Parallel Flow-Sensitive Pointer Analysis by Graph-Rewriting

Vaivaswatha Nagaraj
R. Govindarajan

Indian Institute of Science, Bangalore

PACT 2013

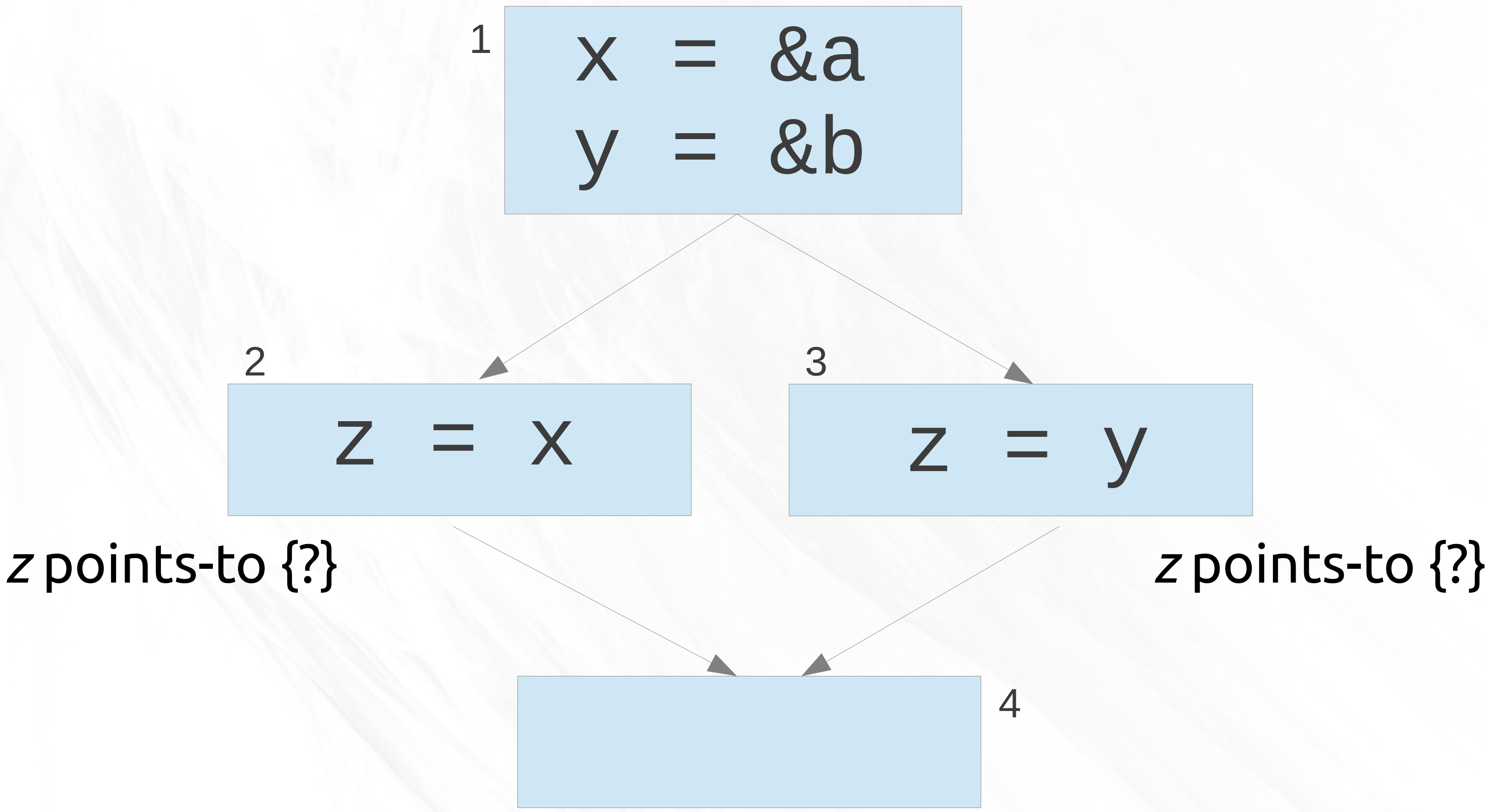
Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion

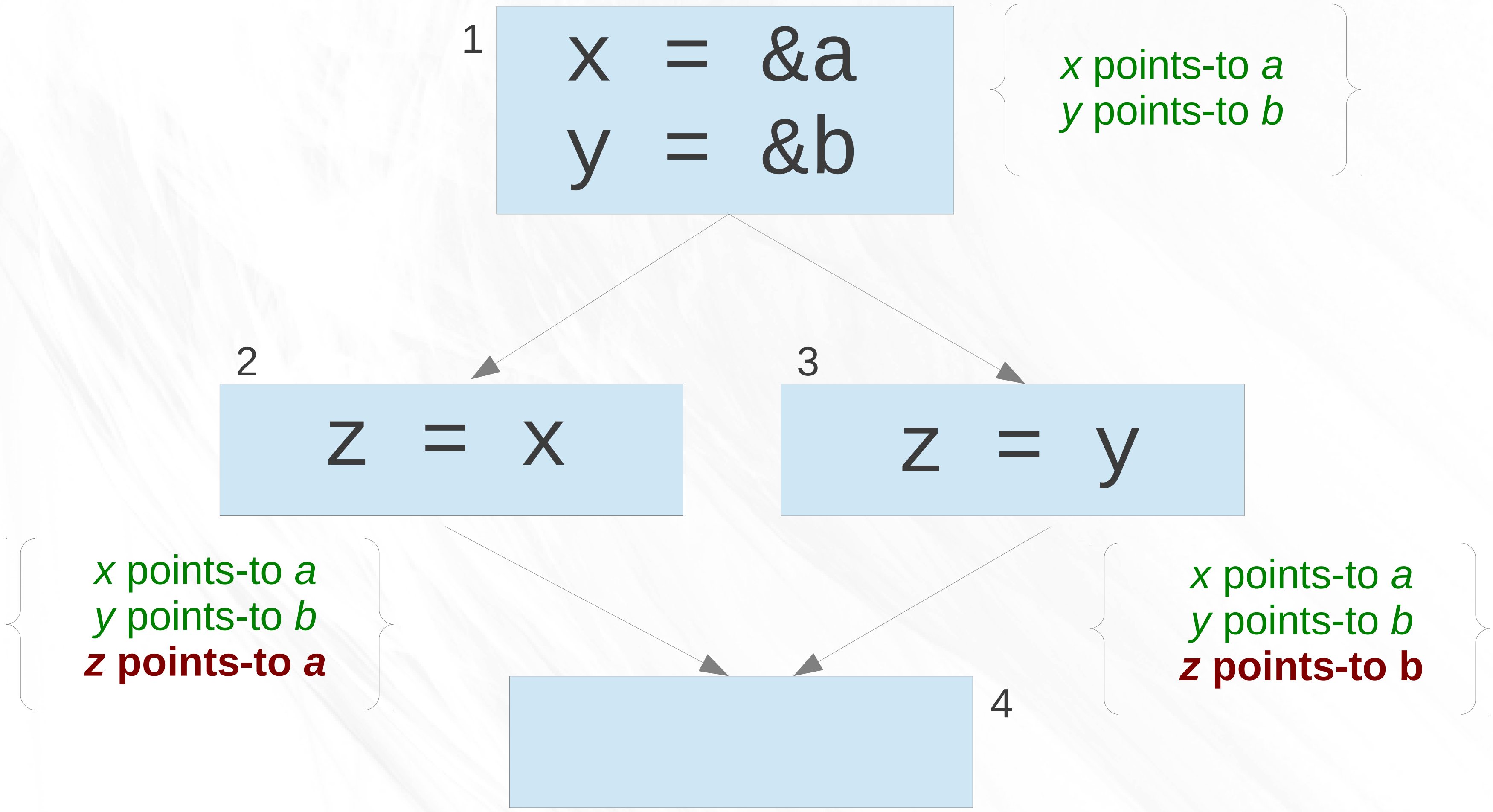
Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion

