AD source transformation & Performance Metrics

Ian Karlin (U of Colorado; summer student at Argonne)
Jean Utke(Argonne, MCS)

- what is automatic differentiation (AD)
- how is AD done with source transformation
- what variations need metrics
- how far have we come
- open issues

4 motivations for AD

! some numerical model given as a (large) program

? sensitivity analysis, optimization, parameter (state) estimation

1. don't pretend we know nothing about the program (and take finite differences of an oracle?)

2. get machine precision derivatives (avoid approximation-versus-rounding problem)

3. the reverse mode (adjoint) yields "cheap" gradients

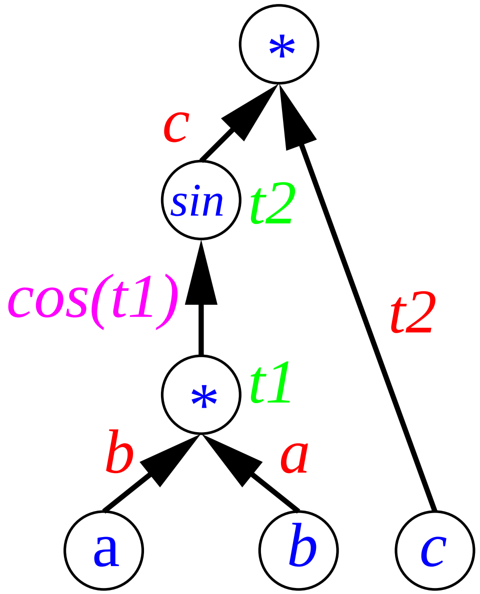
4. if the program is large, so is the adjoint, so is the effort to do it manually ... and it is easy to get wrong but hard to debug

get a tool to do it "automatically"

example - how do directional derivatives come about?

$f : y = sin(a * b) * c$

yields a graph representing the order of computation:



- intrinsics $\phi(\ldots, w, \ldots)$ have local partial derivatives $\frac{\partial\phi}{\partial w}$
- e.g. `sin(t1)` yields `cos(t1)`
- *code list*$\longrightarrow$ intermediate values $t1$ and $t2$
- all others already stored in variables
- data and statement-level code augmentation

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y  = t2*c
```

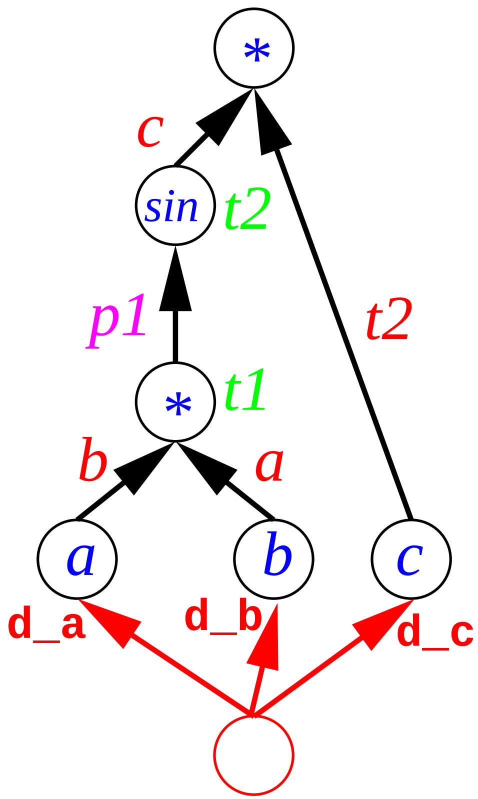
What can we do with this?

forward with directional derivatives

$f(g(x)) \Rightarrow \dot{f}(g(x))\dot{g}(x)\dot{x}$ multiplications along paths

Assume a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$ =`(d_a,d_b,d_c)`

variable and directional derivatives associated in pairs `(v,d_v)`:



`d_a*b*p1*c+d_b*a*p1*c+d_c*t2`

has common subexpressions

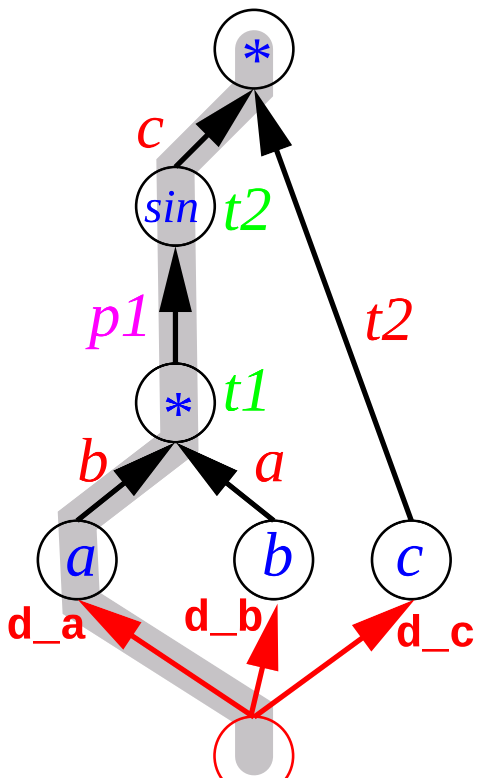
interleave computations of directional derivatives

