

Recompute vs. Store for Adjoint

Jean Utke[?]

[?]University of Chicago or Argonne National Laboratory

Apr/4/2012



UChicago ►
Argonne_{LLC}



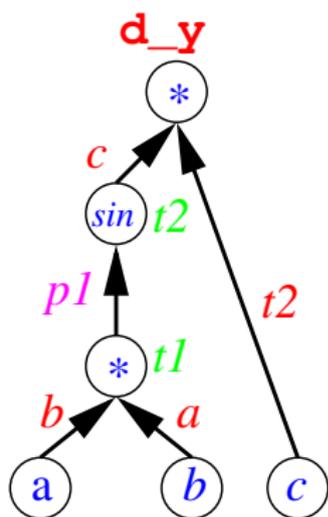
overview

context: computing adjoints via *algorithmic* differentiation (AD)

- background
 - storing checkpoints / “tape”
 - performance implications
 - what to recompute
 - store all vs recompute all
 - existing approaches
- new approach
 - as a graph problem
 - cost function
 - heuristic
 - implementation
- observations and summary

reverse mode with adjoints

- name/address association model
- take a point (a_0, b_0, c_0) , compute y , pick a weight \bar{y}
- for each $v = \phi(w, u)$ propagate backward
 $\bar{w}_+ = \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u}_+ = \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$

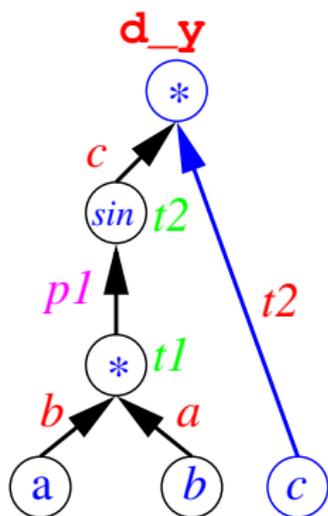


backward propagation code appended:

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

reverse mode with adjoints

- name/address association model
- take a point (a_0, b_0, c_0) , compute y , pick a weight \bar{y}
- for each $v = \phi(w, u)$ propagate backward
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$



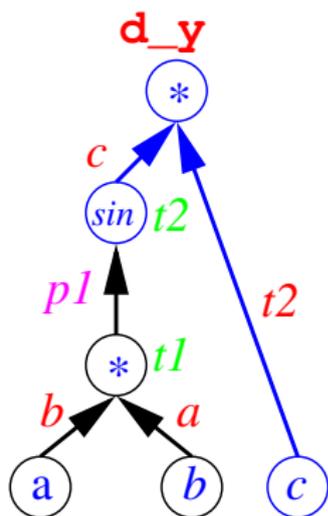
backward propagation code appended:

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

reverse mode with adjoints

- name/address association model
- take a point (a_0, b_0, c_0) , compute y , pick a weight \bar{y}
- for each $v = \phi(w, u)$ propagate backward

$$\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$$

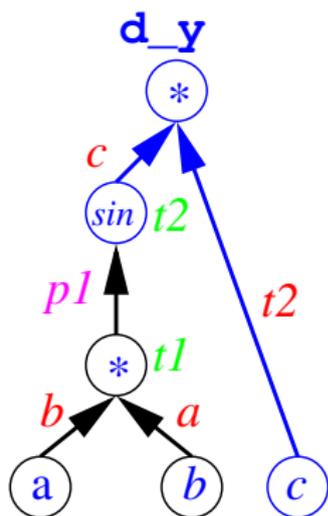


backward propagation code appended:

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

reverse mode with adjoints

- name/address association model
- take a point (a_0, b_0, c_0) , compute y , pick a weight \bar{y}
- for each $v = \phi(w, u)$ propagate backward
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

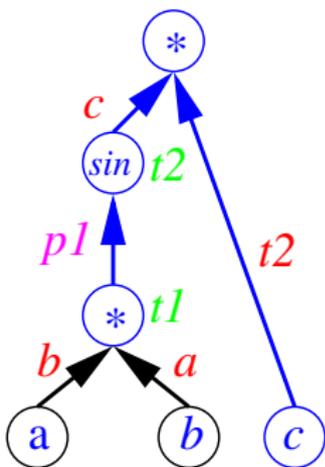


backward propagation code appended:

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

reverse mode with adjoints

- name/address association model
- take a point (a_0, b_0, c_0) , compute y , pick a weight \bar{y}
- for each $v = \phi(w, u)$ propagate backward
 $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u} += \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$

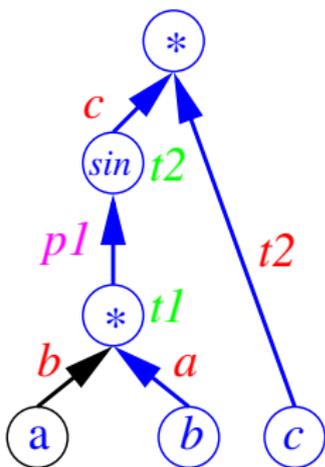


backward propagation code appended:

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

reverse mode with adjoints

- name/address association model
- take a point (a_0, b_0, c_0) , compute y , pick a weight \bar{y}
- for each $v = \phi(w, u)$ propagate backward
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$

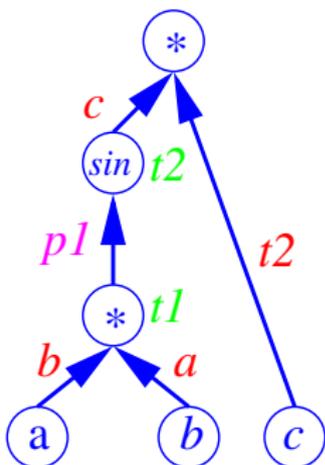


backward propagation code appended:

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

reverse mode with adjoints

- name/address association model
- take a point (a_0, b_0, c_0) , compute y , pick a weight \bar{y}
- for each $v = \phi(w, u)$ propagate backward
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$

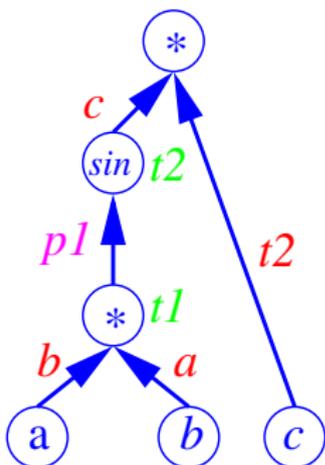


backward propagation code appended:

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

reverse mode with adjoints

- name/address association model
- take a point (a_0, b_0, c_0) , compute y , pick a weight \bar{y}
- for each $v = \phi(w, u)$ propagate backward
 $\bar{w}+ = \frac{\partial \phi}{\partial w} \bar{v}$; $\bar{u}+ = \frac{\partial \phi}{\partial u} \bar{v}$; $\bar{v} = 0$



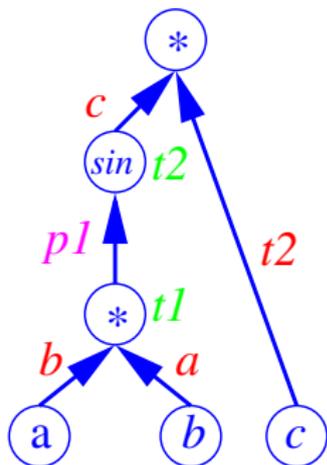
backward propagation code appended:

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

What is in (d_a, d_b, d_c) ?

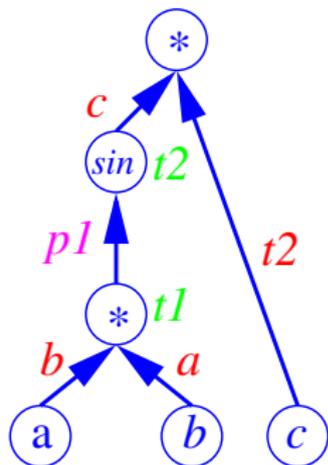
(d_a, d_b, d_c) contains a projection

- $\bar{x} = \bar{y}^T J$ computed at x_0



(d_a, d_b, d_c) contains a projection

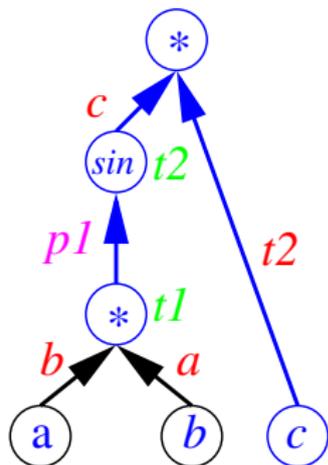
- $\bar{x} = \bar{y}^T J$ computed at x_0
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $O(1)$ function evaluations

(d_a, d_b, d_c) contains a projection

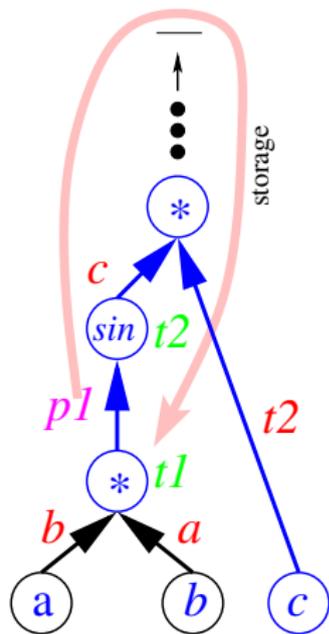
- $\bar{x} = \bar{y}^T J$ computed at x_0
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $O(1)$ function evaluations
- but consider when $p1$ is computed and when it is used

(d_a, d_b, d_c) contains a projection

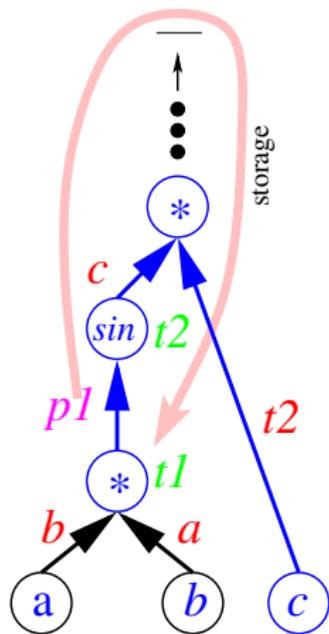
- $\bar{x} = \bar{y}^T J$ computed at x_0
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $O(1)$ function evaluations
- but consider when $p1$ is computed and when it is used
- **storage requirements** grow with the length of the computation
- typically mitigated by recomputation

(d_a, d_b, d_c) contains a projection

- $\bar{x} = \bar{y}^T J$ computed at x_0
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $O(1)$ function evaluations
- but consider when $p1$ is computed and when it is used
- **storage requirements** grow with the length of the computation
- typically mitigated by recomputation

recomputation at different levels

high-level recomputation = checkpointing



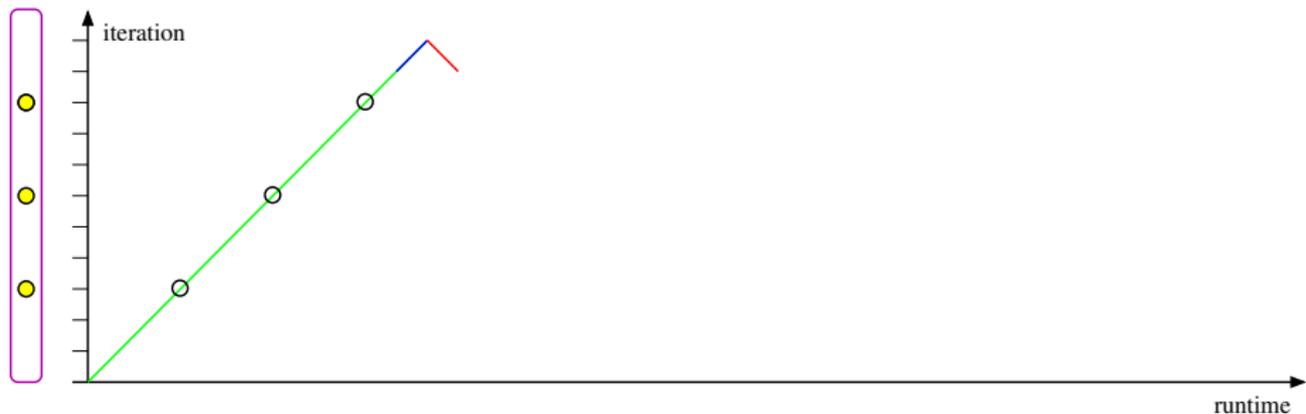
- 11 iters.