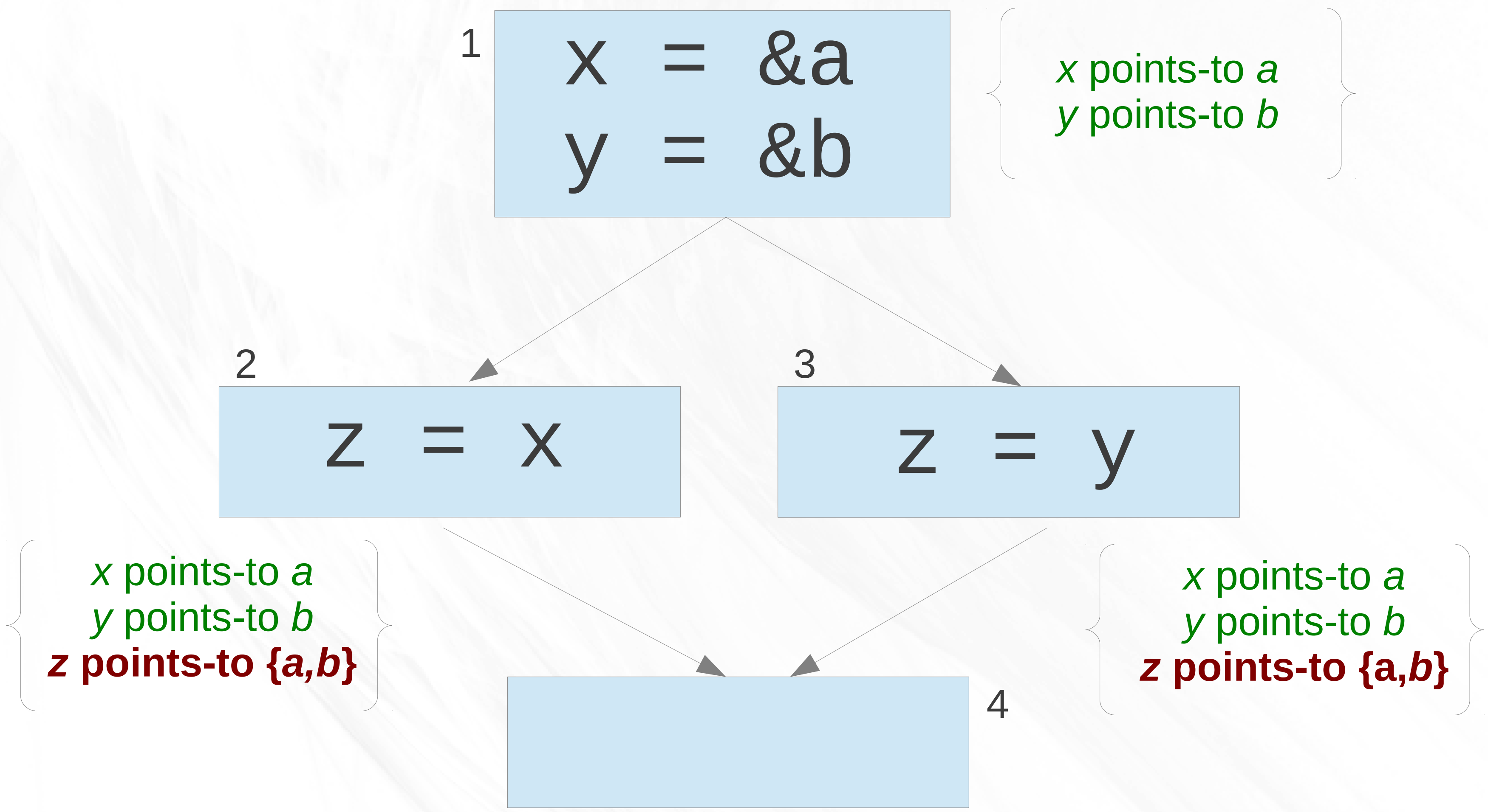
Flow-sensitive pointer analysis



Flow-sensitive pointer analysis



Flow-sensitive vs flow-insensitive



Outline

- Introduction
- Background
  - Staged flow-sensitive analysis
  - Graph-rewriting
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion

Staged flow-sensitive pointer analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]

- Scales to large programs
- Faster, less precise analysis used to speed up the primary analysis

Staged flow-sensitive pointer analysis

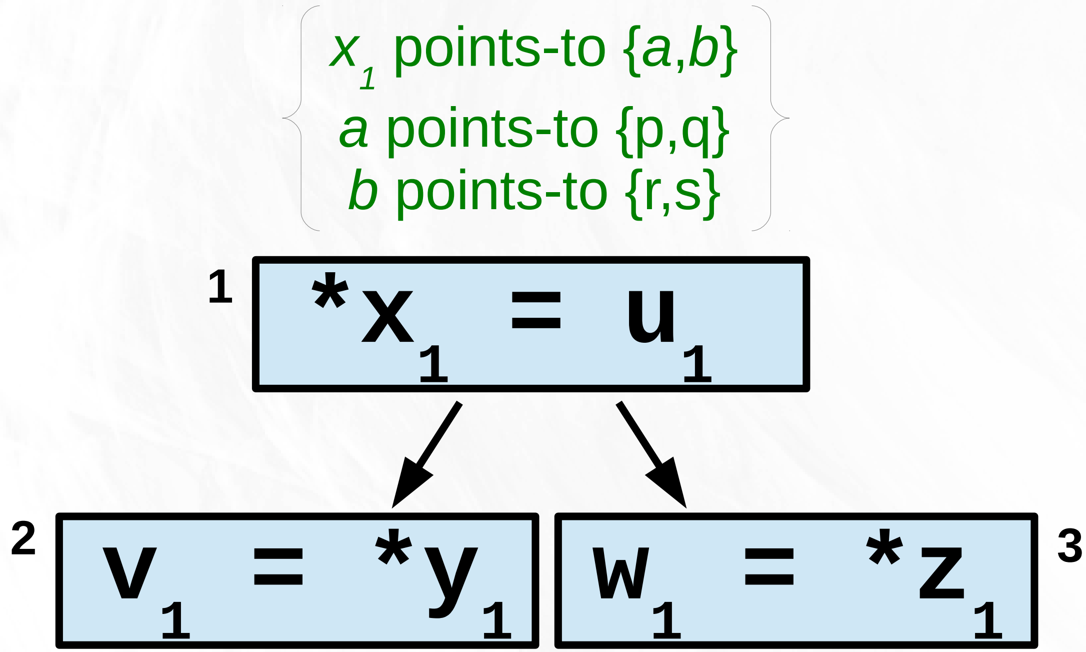
[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]