```
t1   = a*b
d_t1 = d_a*b + d_b*a
p1   = cos(t1)
t2   = sin(t1)
d_t2 = d_t1*p1
y    = t2*c
d_y  = d_t2*c + d_c*t2
```

What is in `d_y`?

note: graph-level code augmentation

forward with directional derivatives II



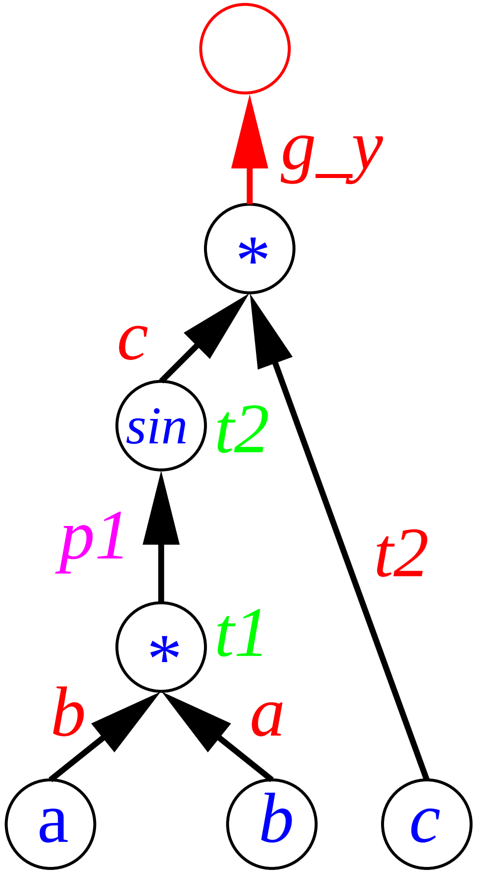
- if $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$ then d_y$=\frac{\partial f}{\partial a}(a_0, b_0, c_0)$

```
t1   = a*b
d_t1 = d_a*b + 0*a
p1   = cos(t1)
t2   = sin(t1)
d_t2 = d_t1*p1
y    = t2*c
d_y  = d_t2*c + 0*t2
```

- 3 directions give $\nabla f(a_0, b_0, c_0)$ and d_y$=\nabla f^T(\dot{a}, \dot{b}, \dot{c}) = \nabla f^T \dot{x}$
- floating point accuracy for derivative calculation !
- gradient calculation cost $\sim n$

reverse with adjoints

Assume variable and adjoints associated in pairs (v,g_v):



append computations of adjoints