high-level recomputation = checkpointing



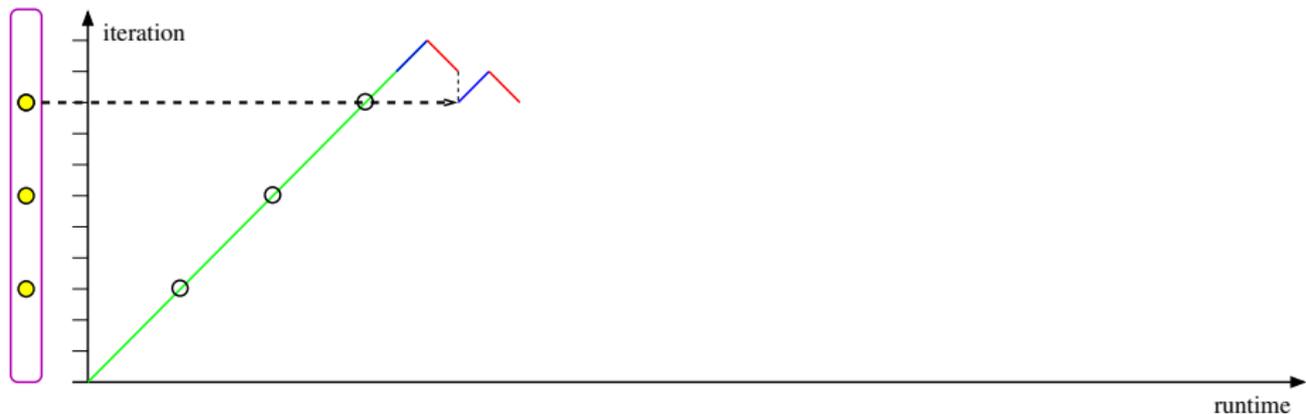
- 11 iters., memory limited to one iter. of storing J_i
- run forward, store = “tape” the last step, and **adjoin**

high-level recomputation = checkpointing



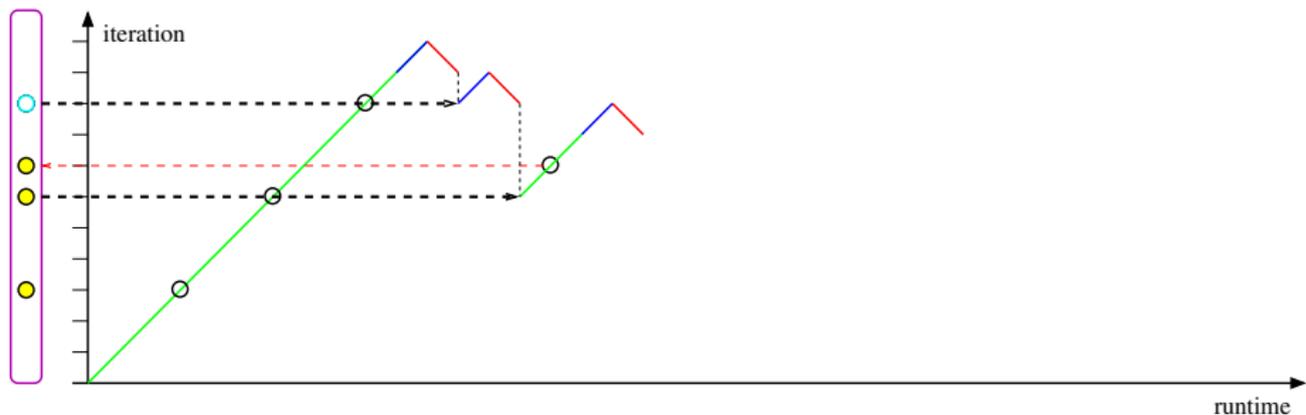
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and **adjoin**

high-level recomputation = checkpointing



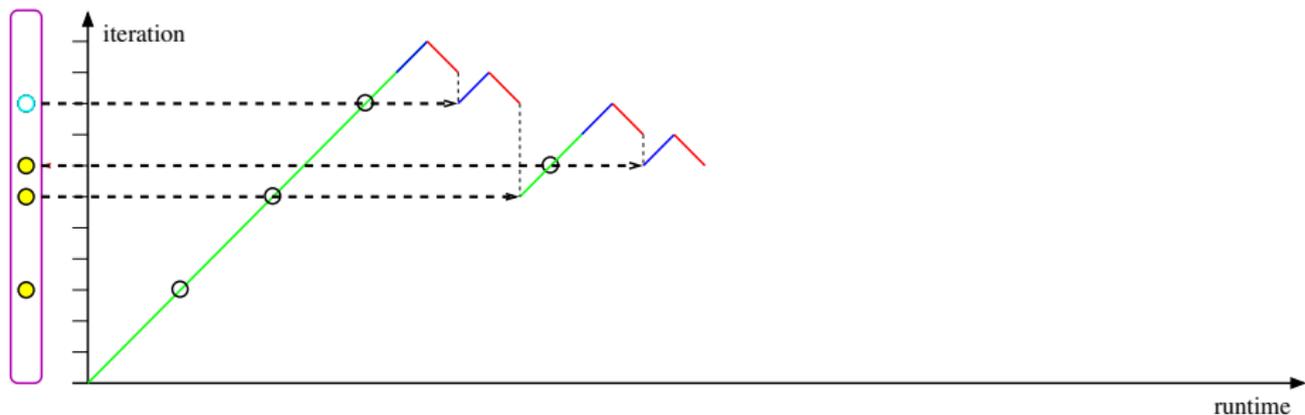
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and **adjoin**
- restore checkpoints and recompute

high-level recomputation = checkpointing



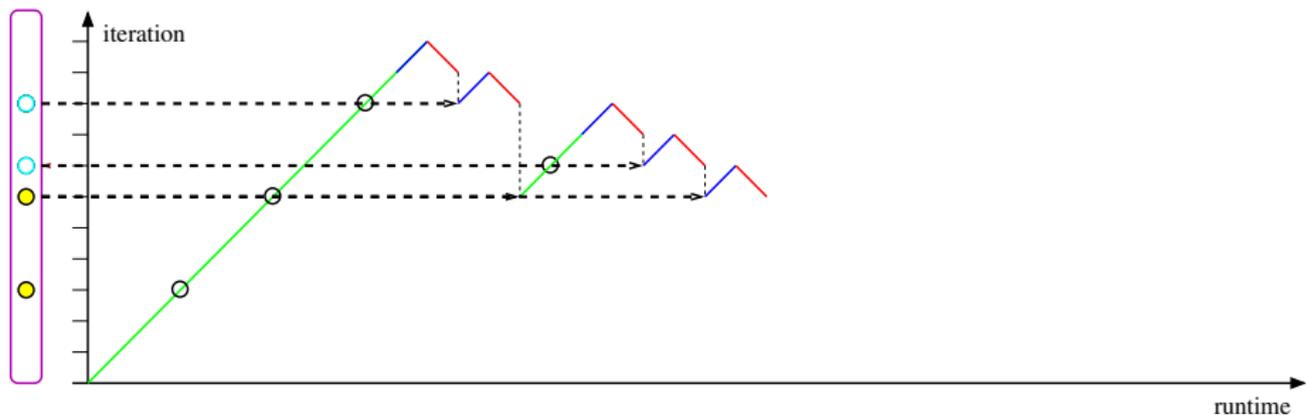
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



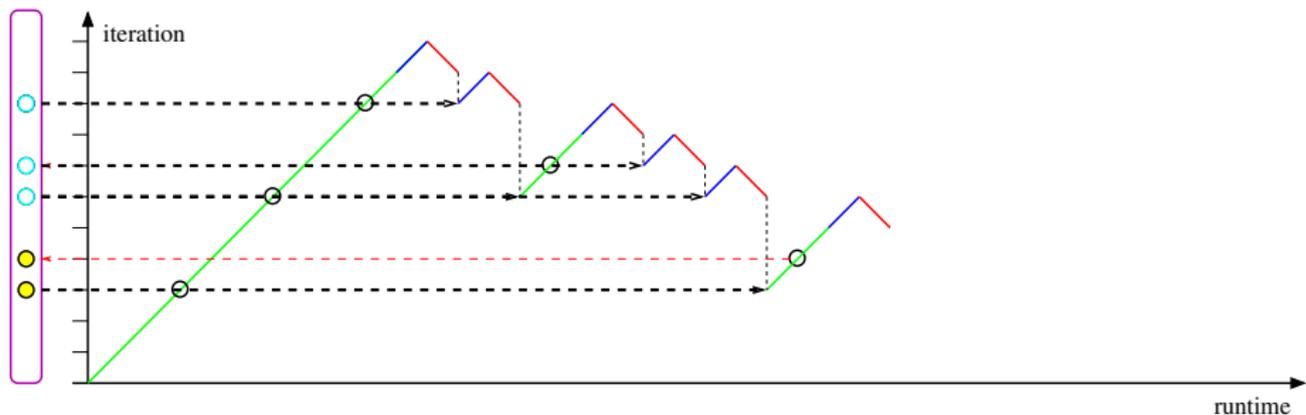
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and **adjoin**
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



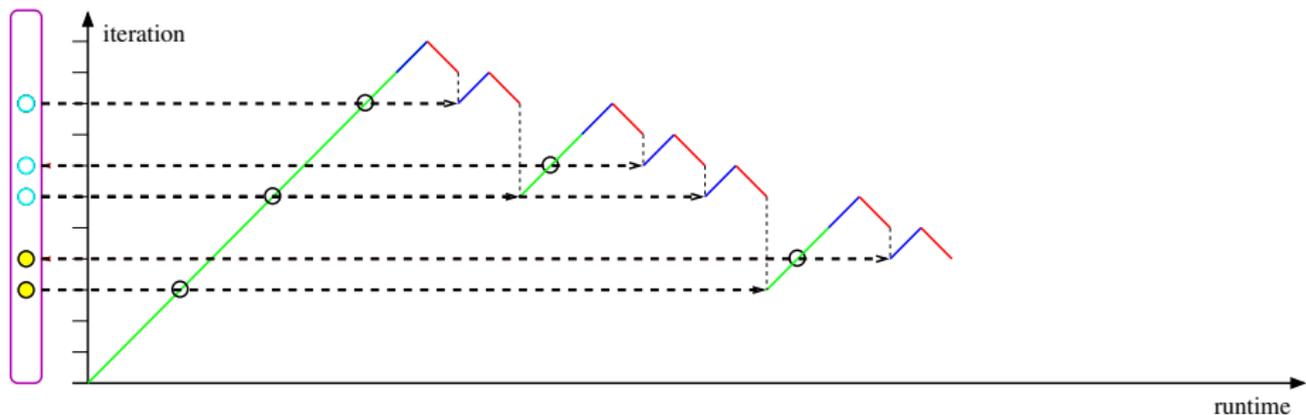
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and **adjoin**
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



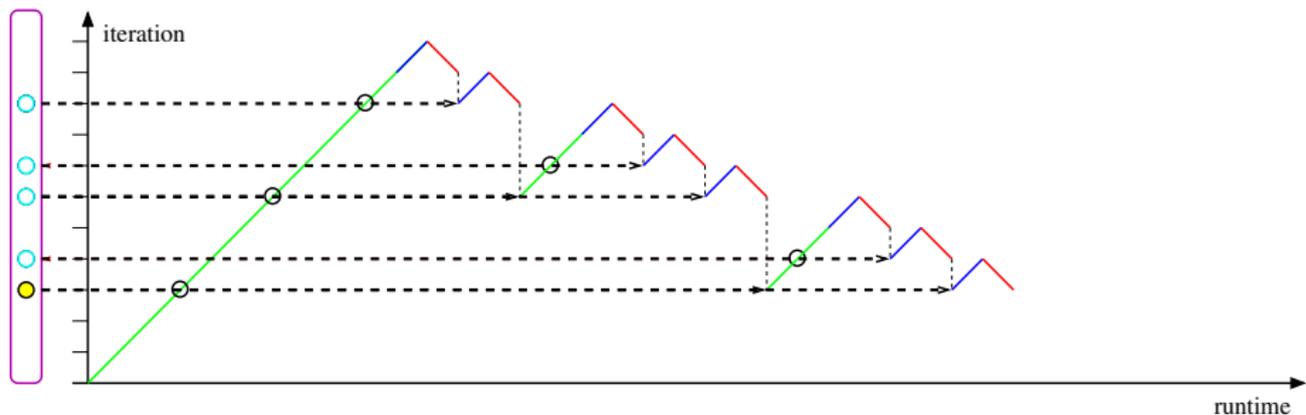
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