- Scales to large programs
- Faster, less precise analysis used to speed up the primary analysis

Our goal: Parallelize the staged flow-sensitive pointer analysis

Staged flow-sensitive analysis

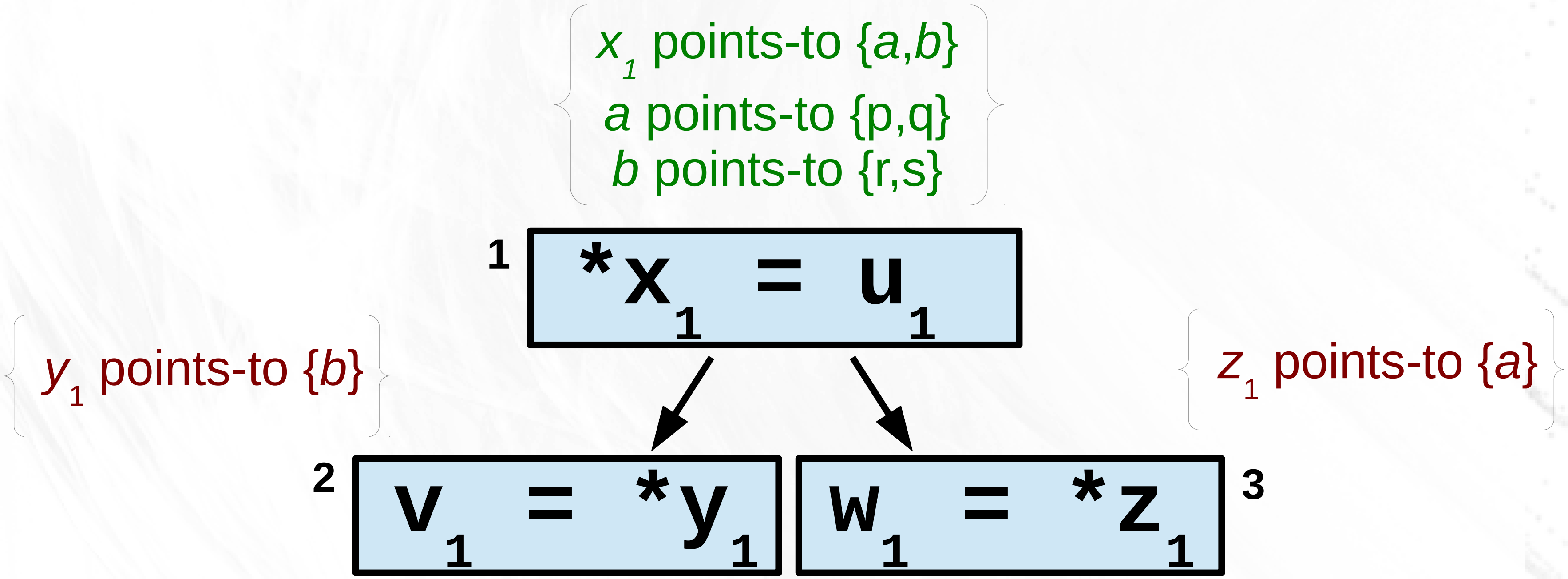
[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]



In a traditional analysis, points-to info of both *a* & *b* will be propagated to both **2** and **3**

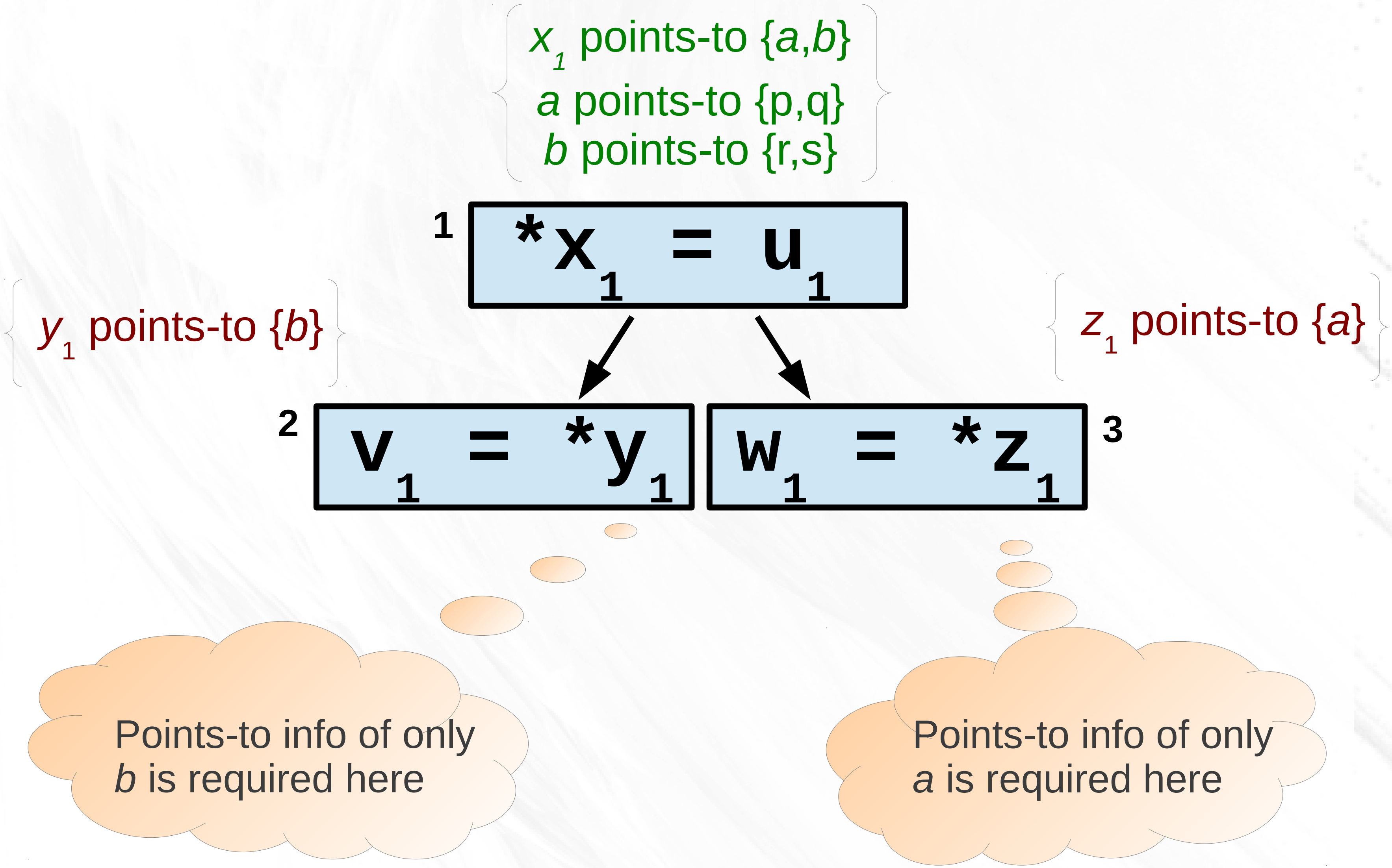
Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]



Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]



Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]

How does staged analysis work?