```
t1 = a*b
p1 = cos(t1)    // push(p1)
t2 = sin(t1)
y  = t2*c
g_c  = g_y*t2
g_t2 = g_y*c
g_t1 = g_t2*p1  // pop()
g_b  = g_t1*a
g_a  = g_t1*b
```

What is in (g_a,g_b,g_c)? If g_y=1, then $\nabla f(a_0, b_0, c_0)$!

. notice the lifetime of p1 ⇒ insert stack operations

transformation elements so far ...

- $\surd$ data augmentation
- $\surd$ linearized model - by intrinsic
- $\surd$ accumulate derivatives - given the computational graph(s)
- $\surd$ stack for certain values used in the adjoint computation

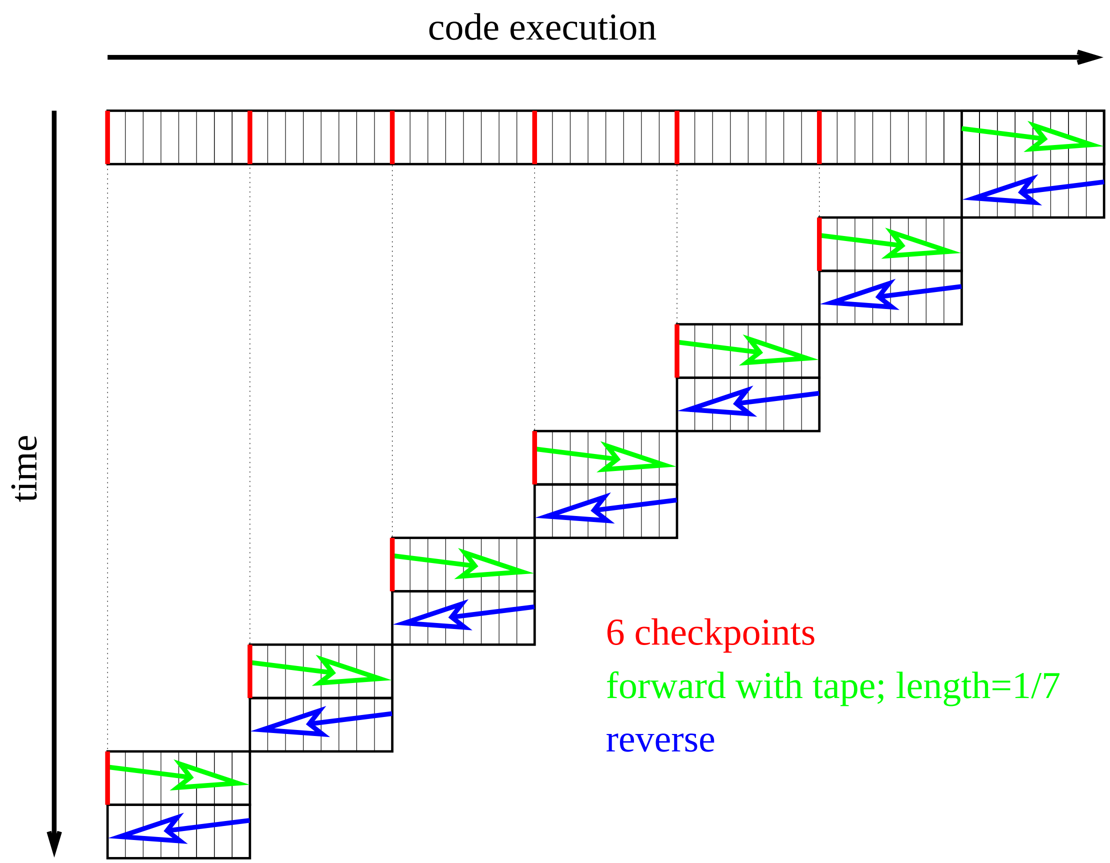
control flow / subroutine calls?

- graph sequence
  - elimination in graphs to bipartite $\cong$ preaccumulation of local Jacobians
  - propagation of Jacobians $\cong$ chained sparse matric product
- explains the necessity of the value stack
- transformations for adjoint code cover control flow & call sequence reversal

Problem: value stack size $\sim$ problem size & runtime

