never return; they just call their continuation. With tail-call merging, such corresponds to a computed go to and does not incur stack growth. This crucially allows the interruption process to actually return a checkpoint data structure containing the saved state of the evaluator, including its continuation, allowing the evaluation to be resumed by calling the evaluator with this saved state. This ‘level shift’ of return to calling a continuation allowing an actual return to constitute checkpointing interruption is analogous to the way backtracking is classically implemented in Prolog, with success implemented as calling a continuation and failure implemented as actual return. In our case, we further instrument the evaluator to thread two values as inputs and outputs: the count $n$ of the number of evaluation steps, which is incremented at each call to $\mathcal{E}$, and the limit $l$ of the number of steps, after which a checkpointing interrupt is triggered.

With this CPS evaluator, it is possible to implement the general-purpose checkpointing interface (Fig. 4), not for programs in the host, but for programs in the target; hence our choice of formulating the implementation around an evaluator (interpreter).

```
PRIMOPS f x \to \ (y,n)  
CHECKPOINT f x \ n \to u  
RESUME u \to y  
```

To compute $(y,\overrightarrow{x}) = \overrightarrow{J} f \ x \ y$:

**base case (f x fast):**

$$f (x) = \overleftarrow{J} f x y$$

**inductive case:**

$$h \circ g = f$$

$$u = g x$$

$$h u y$$

$$(u,\overrightarrow{x}) = \overrightarrow{J} g x u$$

Figure 2: Algorithm for binomial checkpointing.

To compute $(y,\overleftarrow{x}) = \overleftarrow{J} f x y$:

**base case:**

$$f (x) = \overrightarrow{J} f x y$$

**inductive case:**

$$\lambda x.\text{PRIMOPS } f \ x$$

$$u = \text{CHECKPOINT } f \ x \ n$$

$$(y,\overrightarrow{u}) = \overrightarrow{J} (\lambda u.\text{RESUME } u) u y$$

$$(u,\overrightarrow{x}) = \overleftarrow{J} (\lambda x.\text{CHECKPOINT } f \ x \ n) u \overrightarrow{x}$$

Figure 3: Binomial checkpointing via general checkpointing interface.

```
PRIMOPS f x = A (λ n l v.(v, n)) 0 \infty f x 
CHECKPOINT f x \ n = A 1 0 n f x 
RESUME [k,l,p,e] = E k l \infty p e
```

Figure 4: Implementation of the general-purpose checkpointing interface using the CPS evaluator.
We remove this restriction below. The implementation of PRIME- MOPS calls the evaluator with no limit and simply counts the number of steps to completion. The implementation of CHECKPOINT calls the evaluator with a limit that must be smaller than that needed to complete so a checkpointing interrupt is forced and the checkpoint data structure must be updated. We remove this restriction below. The implementation of PRIORITY (Listing 2) was run without checkpointing interruptions. A driver can be wrapped around such code to implement \( J \). Existing high-performance compilers, like SML/NJ [9], for functional languages like SML, already generate target code in CPS, so by adapting such to the purpose of AD with binomial checkpointing, it seems feasible to achieve high performance. In fact, the overhead of the requisite instrumentation for step counting, step limits, and checkpointing interruptions need not be onerous because the step counting, step limits, and checkpointing interruptions for basic blocks can be factored, and those for loops can be hoisted, much as is done for the instrumentation needed to support storage allocation and garbage collection in implementations like MLTON [10], for languages like SML, that achieve very low overhead for automatic

\[
A k l ((\lambda x . e), \rho) = E k l \rho[x \mapsto e] c
\]

\[
J k l n v_1 v_2 \hat{v}_3 = A (\lambda n l v_1 v_2) k n l v_1 v_2 \hat{v}_3
\]

\[
J k l n v_1 v_2 \hat{v}_3 = A (\lambda n l v_1 \text{let} (v_4 \leftarrow \hat{v}_3) = v \in v_3 \text{in} k n l v_4 \hat{v}_3) n l v_1 v_2
\]

\[
E k l n p e = [k, l, p, e]
\]

\[
E k n l p x = k (n + 1) l c
\]

\[
E k n l p x = k (n + 1) l (p x)
\]

\[
E k n l p (\lambda x . e) = k (n + 1) l ((\lambda x . e), \rho)
\]

\[
E k n l p (e_1 \ e_2) = E (\lambda n l v_1 . (E (\lambda n l v_2 . (A k n l v_1 v_2)) n l p e_2)) (n + 1) l r e_1
\]

\[
E k n l p (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) = E (\lambda n l v_1 . (\text{if } e_1 \text{ then } (E k n l p e_2) \text{ else } (E k n l p e_3))) (n + 1) l r e_1
\]

\[
E k n l p (\text{let } \alpha \to e \text{ in } \beta) = (E k n l p \alpha) (E k n l p e) (n + 1) l r e_1
\]

\[
E k n l p (\text{let } (\alpha \to e) \text{ in } \beta) = E (\lambda n l v_1 . (E (\lambda n l v_2 . (E (\lambda n l v_3 . (J k n l v_1 v_2 v_3)) n l p e_3)) n l p e_2)) (n + 1) l r e_1
\]

\[
E k n l p (\text{let } (\alpha \to e) \text{ in } \beta) = E (\lambda n l v_1 . (E (\lambda n l v_2 . (E (\lambda n l v_3 . (J k n l v_1 v_2 v_3)) n l p e_3)) n l p e_2)) (n + 1) l r e_1
\]
Figure 5: Space and time usage of reverse-mode AD with various checkpointing strategies, relative to the space and time for the first datapoint for each respective strategy.

storage management.

Acknowledgments

This work was supported, in part, by NSF grant 1522954-IIS and by Science Foundation Ireland grant 09/IN.1/I2637. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References