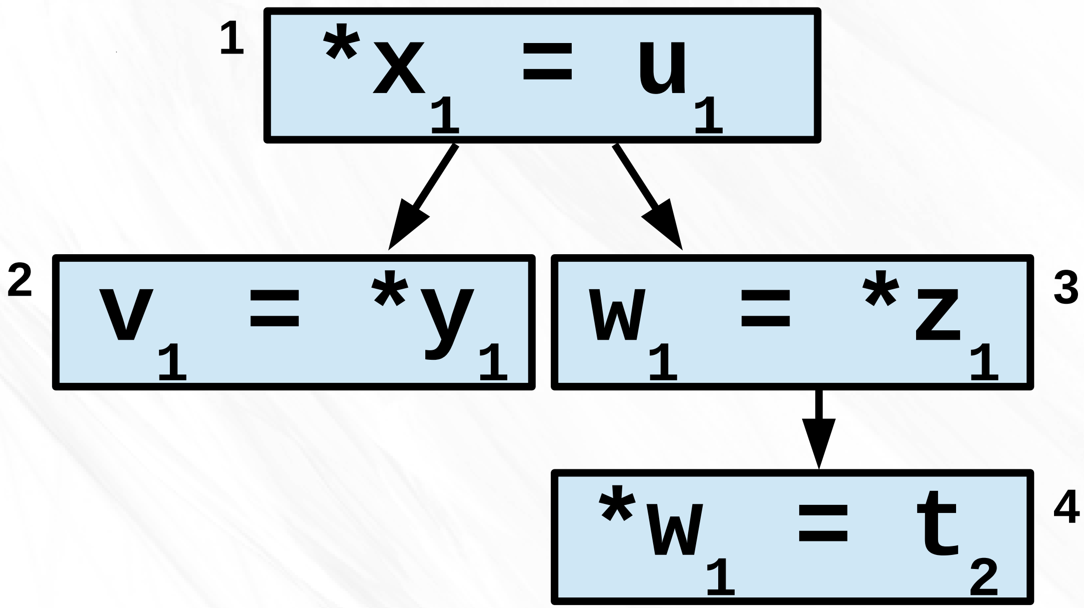
(1) Use a fast analysis to get less precise points-to information

$x_1 \rightarrow \{a,b\}$

$z_1 \rightarrow \{a\}$

$y_1 \rightarrow \{b\}$

$w_1 \rightarrow \{c\}$



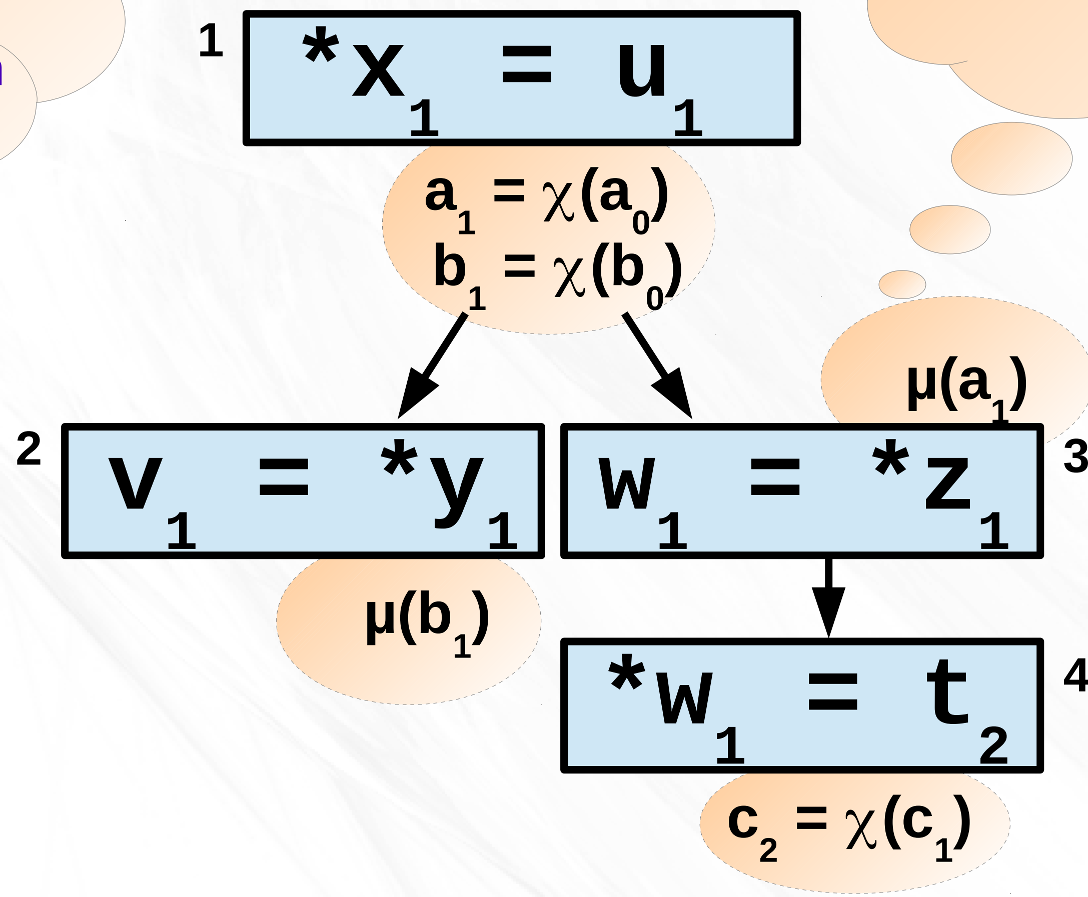
Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]

How does staged analysis work?