$\Rightarrow$ trade-off stack size (memory) for storing checkpoints (less memory) and recomputations from checkpoints (extra runtime)

trade memory consumption for recomputation



- checkpoint placement
- determine checkpoint contents using side effect analysis
- hierarchal checkpoints
- estimating checkpoint size vs. tape size reductions
- control irregular checkpointing schemes

variations for the source transformation (1)

variations are hierarchical, starting with the lowest level
the elimination order in the computational graph

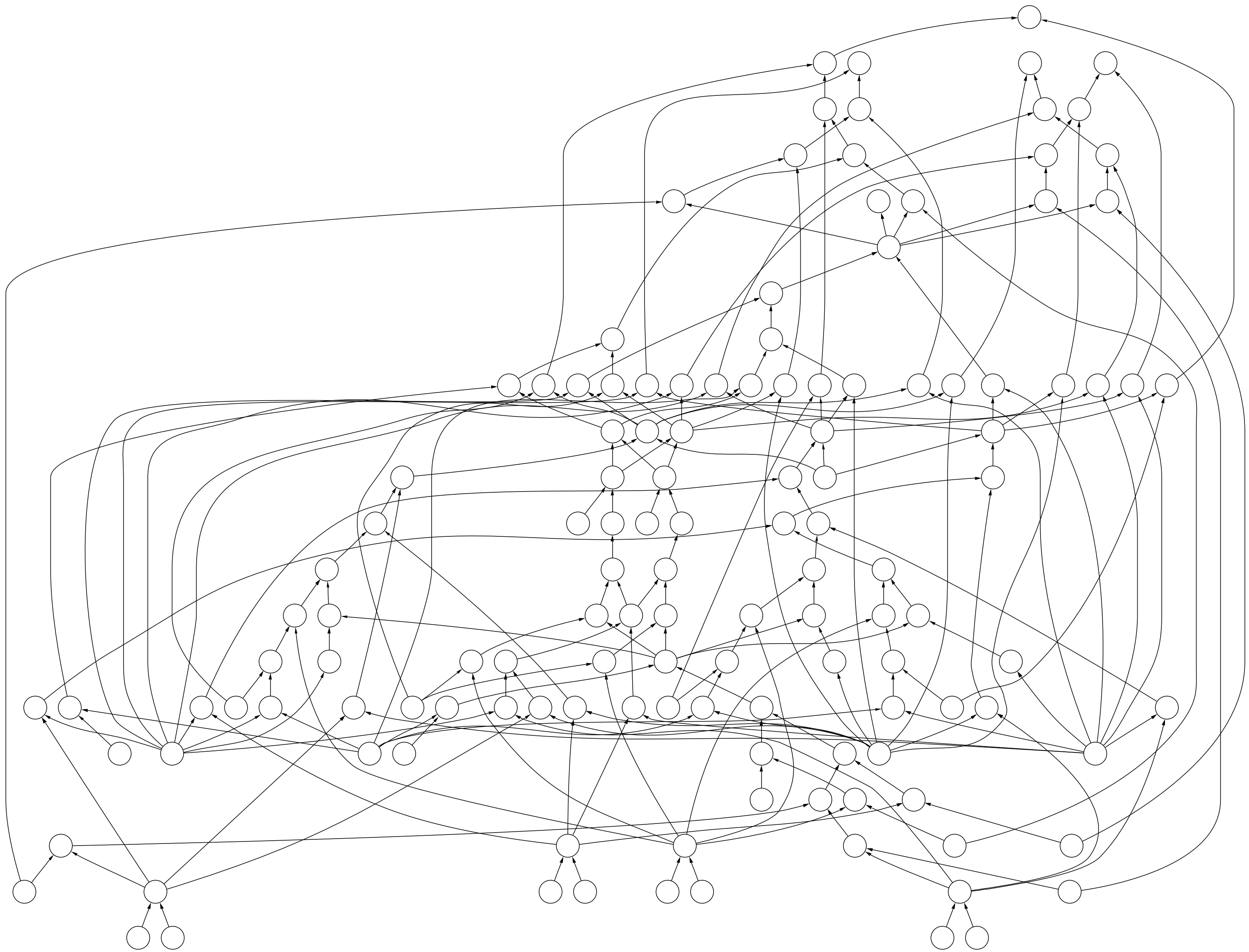
- flop counts
- optimal solution known for single-expression-use graphs
- np-hard $\Rightarrow$ comparing various heuristics

(*,+)

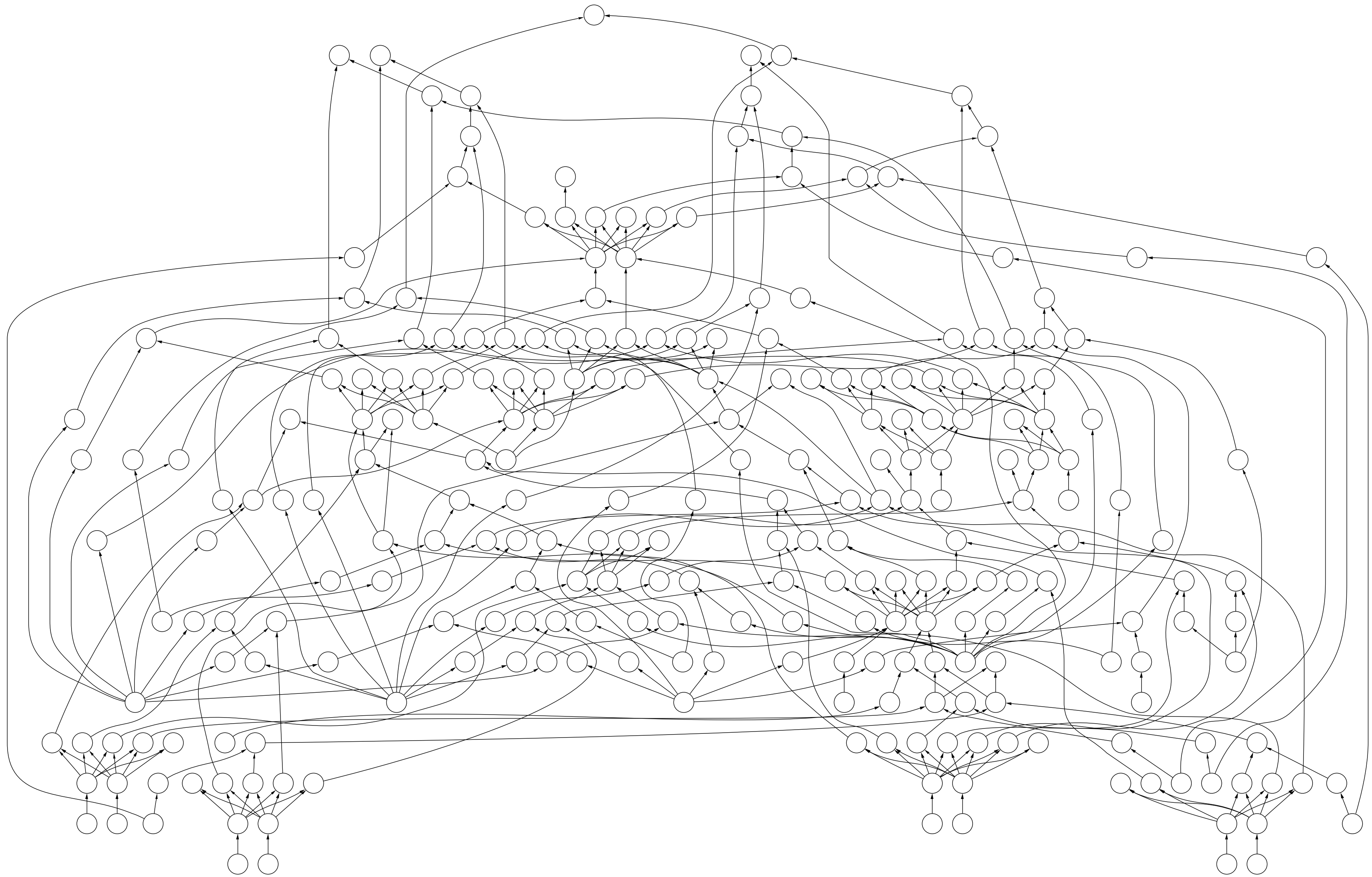
Test	Default	Vertex	Face	Switching	Savings	time
RoehFlux	1469, 370	1245, 170	1400, 168	1217, 181	17.2%, 51.1%,	4.5%
RoehFlux_MO	2121, 1088	1175, 415	1495, 461	1175, 415	44.6%, 61.9%,	29.4%
mini1	450, 199	410, 91	383, 88	383, 88	14.9%, 55.8%	
todd1	1461, 738	909, 254	1182, 316	909, 254	37.8%, 65.6%,	19.7%