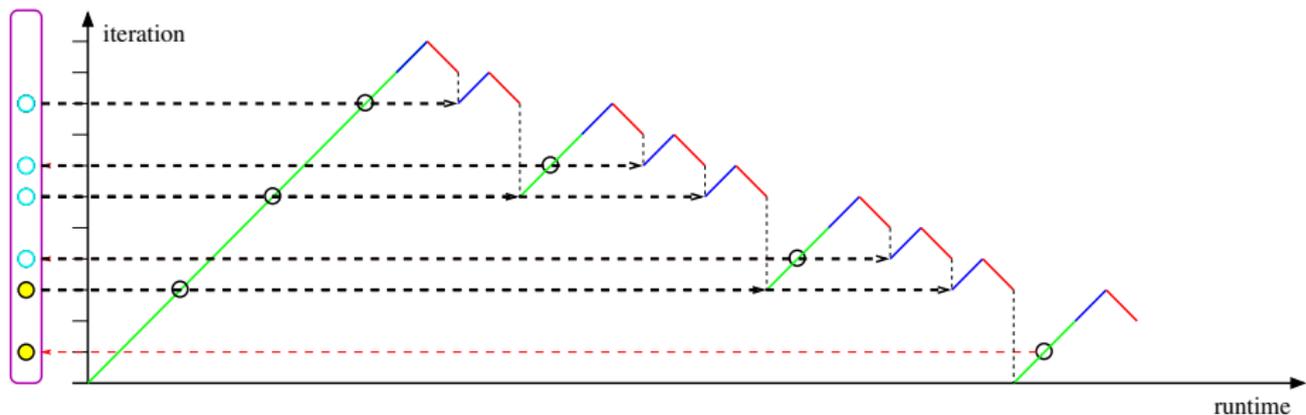
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



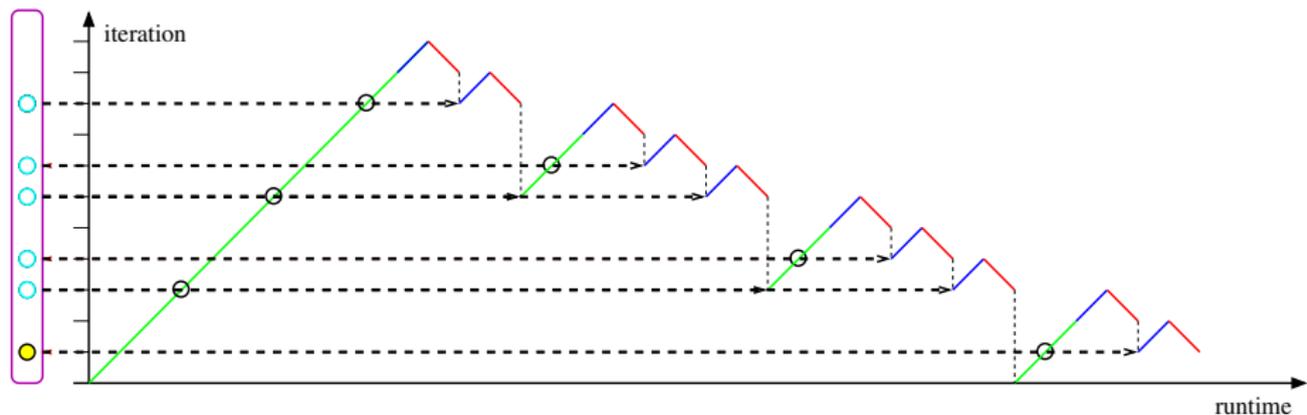
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



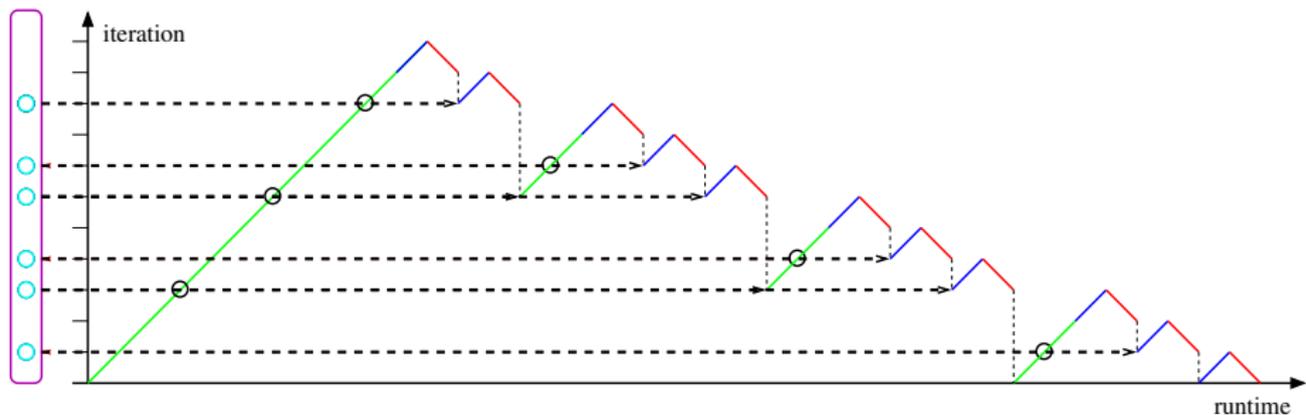
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



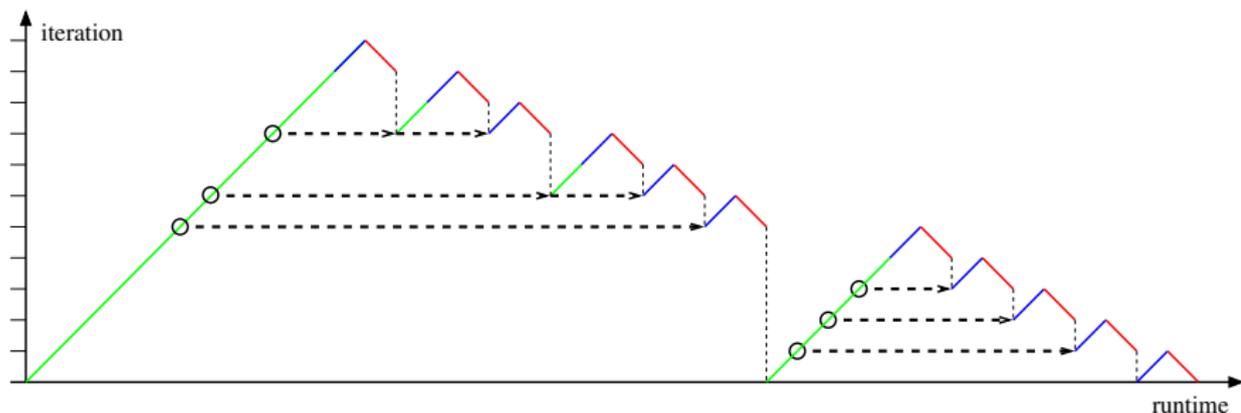
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



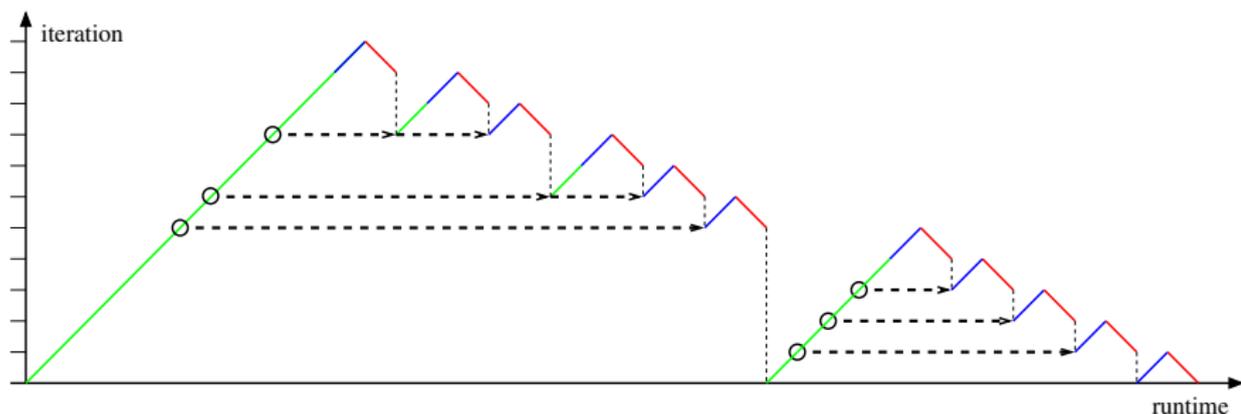
- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

high-level recomputation = checkpointing



- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
- optimal (binomial) scheme encoded in `revolve`; F9X implementation available at <http://mercurial.mcs.anl.gov/ad/RevolveF9X>

high-level recomputation = checkpointing



- 11 iters., memory limited to one iter. of storing J_i & 3 checkpoints
- run forward, store = “tape” the last step, and adjoint
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
- optimal (binomial) scheme encoded in `revolve`; F9X implementation available at <http://mercurial.mcs.anl.gov/ad/RevolveF9X>
- source transformation tool needs to provide four variants

checkpointing usage

- checkpoints on disk* vs. tape in memory

checkpointing usage

- checkpoints on disk* vs. tape in memory
- limited tape memory
→ limited length of computation between checkpoints

checkpointing usage

- checkpoints on disk* vs. tape in memory
- limited tape memory
→ limited length of computation between checkpoints
- limited disk space for checkpoints

checkpointing usage

- checkpoints on disk* vs. tape in memory
- limited tape memory
→ limited length of computation between checkpoints
- limited disk space for checkpoints
- hierarchical scheme → recomputation overhead

checkpointing usage

- checkpoints on disk* vs. tape in memory
- limited tape memory
→ limited length of computation between checkpoints
- limited disk space for checkpoints
- hierarchical scheme → recomputation overhead
- simple time stepping loop with k iterations, worst case overhead factor up to $O(k^2)$

checkpointing usage

- checkpoints on disk* vs. tape in memory
- limited tape memory
→ limited length of computation between checkpoints
- limited disk space for checkpoints
- hierarchical scheme → recomputation overhead
- simple time stepping loop with k iterations, worst case overhead factor up to $O(k^2)$
- in practice up to 3-4 nesting levels (revolve precomputes)
- joint reversal or manually prescribed (at loop)

checkpointing usage

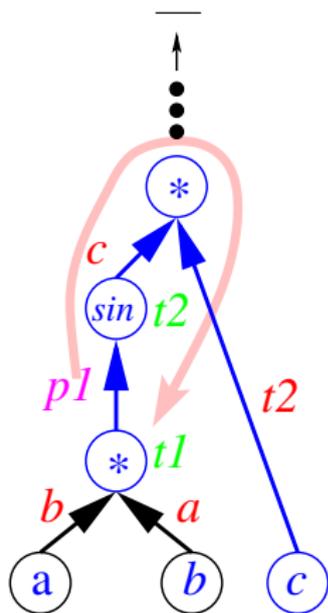
- checkpoints on disk* vs. tape in memory
- limited tape memory
→ limited length of computation between checkpoints
- limited disk space for checkpoints
- hierarchical scheme → recomputation overhead
- simple time stepping loop with k iterations, worst case overhead factor up to $O(k^2)$
- in practice up to 3-4 nesting levels (revolve precomputes)
- joint reversal or manually prescribed (at loop)
- in theory:
 - parallelized recomputation
 - cost of writes

checkpointing usage

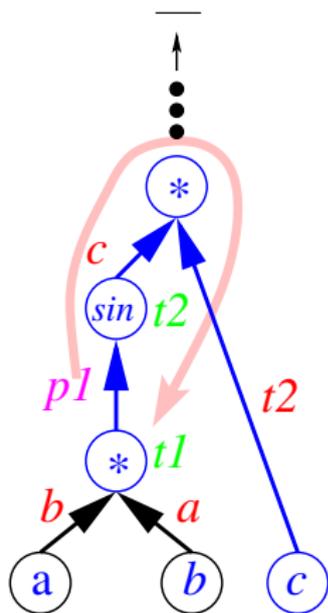
- checkpoints on disk* vs. tape in memory
- limited tape memory
→ limited length of computation between checkpoints
- limited disk space for checkpoints
- hierarchical scheme → recomputation overhead
- simple time stepping loop with k iterations, worst case overhead factor up to $O(k^2)$
- in practice up to 3-4 nesting levels (revolve precomputes)
- joint reversal or manually prescribed (at loop)
- in theory:
 - parallelized recomputation
 - cost of writes
 - nested checkpointing for non- split/joint reversal schemes
 - unified view of checkpointing/taping (data flow eqn.)
 - result checkpointing
 - ...