(1) Use a fast analysis to get less precise points-to information

(2) Use the less precise info to find variables potentially referenced at indirections



$x_1 \rightarrow \{a,b\}$

$z_1 \rightarrow \{a\}$

$y_1 \rightarrow \{b\}$

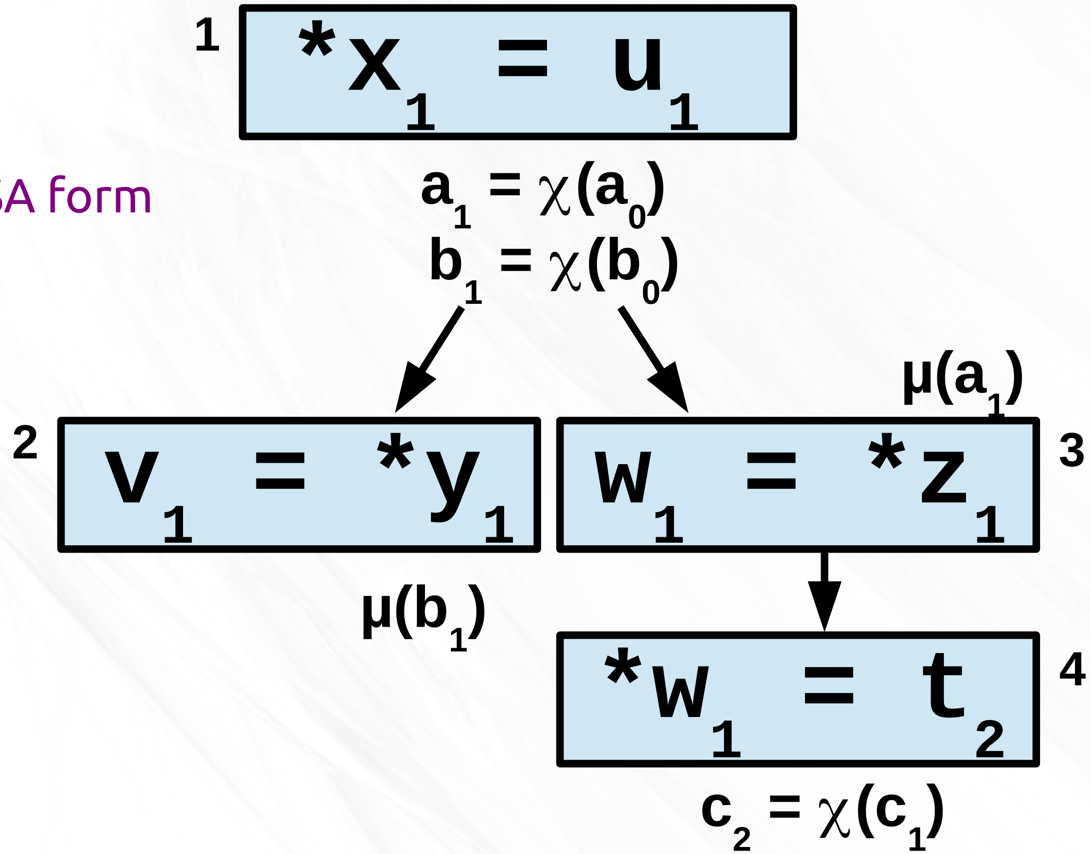
$w_1 \rightarrow \{c\}$

Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]

How does staged analysis work?

Flow-sensitivity -
All variables in SSA form



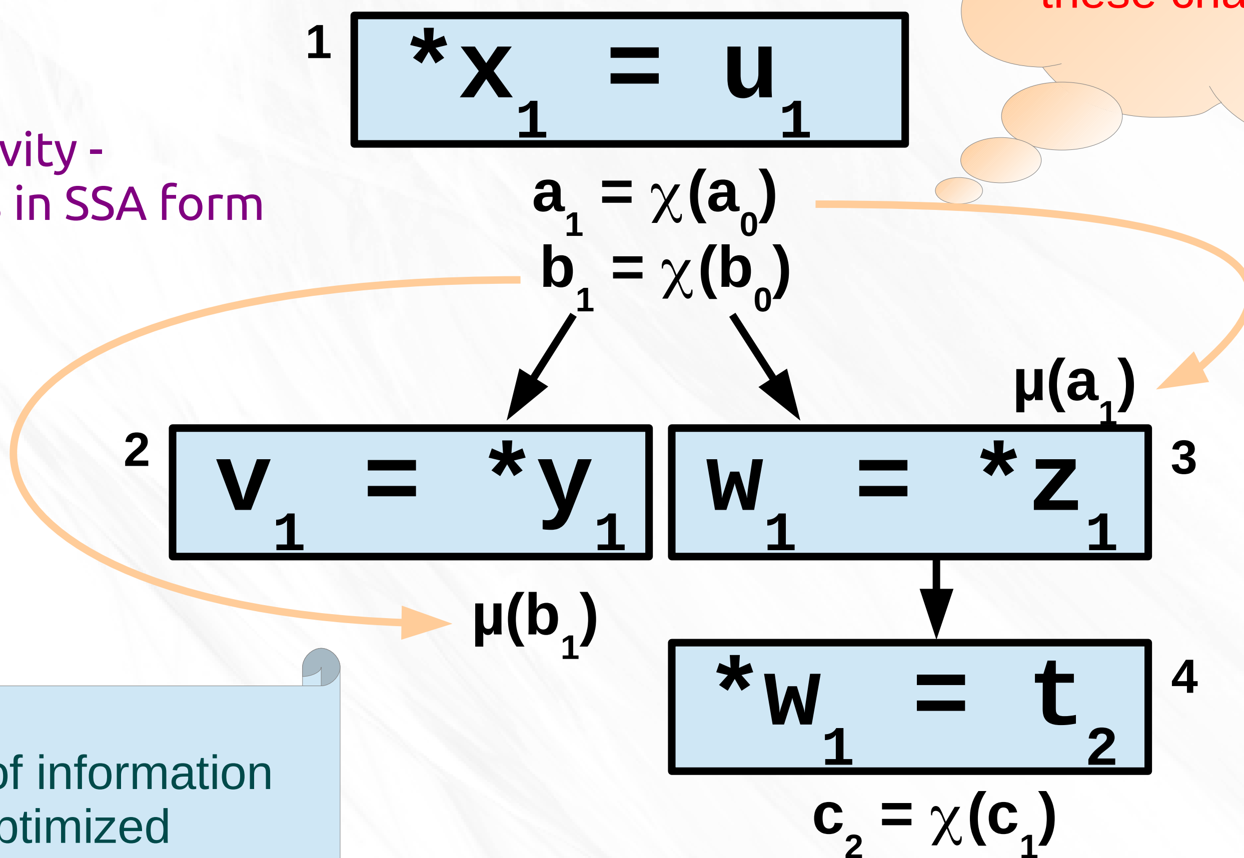
Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO'11]

How does staged analysis work?

(3) Build def-use chains and propagate only along these chains

Flow-sensitivity -
All variables in SSA form



Propagation of information is thus optimized

Parallelizing the staged analysis

Can the staged flow-sensitive pointer analysis be parallelized and scaled to multiple cores?