- for smaller basic blocks and problem significant savings still occur though the largest savings are found in the largest basic blocks

example computational graph



corresponding dual graph



variations for the source transformation (2)

next higher level is the scope of the individual computational graphs

- smaller scope ⇒ fewer flops for elimination within graphs
- smaller scope ⇒ more graphs in sequence, suspect more flops for propagation
- varying scope ⇒ varying number of nonzeros in sparse local Jacobian
- two practical choices, statement level (known optimal elimination) and maximal graphs

test	basicblock ??			statement ??			switching ??			Δ time
	*	+	J_{ij}	*	+	J_{ij}	*	+	J_{ij}	
RoehFlux	1217	181	615	308	0	278	310	0	277	28.9%
dfdcfj	103	5	34	91	5	41	103	5	34	-48.1%
todd1	909	254	280	75	1	165	75	1	165	-1.6%

- there can be small advantages to switching at the block level
- big advantage is that the better of statement level or block is picked without user effort

variations for the source transformation (3)

next higher level is the checkpoint placement for an adjoint code

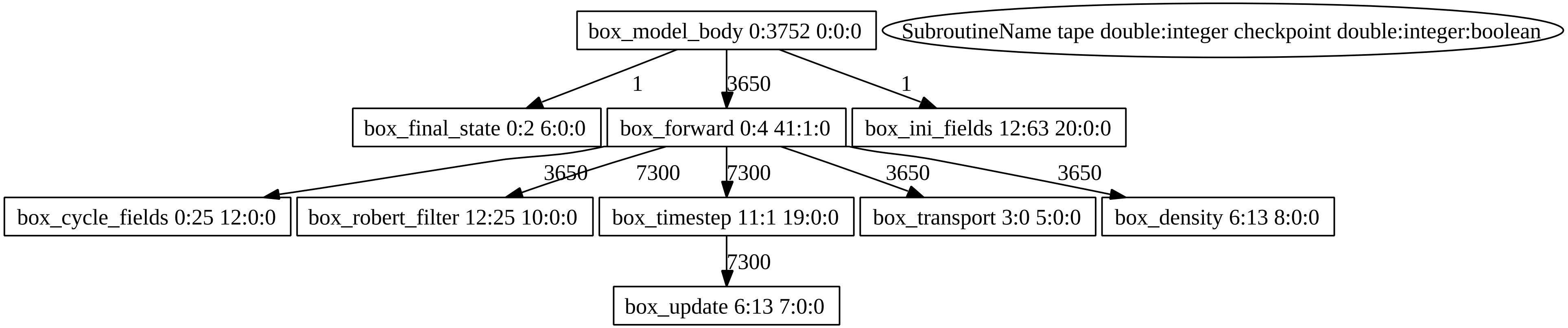
- lower level choices determine value stack growth rate
- checkpoints close enough to keep stack size limited
- checkpoint size

transformation-time information by itself insufficient, need profile data because

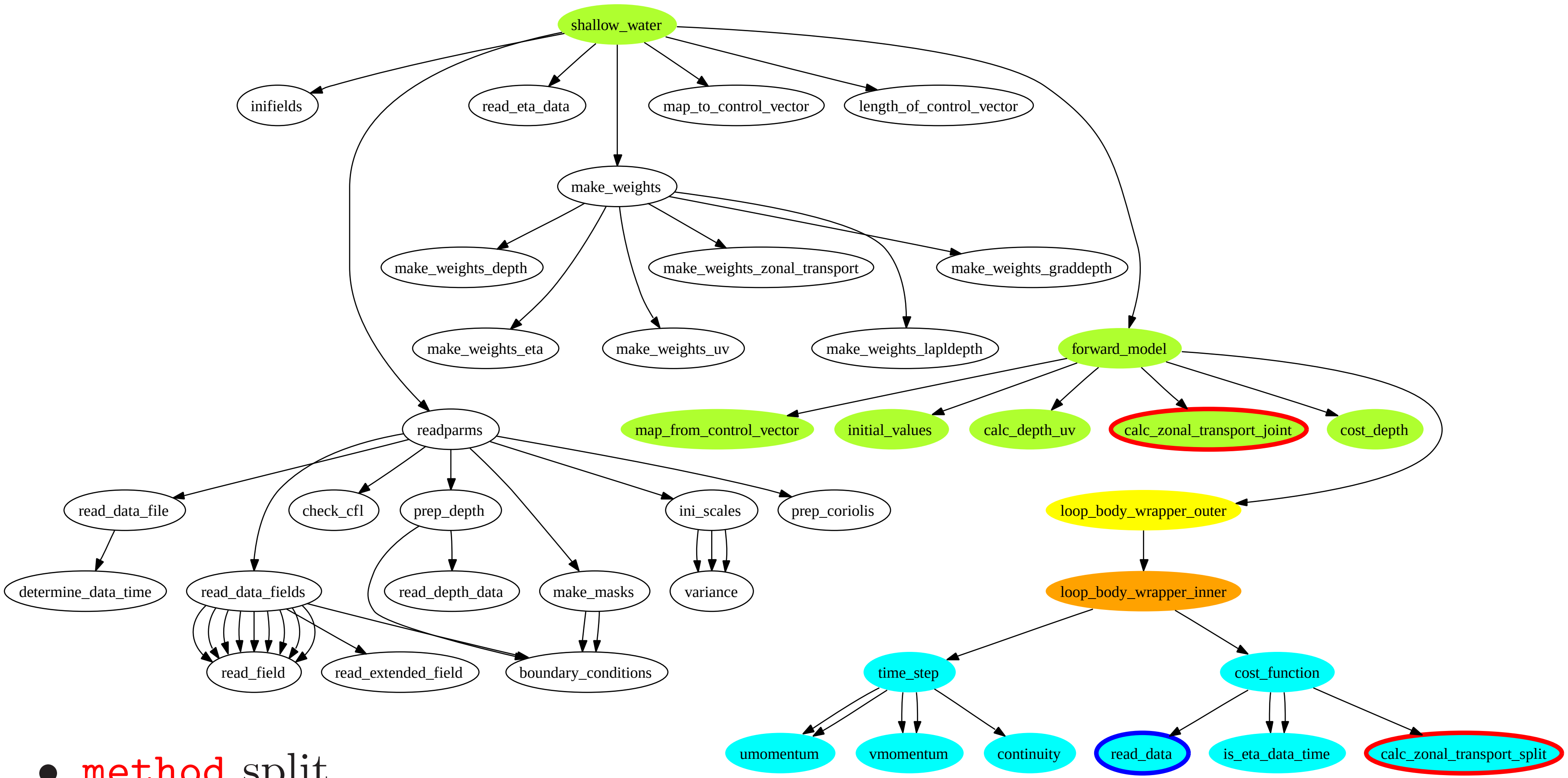
- problem size impacts checkpoint size
- loop iteration counts impact stack size
- code generation inserts optional profiling code

checkpoint related transformation data is still useful.

checkpoint information



checkpoint information (2)



- method split
- outer and inner checkpoints
- data visibility

conclusions

- variations useful in practice
  (implemented in OpenAD, see `www.mcs.anl.gov/openad`)
- automatic heuristics consistent with runtime results
- transformation complexity is increasing
  - variants have a huge domain
  - transformation environment stays close to compiler IR
  - few abstractions possible at a higher level
  - e.g. the checkpointing scheme on the dynamic call tree
- gaining some insight into profile feedback to transformations (with user-hints)
- Can the AD niche tools and the general purpose STSs meet somewhere?

AD community

- most active: Germany (Aachen, Berlin, Dresden, Hamburg), US (Argonne, Rice U, MSU, Sandia), UK (Cranfield, RAL, Hatfield), France (INRIA)
- connections to application areas (engineering, oceanography, meteorology), numerical optimization, compiler research
- not many connections to "other" source transformation fields
- common problems: **stable and up-to-date** parsing/unparsing environments, advanced compiler analyses
- high level IR, but want e.g. type analysis
- semantic enhancements
- AD community has two informal 2 day workshops per year (next one is Dec 7/8 in Aachen), various workshops/minisymposia attached to conferences, the 5th International AD conference in 2008.
- community website: `www.autodiff.org`