low-level recomputation - existing approaches

- "recompute all" (taf) - manual store pragmas & automatic slicing

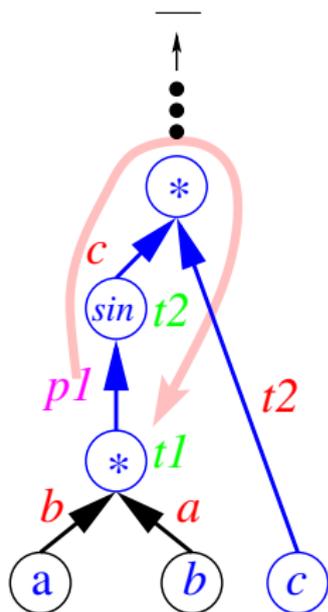


low-level recomputation - existing approaches



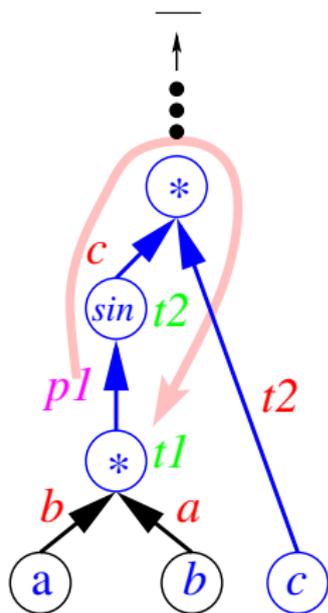
- "recompute all" (taf) - manual store pragmas & automatic slicing
- "store all"

low-level recomputation - existing approaches



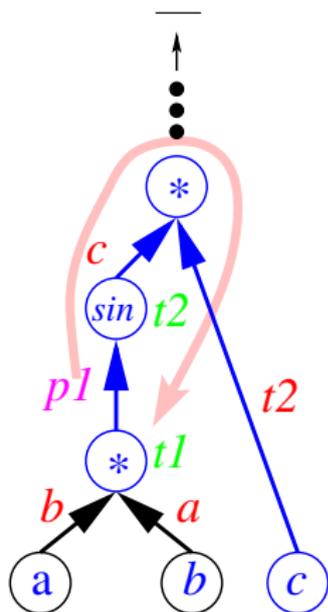
- "recompute all" (taf) - manual store pragmas & automatic slicing
- "store all"
 - values, ops, pseudo-addresses (adol-c)

low-level recomputation - existing approaches



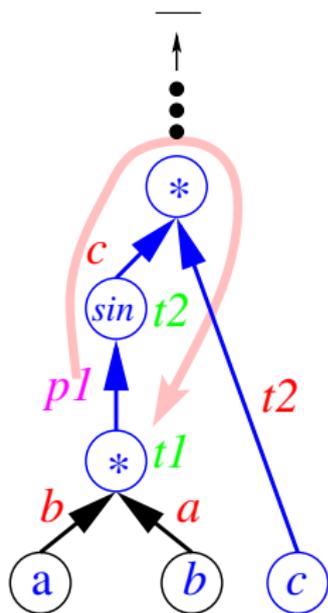
- "recompute all" (taf) - manual store pragmas & automatic slicing
- "store all"
 - values, ops, pseudo-addresses (adol-c)
 - preaccumulated partials (dco,sacado,openad*)

low-level recomputation - existing approaches



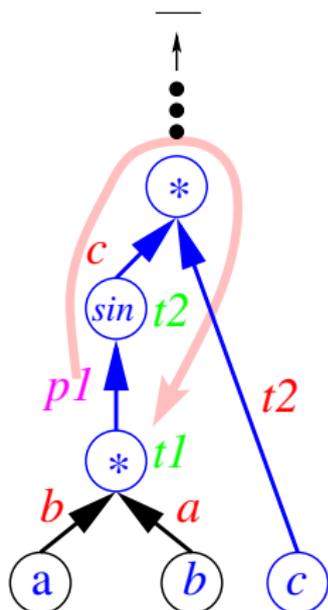
- "recompute all" (taf) - manual store pragmas & automatic slicing
- "store all"
 - values, ops, pseudo-addresses (adol-c)
 - preaccumulated partials (dco,sacado,openad*)
 - store on overwrite program variables referenced in non-linear ops aka TBR (tapenade)

low-level recomputation - existing approaches



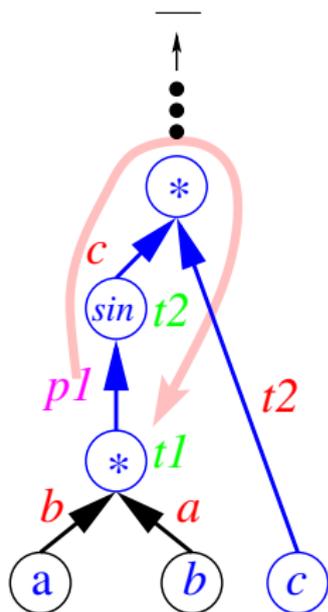
- "recompute all" (taf) - manual store pragmas & automatic slicing
- "store all"
 - values, ops, pseudo-addresses (adol-c)
 - preaccumulated partials (dco,sacado,openad*)
 - store on overwrite program variables referenced in non-linear ops aka TBR (tapenade)
 - ad-hoc for address computations (data & instructions), i.e. pointers, indices, control flow

low-level recomputation - existing approaches



- "recompute all" (taf) - manual store pragmas & automatic slicing
- "store all"
 - values, ops, pseudo-addresses (adol-c)
 - preaccumulated partials (dco,sacado,openad*)
 - store on overwrite program variables referenced in non-linear ops aka TBR (tapenade)
 - ad-hoc for address computations (data & instructions), i.e. pointers, indices, control flow
- TBR rationale: a value is overwritten only once* but used at least once.

low-level recomputation - existing approaches



- "recompute all" (taf) - manual store pragmas & automatic slicing
- "store all"
 - values, ops, pseudo-addresses (adol-c)
 - preaccumulated partials (dco,sacado,openad*)
 - store on overwrite program variables referenced in non-linear ops aka TBR (tapenade)
 - ad-hoc for address computations (data & instructions), i.e. pointers, indices, control flow
- TBR rationale: a value is overwritten only once* but used at least once.

What is the problem then?

observations regarding existing approaches

- existing approaches are fixed schemes based on data flow analysis (i.e. not computational graphs)

observations regarding existing approaches

- existing approaches are fixed schemes based on data flow analysis (i.e. not computational graphs)
- is there a data flow formulation for the store vs. recompute tradeoff?

observations regarding existing approaches

- existing approaches are fixed schemes based on data flow analysis (i.e. not computational graphs)
- is there a data flow formulation for the store vs. recompute tradeoff?
- prefer graph representation (already used for cross-country eliminations, scarcity)

observations regarding existing approaches

- existing approaches are fixed schemes based on data flow analysis (i.e. not computational graphs)
- is there a data flow formulation for the store vs. recompute tradeoff?
- prefer graph representation (already used for cross-country eliminations, scarcity)
- graph based approach plausible for source transformation (limited scope, globally valid),

observations regarding existing approaches

- existing approaches are fixed schemes based on data flow analysis (i.e. not computational graphs)
- is there a data flow formulation for the store vs. recompute tradeoff?
- prefer graph representation (already used for cross-country eliminations, scarcity)
- graph based approach plausible for source transformation (limited scope, globally valid),
- less so for operator overloading (complete scope, locally valid)

observations regarding existing approaches

- existing approaches are fixed schemes based on data flow analysis (i.e. not computational graphs)
- is there a data flow formulation for the store vs. recompute tradeoff?
- prefer graph representation (already used for cross-country eliminations, scarcity)
- graph based approach plausible for source transformation (limited scope, globally valid),
- less so for operator overloading (complete scope, locally valid)
- significant implementation effort & strong dependencies on analysis for graph representations in source transformation

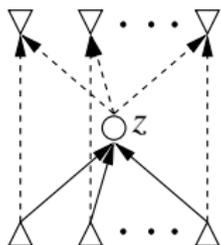
observations regarding existing approaches

- existing approaches are fixed schemes based on data flow analysis (i.e. not computational graphs)
- is there a data flow formulation for the store vs. recompute tradeoff?
- prefer graph representation (already used for cross-country eliminations, scarcity)
- graph based approach plausible for source transformation (limited scope, globally valid),
- less so for operator overloading (complete scope, locally valid)
- significant implementation effort & strong dependencies on analysis for graph representations in source transformation

Is the graph representation worth the effort?

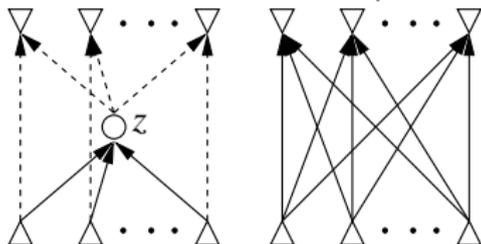
the case for using graphs

- scarcity preserving elim./rerouting, e.g. $f(\mathbf{x}) = (\mathbf{D} + \mathbf{a}\mathbf{x}^T)\mathbf{x}$ with intermediate $z = \mathbf{x}^T\mathbf{x}$ (constant edges dashed)



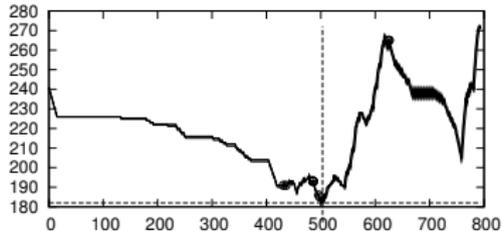
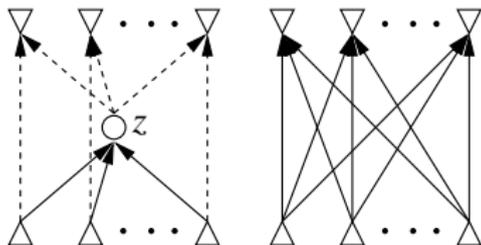
the case for using graphs

- scarcity preserving elim./rerouting, e.g. $f(\mathbf{x}) = (\mathbf{D} + \mathbf{a}\mathbf{x}^T)\mathbf{x}$ with intermediate $z = \mathbf{x}^T\mathbf{x}$ (constant edges dashed)



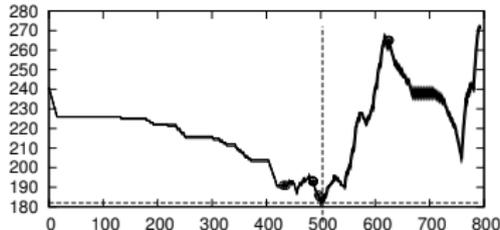
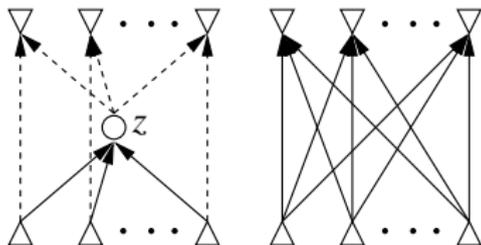
the case for using graphs

- scarcity preserving elim./rerouting, e.g. $f(x) = (D + ax^T)x$ with intermediate $z = x^T x$ (constant edges dashed)



the case for using graphs

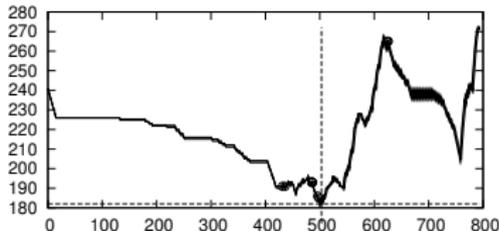
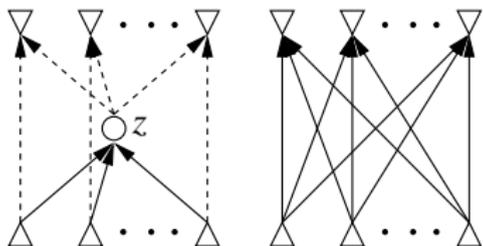
- scarcity preserving elim./rerouting, e.g. $f(\mathbf{x}) = (\mathbf{D} + \mathbf{a}\mathbf{x}^T)\mathbf{x}$ with intermediate $z = \mathbf{x}^T\mathbf{x}$ (constant edges dashed)