Towards a parallel algorithm

- Flow-insensitive pointer analysis has already been parallelized#
- Parallelization of flow-insensitive analysis involves transforming it to a graph-rewriting problem – expose *amorphous data-parallelism*

#[Parallel Inclusion-based Points-to Analysis - Mario Mendez-Lojo, et al. - OOPSLA'10]

Towards a parallel algorithm

- Flow-insensitive pointer analysis has already been parallelized#
- Parallelization of flow-insensitive analysis involves transforming it to a graph-rewriting problem – expose *amorphous data-parallelism*

Our goal: Parallelize the staged flow-sensitive pointer analysis

→ Formulate it as a graph-rewriting problem

#[Parallel Inclusion-based Points-to Analysis - Mario Mendez-Lojo, et al. - OOPSLA'10]

Outline

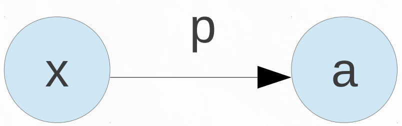
- Introduction
- Background
  - Staged flow-sensitive analysis
  - **Graph-rewriting**
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion

Graph-rewriting

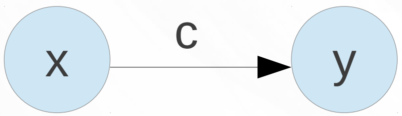
[Parallel Inclusion-based Points-to Analysis - Mario Mendez-Lojo, et al. - OOPSLA'10]

Points-to constraint | Constraint graph

x = &a

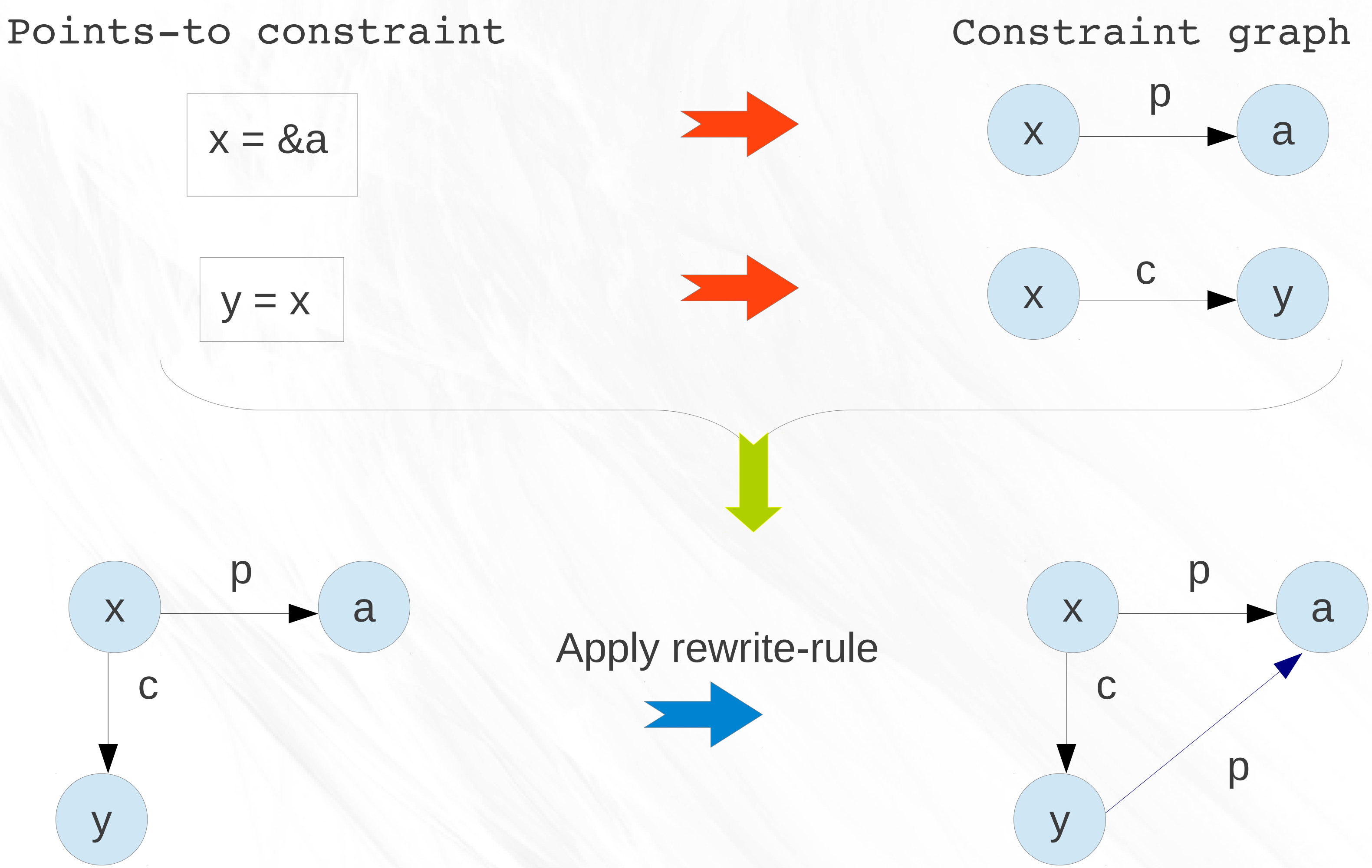


y = x



Graph-rewriting

[Parallel Inclusion-based Points-to Analysis - Mario Mendez-Lojo, et al. - OOPSLA'10]

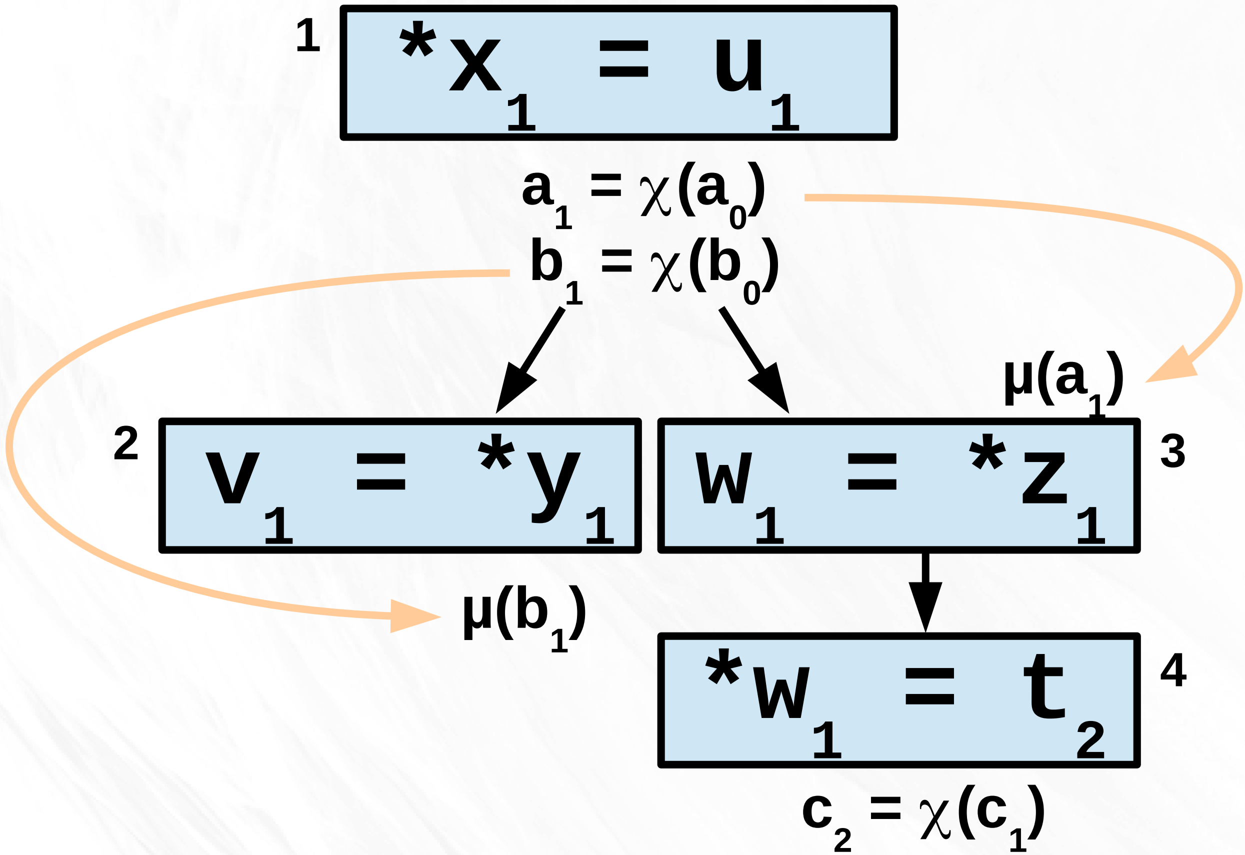


Example: copy rewrite rule

Outline

- Introduction
- Background
- **Flow-sensitive graph-rewriting formulation**
- Implementation and Results
- Conclusion

Graph-rewriting formulation



Graph:

- Nodes: Variables in the program
- Edges: Points-to, copy, load, store, etc ...
- Rewrite rules?

Challenges

Directly using the flow-insensitive graph formulation for flow-sensitive analysis leads to the following challenges

- Spurious edges – leads to imprecision
- Strong and weak updates at store constraints are not handled

Challenges

Directly using the flow-insensitive graph formulation for flow-sensitive analysis leads to the following challenges

- Spurious edges – leads to imprecision
  - Solution: potential edges
- Strong and weak updates at store constraints are not handled
  - Solution: *klique* nodes

Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
  - Problem of spurious edges
  - Handling strong and weak updates
- Implementation and Results
- Conclusion

Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

$y_1 = \&a$

$x_1 = *y_1$

Spurious edges

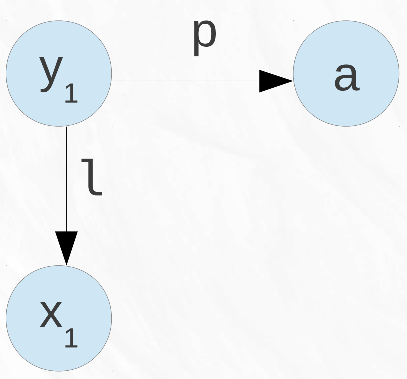
- Load rule for flow-insensitive analysis:

Points-to constraints

$y_1 = \&a$

$x_1 = *y_1$

Formulate graph



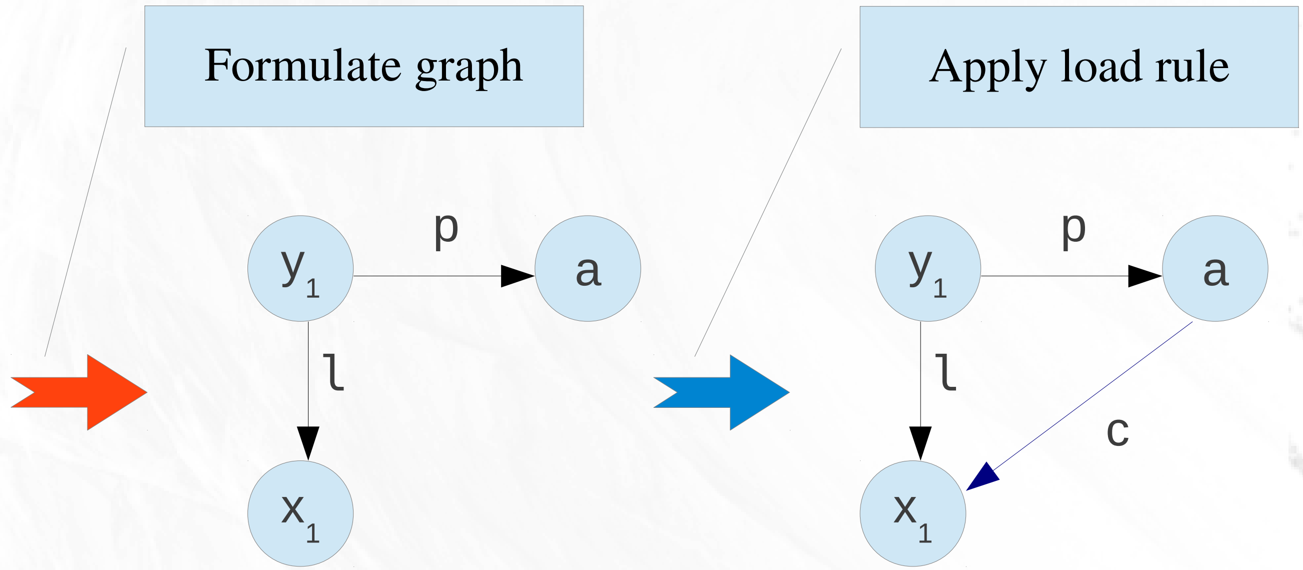
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