- inverse (re)computations of loop variables during reverse sweep/combined with forward computations (loop body)

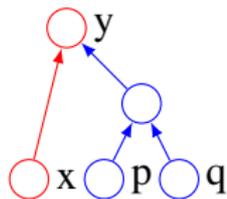
the case for using graphs

- scarcity preserving elim./rerouting, e.g. $f(x) = (D + ax^T)x$ with intermediate $z = x^T x$ (constant edges dashed)



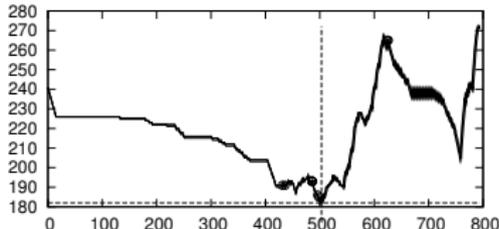
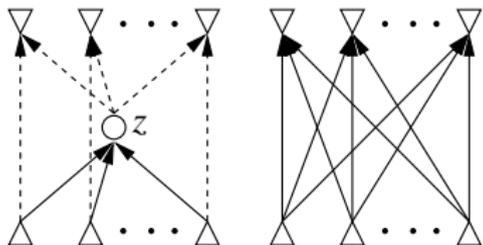
- inverse (re)computations of loop variables during reverse sweep/combined with forward computations (loop body)
- TBR rationale & simple observations on storage decision:

$$y = x * p * q$$



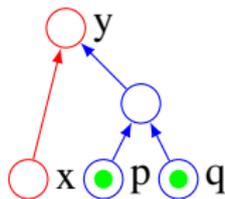
the case for using graphs

- scarcity preserving elim./rerouting, e.g. $f(x) = (D + ax^T)x$ with intermediate $z = x^T x$ (constant edges dashed)



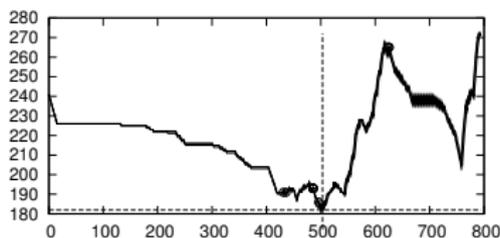
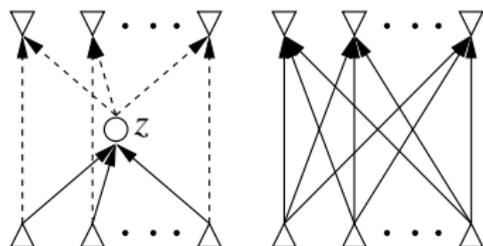
- inverse (re)computations of loop variables during reverse sweep/combined with forward computations (loop body)
- TBR rationale & simple observations on storage decision:

$y = x * p * q$
store p, q or $p * q$?



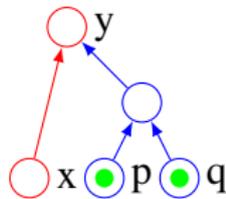
the case for using graphs

- scarcity preserving elim./rerouting, e.g. $f(x) = (D + ax^T)x$ with intermediate $z = x^T x$ (constant edges dashed)



- inverse (re)computations of loop variables during reverse sweep/combined with forward computations (loop body)
- TBR rationale & simple observations on storage decision:

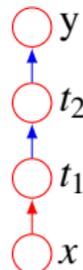
$y = x * p * q$
store p, q or $p * q$?



$$t_1 = \phi_1(x);$$

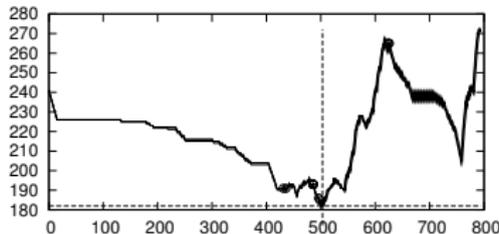
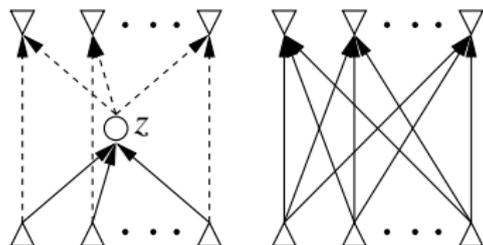
$$t_2 = \phi_2(t_1);$$

$$y = \phi_3(t_2);$$



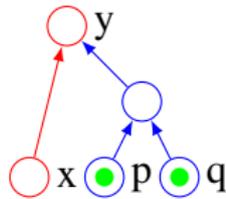
the case for using graphs

- scarcity preserving elim./rerouting, e.g. $f(x) = (D + ax^T)x$ with intermediate $z = x^T x$ (constant edges dashed)

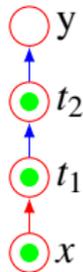


- inverse (re)computations of loop variables during reverse sweep/combined with forward computations (loop body)
- TBR rationale & simple observations on storage decision:

$y = x * p * q$
store p, q or $p * q$?

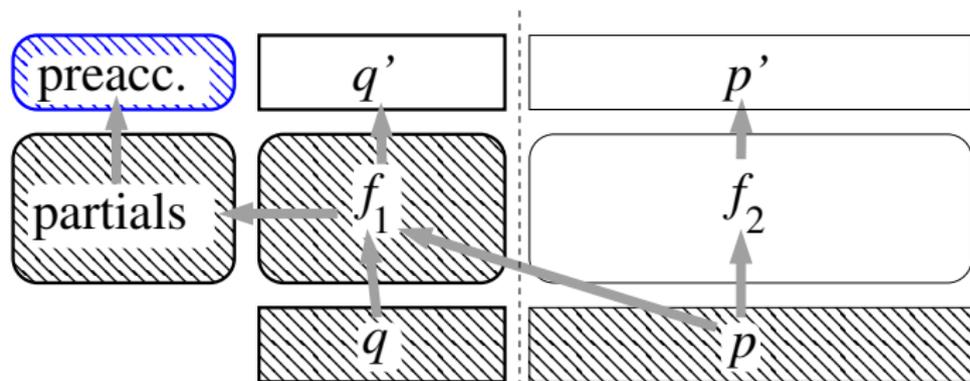


$t_1 = \phi_1(x);$
 $t_2 = \phi_2(t_1);$
 $y = \phi_3(t_2);$
instead store just x
and recompute?



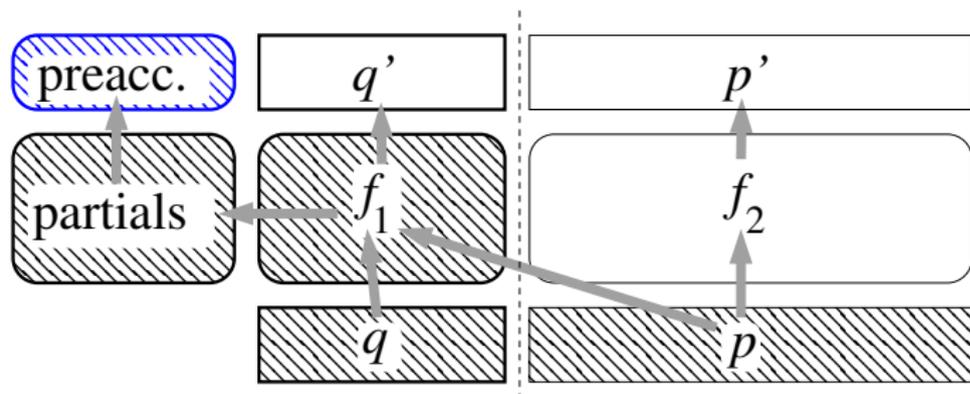
macroscopic view

- use case for preferring preaccumulation store over store (on overwrite) of (input) values
- state (p, q) , p is some passive forcing
- time stepping loop



macroscopic view

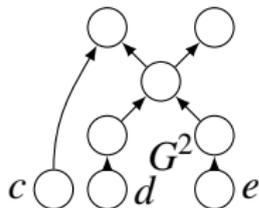
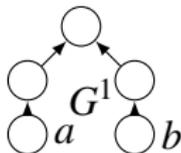
- use case for preferring preaccumulation store over store (on overwrite) of (input) values
- state (p, q) , p is some passive forcing
- time stepping loop



BUT ,, preaccumulated partials are never shared!

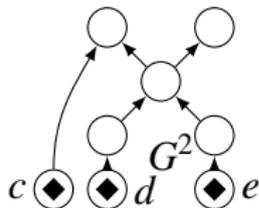
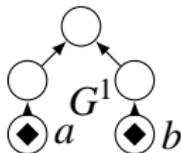
Multiple DAG formulation to capture shared use

- consider a set $\{G^1, G^2, \dots\}$ of DAGs, i.e. sequences of statements in a given scope



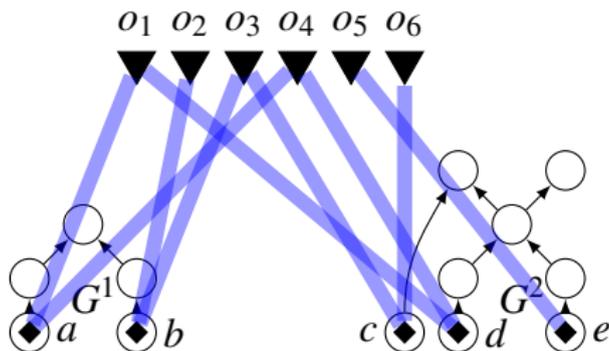
Multiple DAG formulation to capture shared use

- consider a set $\{G^1, G^2, \dots\}$ of DAGs, i.e. sequences of statements in a given scope
- minimal vertices = program variables ($a \dots e$)



Multiple DAG formulation to capture shared use

- consider a set $\{G^1, G^2, \dots\}$ of DAGs, i.e. sequences of statements in a given scope
- minimal vertices = program variables ($a \dots e$)
- values held in $a \dots e$ are overwritten by instructions $o_1 \dots o_6$
- overlay use-overwrite graph



reasons for ambiguous overwrites

overwrite instructions in *static* analysis

- aliasing
 - *p vs *q

reasons for ambiguous overwrites

overwrite instructions in *static* analysis

- aliasing
 - `*p` vs `*q`
 - `a[i]` vs `a[j]`
 - ...

reasons for ambiguous overwrites

overwrite instructions in *static* analysis

- aliasing
 - $*p$ vs $*q$
 - $a[i]$ vs $a[j]$
 - ...
- unproven full array overwrites

reasons for ambiguous overwrites

overwrite instructions in *static* analysis

- aliasing
 - $*p$ vs $*q$
 - $a[i]$ vs $a[j]$
 - ...
- unproven full array overwrites
- control flow branches

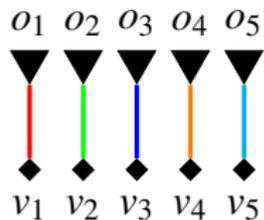
reasons for ambiguous overwrites

overwrite instructions in *static* analysis

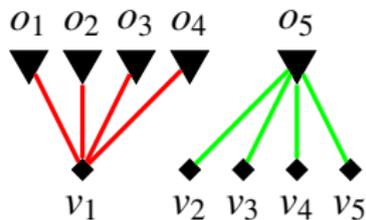
- aliasing
 - $*p$ vs $*q$
 - $a[i]$ vs $a[j]$
 - ...
- unproven full array overwrites
- control flow branches

multiple overwrites are paired with multiple uses

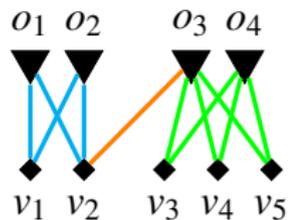
rationale for looking at bicliques



(a)

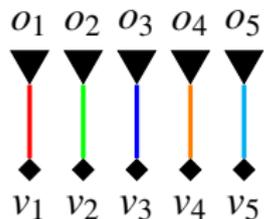


(b)

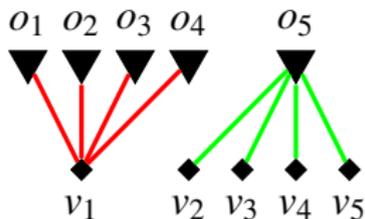


(c)

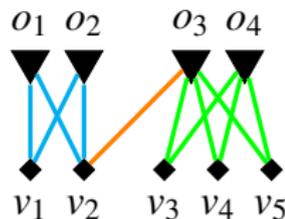
rationale for looking at bicliques



(a)



(b)

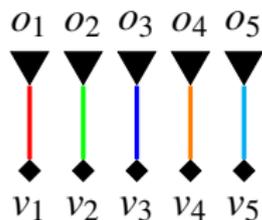


(c)

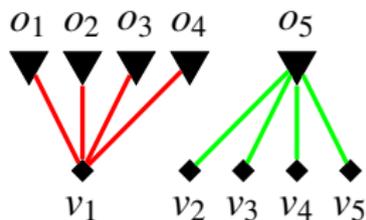
- (static) cost estimate via node count ratios

- (a) inconclusive (tie breakers: control flow, memory reference,...)
- (b) v_1 store on use (or temporary), $v_2 \dots v_5$ store on overwrite
- (c) $v_3 \dots v_5$ store on overwrite

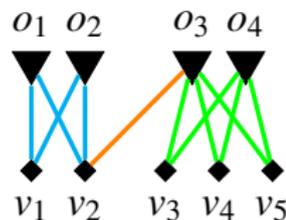
rationale for looking at bicliques



(a)



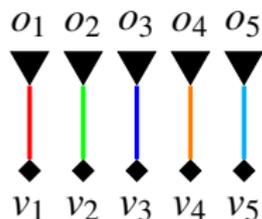
(b)



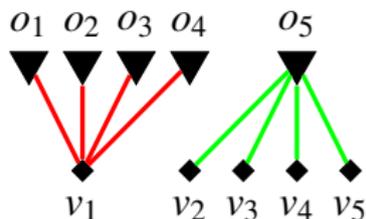
(c)

- (static) cost estimate via node count ratios
 - (a) inconclusive (tie breakers: control flow, memory reference,...)
 - (b) v_1 store on use (or temporary), $v_2 \dots v_5$ store on overwrite
 - (c) $v_3 \dots v_5$ store on overwrite
- bicliques may share nodes \rightarrow need adjustments

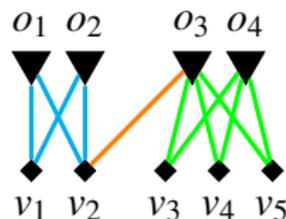
rationale for looking at bicliques



(a)



(b)



(c)

- (static) cost estimate via node count ratios
 - (a) inconclusive (tie breakers: control flow, memory reference,...)
 - (b) v_1 store on use (or temporary), $v_2 \dots v_5$ store on overwrite
 - (c) $v_3 \dots v_5$ store on overwrite
- bicliques may share nodes \rightarrow need adjustments
- need condition ensuring correct recovery of “required” values.

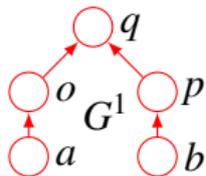
required values and their correct recovery

example G^1

```
o=sin(a);
```

```
p=cos(b);
```

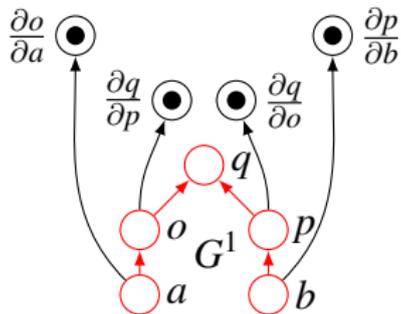
```
q=o*p;
```



required values and their correct recovery

example G^1 , modify to “combined” DAG

```
o=sin(a);  
p=cos(b);  
q=o*p;
```

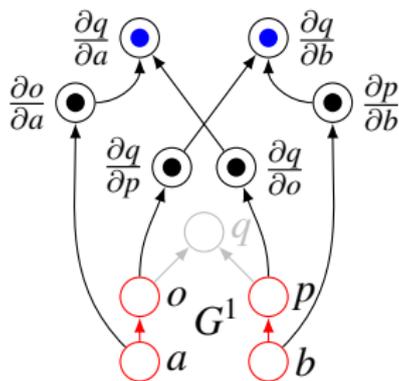


$$\left. \begin{aligned} \frac{\partial o}{\partial a} &= \cos(a); \\ \frac{\partial p}{\partial b} &= -\sin(b); \\ \frac{\partial q}{\partial p} &= o; \\ \frac{\partial q}{\partial o} &= p; \end{aligned} \right\} \text{linearization}$$

required values and their correct recovery

example G^1 , modify to “combined” DAG

```
o=sin(a);  
p=cos(b);  
q=o*p;
```

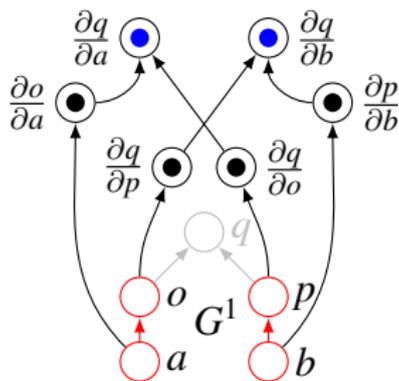


$$\left. \begin{aligned} \frac{\partial o}{\partial a} &= \cos(a); \\ \frac{\partial p}{\partial b} &= -\sin(b); \\ \frac{\partial q}{\partial p} &= o; \\ \frac{\partial q}{\partial o} &= p; \end{aligned} \right\} \text{linearization}$$
$$\left. \begin{aligned} \frac{\partial q}{\partial a} &= \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\ \frac{\partial q}{\partial b} &= \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b}; \end{aligned} \right\} \text{preaccumulation}$$

required values and their correct recovery

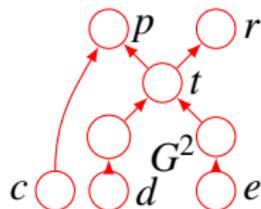
example G^1 , modify to “combined” DAG, same for G^2

```
o=sin(a);  
p=cos(b);  
q=o*p;
```



$$\left. \begin{array}{l} \frac{\partial o}{\partial a} = \cos(a); \\ \frac{\partial p}{\partial b} = -\sin(b); \\ \frac{\partial q}{\partial p} = o; \\ \frac{\partial q}{\partial o} = p; \end{array} \right\} \text{linearization}$$
$$\left. \begin{array}{l} \frac{\partial q}{\partial a} = \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\ \frac{\partial q}{\partial b} = \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b}; \end{array} \right\} \text{preaccumulation}$$

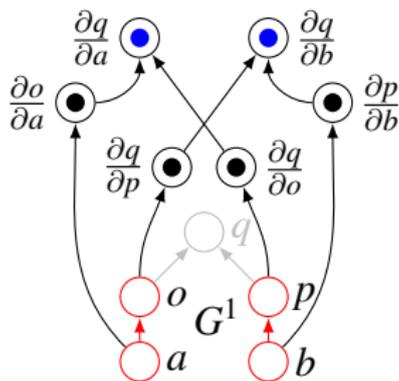
```
t=5*d+4*e;  
p=sin(c)+t;  
r=cos(t);
```



required values and their correct recovery

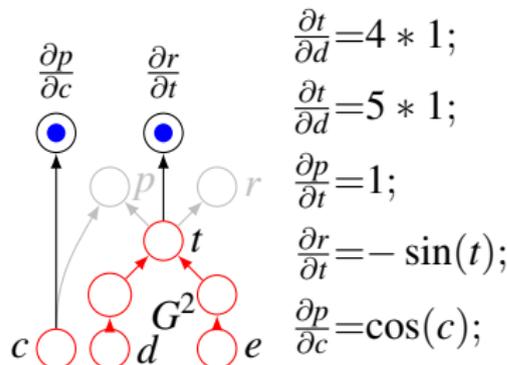
example G^1 , modify to “combined” DAG, same for G^2

$o = \sin(a)$;
 $p = \cos(b)$;
 $q = o * p$;



$$\left. \begin{aligned}
 \frac{\partial o}{\partial a} &= \cos(a); \\
 \frac{\partial p}{\partial b} &= -\sin(b); \\
 \frac{\partial q}{\partial p} &= o; \\
 \frac{\partial q}{\partial o} &= p; \\
 \frac{\partial q}{\partial a} &= \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\
 \frac{\partial q}{\partial b} &= \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b};
 \end{aligned} \right\} \begin{array}{l} \text{linearization} \\ \\ \\ \\ \text{preaccumulation} \end{array}$$

$t = 5 * d + 4 * e$;
 $p = \sin(c) + t$;
 $r = \cos(t)$;
 $R = \left\{ \frac{\partial p}{\partial c}, \frac{\partial r}{\partial t} \right\}$

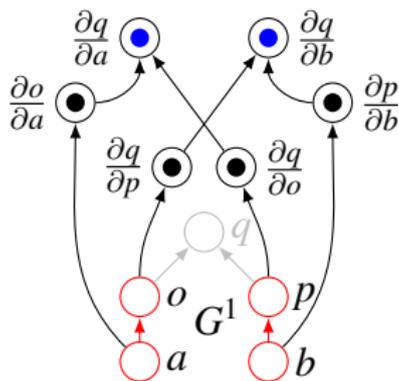


$$\begin{aligned}
 \frac{\partial t}{\partial d} &= 4 * 1; \\
 \frac{\partial t}{\partial e} &= 5 * 1; \\
 \frac{\partial p}{\partial t} &= 1; \\
 \frac{\partial r}{\partial t} &= -\sin(t); \\
 \frac{\partial p}{\partial c} &= \cos(c);
 \end{aligned}$$

required values and their correct recovery

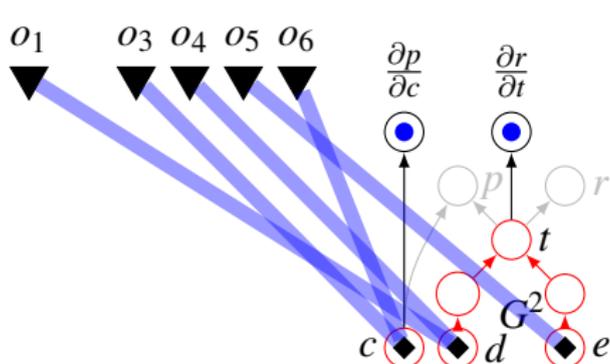
example G^1 , modify to “combined” DAG, same for G^2

$o = \sin(a)$;
 $p = \cos(b)$;
 $q = o * p$;



$$\left. \begin{aligned}
 \frac{\partial o}{\partial a} &= \cos(a); \\
 \frac{\partial p}{\partial b} &= -\sin(b); \\
 \frac{\partial q}{\partial p} &= o; \\
 \frac{\partial q}{\partial o} &= p; \\
 \frac{\partial q}{\partial a} &= \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\
 \frac{\partial q}{\partial b} &= \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b};
 \end{aligned} \right\} \begin{array}{l} \text{linearization} \\ \text{preaccumulation} \end{array}$$

$t = 5 * d + 4 * e$;
 $p = \sin(c) + t$;
 $r = \cos(t)$;
 $R = \left\{ \frac{\partial p}{\partial c}, \frac{\partial r}{\partial t} \right\}$

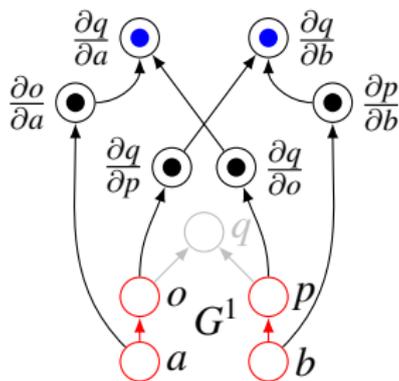


$$\begin{aligned}
 \frac{\partial t}{\partial d} &= 4 * 1; \\
 \frac{\partial t}{\partial e} &= 5 * 1; \\
 \frac{\partial p}{\partial t} &= 1; \\
 \frac{\partial r}{\partial t} &= -\sin(t); \\
 \frac{\partial p}{\partial c} &= \cos(c);
 \end{aligned}$$

required values and their correct recovery

example G^1 , modify to “combined” DAG, same for G^2

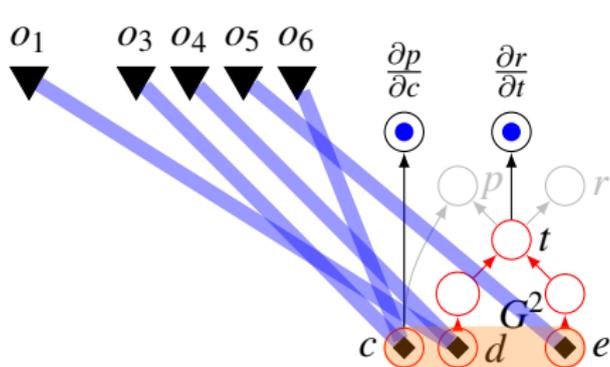
$o = \sin(a)$;
 $p = \cos(b)$;
 $q = o * p$;



$$\left. \begin{array}{l}
 \frac{\partial o}{\partial a} = \cos(a); \\
 \frac{\partial p}{\partial b} = -\sin(b); \\
 \frac{\partial q}{\partial p} = o; \\
 \frac{\partial q}{\partial o} = p; \\
 \frac{\partial q}{\partial a} = \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\
 \frac{\partial q}{\partial b} = \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b};
 \end{array} \right\} \text{linearization}$$

$$\left. \begin{array}{l}
 \frac{\partial q}{\partial p} = o; \\
 \frac{\partial q}{\partial o} = p; \\
 \frac{\partial q}{\partial a} = \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\
 \frac{\partial q}{\partial b} = \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b};
 \end{array} \right\} \text{preaccumulation}$$

$t = 5 * d + 4 * e$;
 $p = \sin(c) + t$;
 $r = \cos(t)$;
 $R = \left\{ \frac{\partial p}{\partial c}, \frac{\partial r}{\partial t} \right\}$
 $S = \{o_1, o_3, \dots, o_6\}$
 $U = \emptyset$

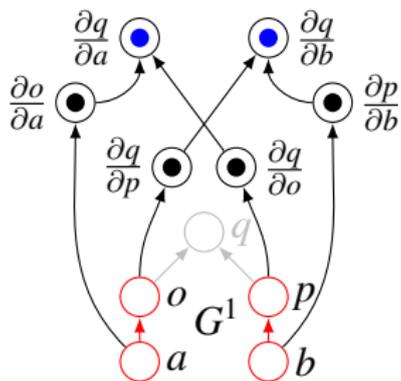


$$\frac{\partial t}{\partial d} = 4 * 1; \\
 \frac{\partial t}{\partial e} = 5 * 1; \\
 \frac{\partial p}{\partial t} = 1; \\
 \frac{\partial r}{\partial t} = -\sin(t); \\
 \frac{\partial p}{\partial c} = \cos(c);$$

required values and their correct recovery

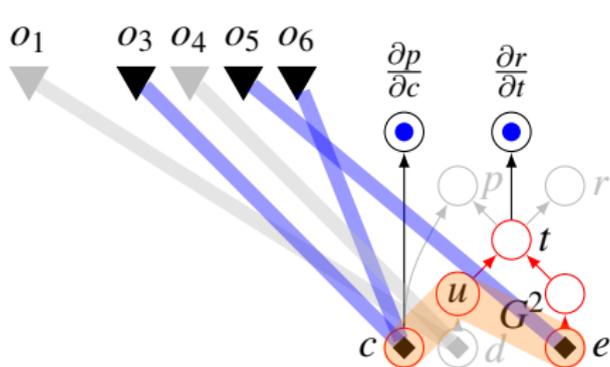
example G^1 , modify to “combined” DAG, same for G^2

$o = \sin(a)$;
 $p = \cos(b)$;
 $q = o * p$;



$$\left. \begin{aligned}
 \frac{\partial o}{\partial a} &= \cos(a); \\
 \frac{\partial p}{\partial b} &= -\sin(b); \\
 \frac{\partial q}{\partial p} &= o; \\
 \frac{\partial q}{\partial o} &= p; \\
 \frac{\partial q}{\partial a} &= \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\
 \frac{\partial q}{\partial b} &= \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b};
 \end{aligned} \right\} \begin{array}{l} \text{linearization} \\ \text{preaccumulation} \end{array}$$

$t = 5 * d + 4 * e$;
 $p = \sin(c) + t$;
 $r = \cos(t)$;
 $R = \left\{ \frac{\partial p}{\partial c}, \frac{\partial r}{\partial t} \right\}$
 $S = \{o_3, o_5, o_6\}$
 $U = \{u\}$

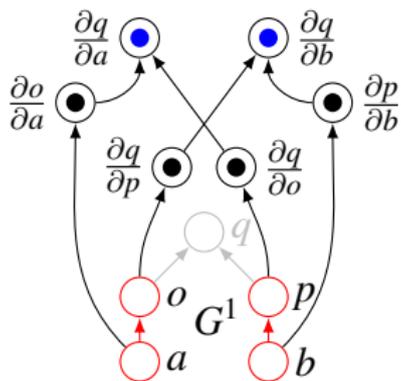


$$\begin{aligned}
 \frac{\partial t}{\partial d} &= 4 * 1; \\
 \frac{\partial t}{\partial e} &= 5 * 1; \\
 \frac{\partial p}{\partial t} &= 1; \\
 \frac{\partial r}{\partial t} &= -\sin(t); \\
 \frac{\partial p}{\partial c} &= \cos(c);
 \end{aligned}$$

required values and their correct recovery

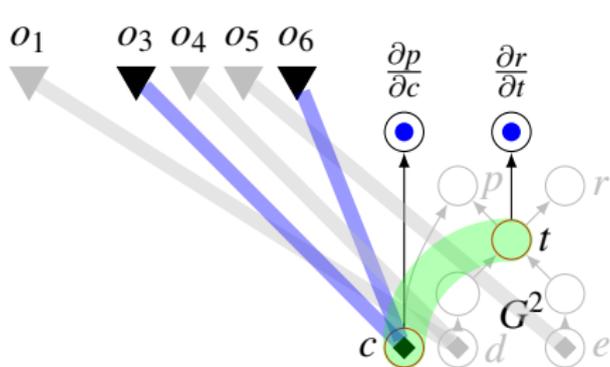
example G^1 , modify to “combined” DAG, same for G^2

$o = \sin(a)$;
 $p = \cos(b)$;
 $q = o * p$;



$$\left. \begin{aligned}
 \frac{\partial o}{\partial a} &= \cos(a); \\
 \frac{\partial p}{\partial b} &= -\sin(b); \\
 \frac{\partial q}{\partial p} &= o; \\
 \frac{\partial q}{\partial o} &= p; \\
 \frac{\partial q}{\partial a} &= \frac{\partial q}{\partial o} * \frac{\partial o}{\partial a}; \\
 \frac{\partial q}{\partial b} &= \frac{\partial q}{\partial p} * \frac{\partial p}{\partial b};
 \end{aligned} \right\} \begin{array}{l} \text{linearization} \\ \text{preaccumulation} \end{array}$$

$t = 5 * d + 4 * e$;
 $p = \sin(c) + t$;
 $r = \cos(t)$;
 $R = \left\{ \frac{\partial p}{\partial c}, \frac{\partial r}{\partial t} \right\}$
 $S = \{o_3, o_6\}$
 $U = \{t\}$



$$\begin{aligned}
 \frac{\partial t}{\partial d} &= 4 * 1; \\
 \frac{\partial t}{\partial e} &= 5 * 1; \\
 \frac{\partial p}{\partial t} &= 1; \\
 \frac{\partial r}{\partial t} &= -\sin(t); \\
 \frac{\partial p}{\partial c} &= \cos(c);
 \end{aligned}$$

recomputation criterion for a set of DAGs

given: $\{G^{*1}, \dots, G^{*k}\}$ with $G^{*i} = (V^{*i}, E^{*i})$ and $\{R^1, \dots, R^k\}$, $S = \bigcup S^i$,
 $U = \bigcup U^i$, and the bipartite use-overwrite graph $G_b = ((\bigcup V_{min}^{*i}) \cup O, E_b)$;
then $S \subseteq O$ and $U \subseteq \bigcup V^{*i}$ are **sufficient to recompute** all vertices in the R^i if
for each combined G^{*i} there is a vertex cut C^i with respect to R^i such that
 $\forall c_j \in C^i : (c_j \in U) \vee ((c_j \in V_{min}^{*i}) \wedge ((c_j, o) \in O \Rightarrow o \in S))$

recomputation criterion for a set of DAGs

given: $\{G^{*1}, \dots, G^{*k}\}$ with $G^{*i} = (V^{*i}, E^{*i})$ and $\{R^1, \dots, R^k\}$, $S = \bigcup S^i$, $U = \bigcup U^i$, and the bipartite use-overwrite graph $G_b = ((\bigcup V_{min}^{*i}) \cup O, E_b)$;
then $S \subseteq O$ and $U \subseteq \bigcup V^{*i}$ are **sufficient to recompute** all vertices in the R^i if for each combined G^{*i} there is a vertex cut C^i with respect to R^i such that $\forall c_j \in C^i : (c_j \in U) \vee ((c_j \in V_{min}^{*i}) \wedge ((c_j, o) \in O \Rightarrow o \in S))$

cost estimate (because it is static): $|S| + |U|$

recomputation criterion for a set of DAGs

given: $\{G^{*1}, \dots, G^{*k}\}$ with $G^{*i} = (V^{*i}, E^{*i})$ and $\{R^1, \dots, R^k\}$, $S = \bigcup S^i$, $U = \bigcup U^i$, and the bipartite use-overwrite graph $G_b = ((\bigcup V_{min}^{*i}) \cup O, E_b)$;
then $S \subseteq O$ and $U \subseteq \bigcup V^{*i}$ are **sufficient to recompute** all vertices in the R^i if for each combined G^{*i} there is a vertex cut C^i with respect to R^i such that $\forall c_j \in C^i : (c_j \in U) \vee ((c_j \in V_{min}^{*i}) \wedge ((c_j, o) \in O \Rightarrow o \in S))$

cost estimate (because it is static): $|S| + |U|$

approach: change S , then use criterion to update U to be sufficient:

Given single comb. DAG $G^* = (V_{min}^* \cup V_{inter}^* \cup V_{max}^*, E)$, R, S, V_S, U

- 01 $U := U \setminus V$; $C := V_S \cap V_{min}$
- 02 form the subgraph $G^{*'}$ induced by all paths from $V_{min} \setminus C$ to R
- 03 determine a minimal vertex cut C' in $G^{*'}$ using as tie breaker the minimal distance from C' to R .
- 04 set $C := C \cup C'$ as the vertex cut for G^* and set $U := U \cup C'$.

How to change S

- compute minimal biclique cover of G_b

How to change S

- compute minimal biclique cover of G_b
- for each biclique $B = (V_B, O_B)$, evaluate

$$r_B^- = \frac{|O_B^-|}{|v_B|}; \quad r_B^+ = \frac{|v_B|}{|O_B^+|}$$

where $O_B^- = O_B \cap S$ and $O_B^+ = O_B \setminus S$.

- they are the indicators for removing from and adding to S

How to change S

- compute minimal biclique cover of G_b
- for each biclique $B = (V_B, O_B)$, evaluate

$$r_B^- = \frac{|O_B^-|}{|v_B|}; \quad r_B^+ = \frac{|v_B|}{|O_B^+|}$$

where $O_B^- = O_B \cap S$ and $O_B^+ = O_B \setminus S$.

- they are the indicators for removing from and adding to S
- the bicliques with the largest node ratios are expected to have the biggest impact on the cost

How to change S

- compute minimal biclique cover of G_b
- for each biclique $B = (V_B, O_B)$, evaluate

$$r_B^- = \frac{|O_B^-|}{|v_B|}; \quad r_B^+ = \frac{|v_B|}{|O_B^+|}$$

where $O_B^- = O_B \cap S$ and $O_B^+ = O_B \setminus S$.

- they are the indicators for removing from and adding to S
- the bicliques with the largest node ratios are expected to have the biggest impact on the cost

•

$$S := \begin{cases} S \cup O_B^+ & \text{if maximal ratio is } r_B^+ \\ S \setminus O_B^- & \text{if maximal ratio is } r_B^- \end{cases}$$

How to change S

- compute minimal biclique cover of G_b
- for each biclique $B = (V_B, O_B)$, evaluate

$$r_B^- = \frac{|O_B^-|}{|v_B|}; \quad r_B^+ = \frac{|v_B|}{|O_B^+|}$$

where $O_B^- = O_B \cap S$ and $O_B^+ = O_B \setminus S$.

- they are the indicators for removing from and adding to S
- the bicliques with the largest node ratios are expected to have the biggest impact on the cost

•

$$S := \begin{cases} S \cup O_B^+ & \text{if maximal ratio is } r_B^+ \\ S \setminus O_B^- & \text{if maximal ratio is } r_B^- \end{cases}$$

- need to adjust overlapping bicliques

as an algorithm

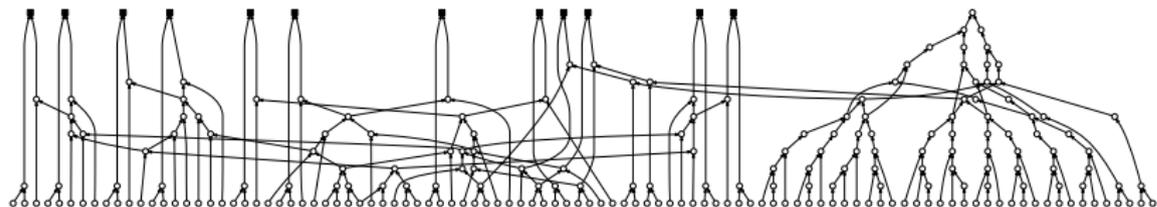
Given $\delta \in [0, 1]$, $R = \bigcup R^i$, $G_{R^i}^{*i}$ for all $G^i \in \mathcal{G}$ and $G_b = (V_b, O, E_b)$;

- 01 if $|O| < |R|$ then $(S, U) := (O, \emptyset)$; $c := |O|$
- 02 else $(S, U) := (\emptyset, R)$; $c := |R|$
- 03 compute minimal biclique cover \mathcal{C} for G_b
- 04 $\forall B = (V_B, O_B) \in \mathcal{C}$ set $O_B := O_B \cup \{o : ((v, o) \in E_b \wedge v \in V_B)\}$
- 05 while $\mathcal{C} \neq \emptyset$
- 06 $\forall B \in \mathcal{C}$ compute ratios r_B^- and r_B^+ and sort
- 07 if maximal ratio is less than $1 - \delta$ exit with current (S, U)
- 08 update S as on previous slide
- 09 $\forall G_{R^i}^{*i}$ update U using criterion
- 10 if $c \geq |S| + |U|$ then set $c := |S| + |U|$
- 11 else reset S to the value it had before line 07
- 12 set $\mathcal{C} := \mathcal{C} \setminus \{B\}$

terminates (05,12); greedy heuristic; in the static estimate equal to or better than TBR (01), and store on use (02)

implementation

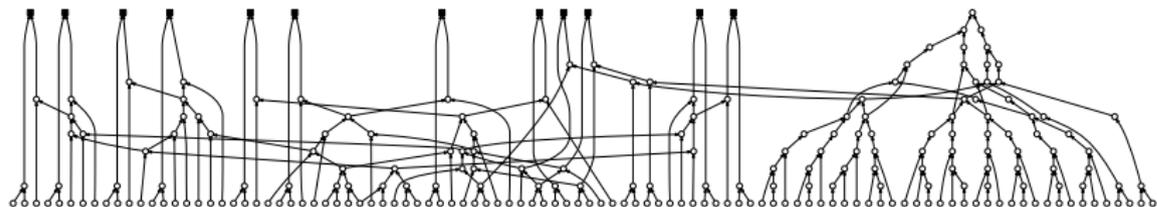
example combined graph in OpenAD:



- started by Andrew Lyons, continued by Heather Cole-Mullen
- outcome depends heavily on (whole program) code analysis (alias, reaching definitions, ...)

implementation

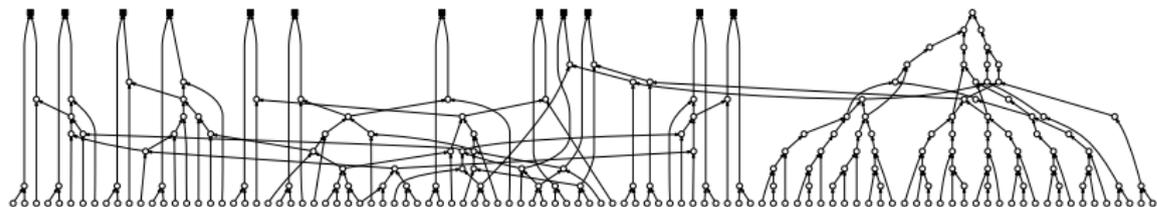
example combined graph in OpenAD:



- started by Andrew Lyons, continued by Heather Cole-Mullen
- outcome depends heavily on (whole program) code analysis (alias, reaching definitions, ...)
- source transformation context prohibits lowering to simpler compiler representations

implementation

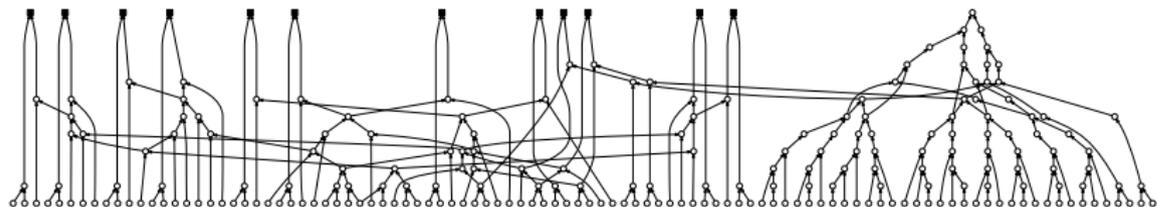
example combined graph in OpenAD:



- started by Andrew Lyons, continued by Heather Cole-Mullen
- outcome depends heavily on (whole program) code analysis (alias, reaching definitions, ...)
- source transformation context prohibits lowering to simpler compiler representations
- once S is determined, need a second round* for address variables occurring in memory references in S and V_S^i , i.e. i for $a[i]$ or p for $*p$

implementation

example combined graph in OpenAD:



- started by Andrew Lyons, continued by Heather Cole-Mullen
- outcome depends heavily on (whole program) code analysis (alias, reaching definitions, ...)
- source transformation context prohibits lowering to simpler compiler representations
- once S is determined, need a second round* for address variables occurring in memory references in S and V_S^i , i.e. i for $a[i]$ or p for $*p$
- ... work in progress

summary & observations

- algorithm complements earlier alg. on inverse loop computations
- first step towards automatic recomputation tradeoff
- addresses TBR and store-on-use shortcomings

summary & observations

- algorithm complements earlier alg. on inverse loop computations
- first step towards automatic recomputation tradeoff
- addresses TBR and store-on-use shortcomings
- correctness does not depend on the scope of the DAG set
(but efficacy & complexity of the algorithm does)

summary & observations

- algorithm complements earlier alg. on inverse loop computations
- first step towards automatic recomputation tradeoff
- addresses TBR and store-on-use shortcomings
- correctness does not depend on the scope of the DAG set
(but efficacy & complexity of the algorithm does)
- needed for hardware with small memory or expensive memory access

summary & observations

- algorithm complements earlier alg. on inverse loop computations
- first step towards automatic recomputation tradeoff
- addresses TBR and store-on-use shortcomings
- correctness does not depend on the scope of the DAG set
(but efficacy & complexity of the algorithm does)
- needed for hardware with small memory or expensive memory access
- starting point for future improvements:
 - consider user directives

summary & observations

- algorithm complements earlier alg. on inverse loop computations
- first step towards automatic recomputation tradeoff
- addresses TBR and store-on-use shortcomings
- correctness does not depend on the scope of the DAG set
(but efficacy & complexity of the algorithm does)
- needed for hardware with small memory or expensive memory access
- starting point for future improvements:
 - consider user directives
 - consider static cost estimates for subroutine calls

summary & observations

- algorithm complements earlier alg. on inverse loop computations
- first step towards automatic recomputation tradeoff
- addresses TBR and store-on-use shortcomings
- correctness does not depend on the scope of the DAG set (but efficacy & complexity of the algorithm does)
- needed for hardware with small memory or expensive memory access
- starting point for future improvements:
 - consider user directives
 - consider static cost estimates for subroutine calls
 - consider other search heuristics

summary & observations

- algorithm complements earlier alg. on inverse loop computations
- first step towards automatic recomputation tradeoff
- addresses TBR and store-on-use shortcomings
- correctness does not depend on the scope of the DAG set
(but efficacy & complexity of the algorithm does)
- needed for hardware with small memory or expensive memory access
- starting point for future improvements:
 - consider user directives
 - consider static cost estimates for subroutine calls
 - consider other search heuristics
 - consider recomputations across DAG sequences

summary & observations

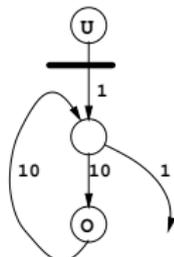
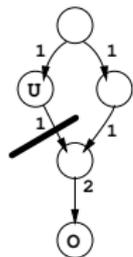
- algorithm complements earlier alg. on inverse loop computations
- first step towards automatic recomputation tradeoff
- addresses TBR and store-on-use shortcomings
- correctness does not depend on the scope of the DAG set
(but efficacy & complexity of the algorithm does)
- needed for hardware with small memory or expensive memory access
- starting point for future improvements:
 - consider user directives
 - consider static cost estimates for subroutine calls
 - consider other search heuristics
 - consider recomputations across DAG sequences
 - weakest point of the cost estimate is the lack of control flow information, so...

the last slide - idea on placing (re)store wrt. control flow

- ongoing work with Hascoët and Naumann
- consider an augmented control flow graph

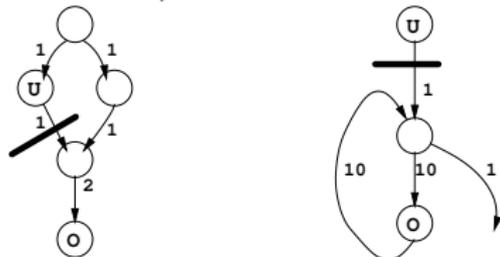
the last slide - idea on placing (re)store wrt. control flow

- ongoing work with Hascoët and Naumann
- consider an augmented control flow graph
- edge labels represent increased cost of stores in loops, decreased cost of stores in (mutually exclusive) branches

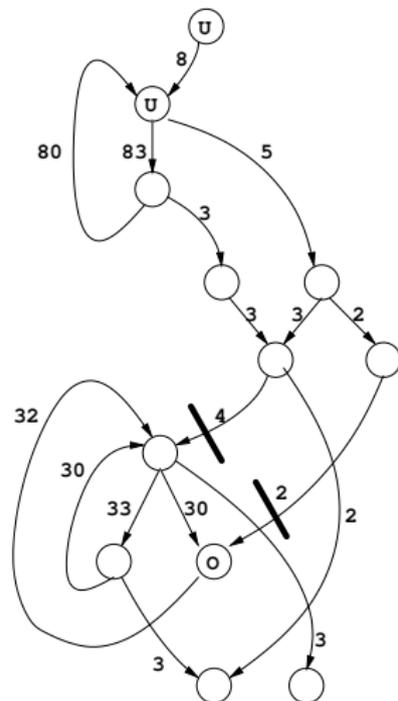


the last slide - idea on placing (re)store wrt. control flow

- ongoing work with Hascoët and Naumann
- consider an augmented control flow graph
- edge labels represent increased cost of stores in loops, decreased cost of stores in (mutually exclusive) branches

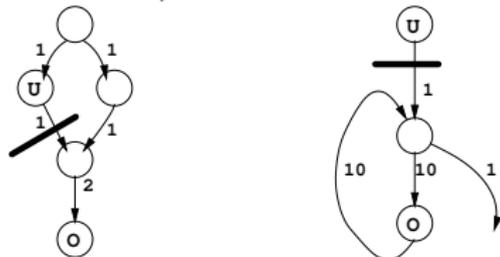


- max flow / min cut gives cheapest* store placement



the last slide - idea on placing (re)store wrt. control flow

- ongoing work with Hascoët and Naumann
- consider an augmented control flow graph
- edge labels represent increased cost of stores in loops, decreased cost of stores in (mutually exclusive) branches



- max flow / min cut gives cheapest* store placement
- eventual connection to recomputation via modified G_b