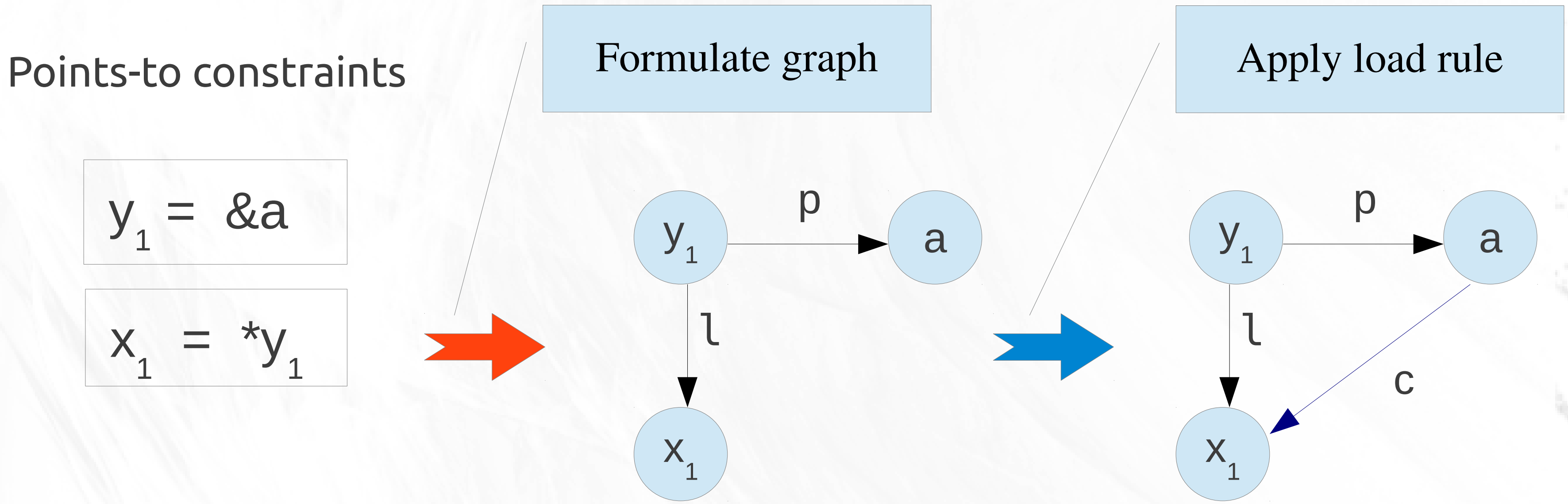
$y_1 = \&a$

$x_1 = *y_1$

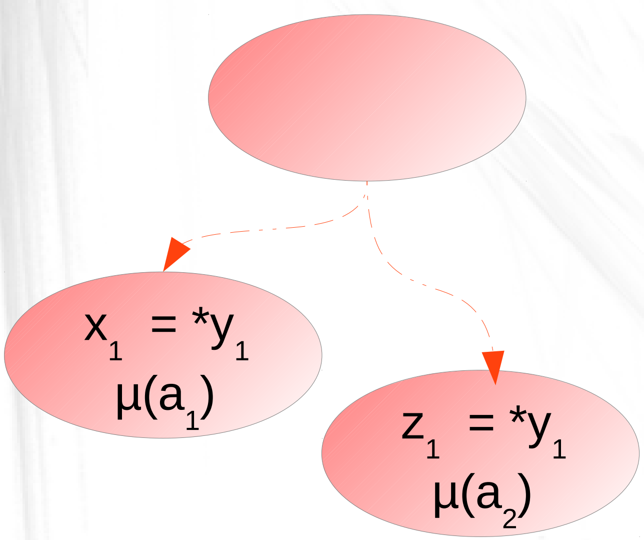


Spurious edges

- Load rule for flow-insensitive analysis:



- Loss of precision when used for flow-sensitive analysis:



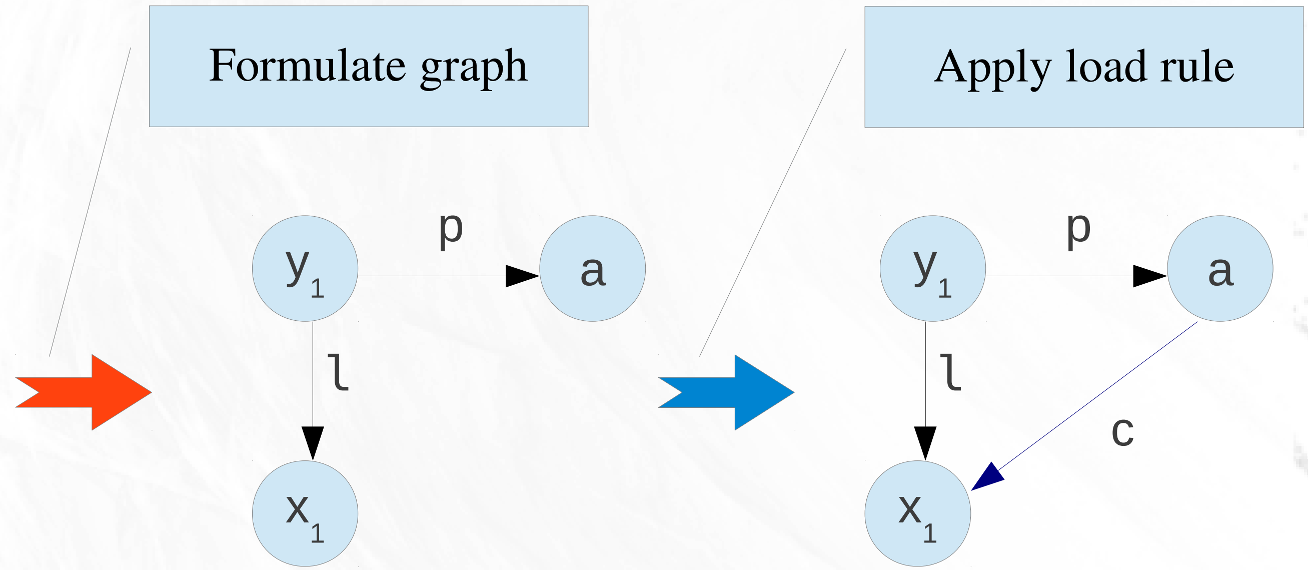
Spurious edges

- Load rule for flow-insensitive analysis:

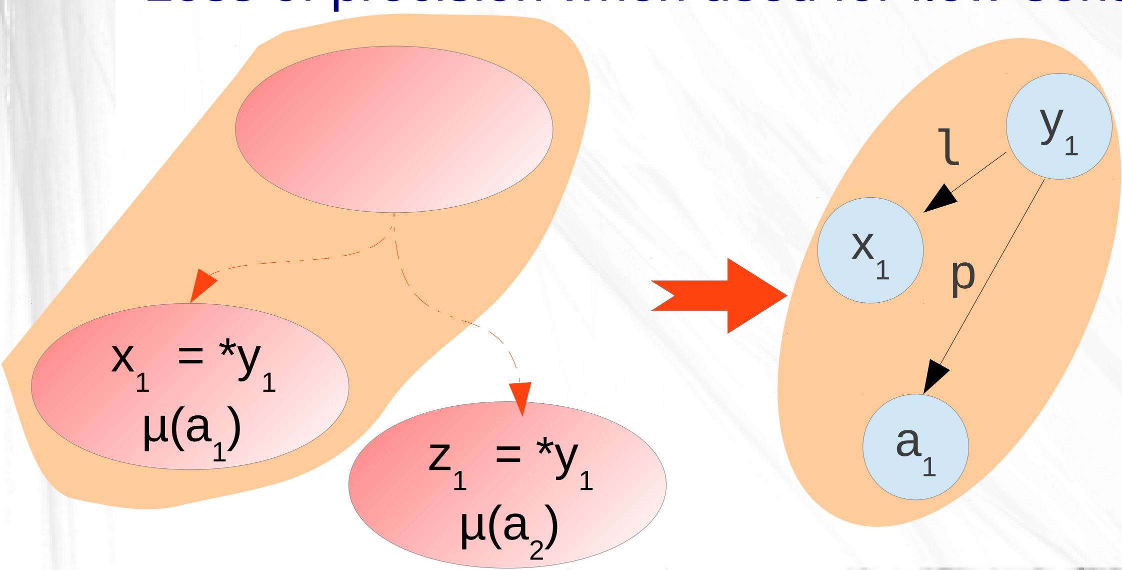
Points-to constraints

y_1 = &a

x_1 = *y_1



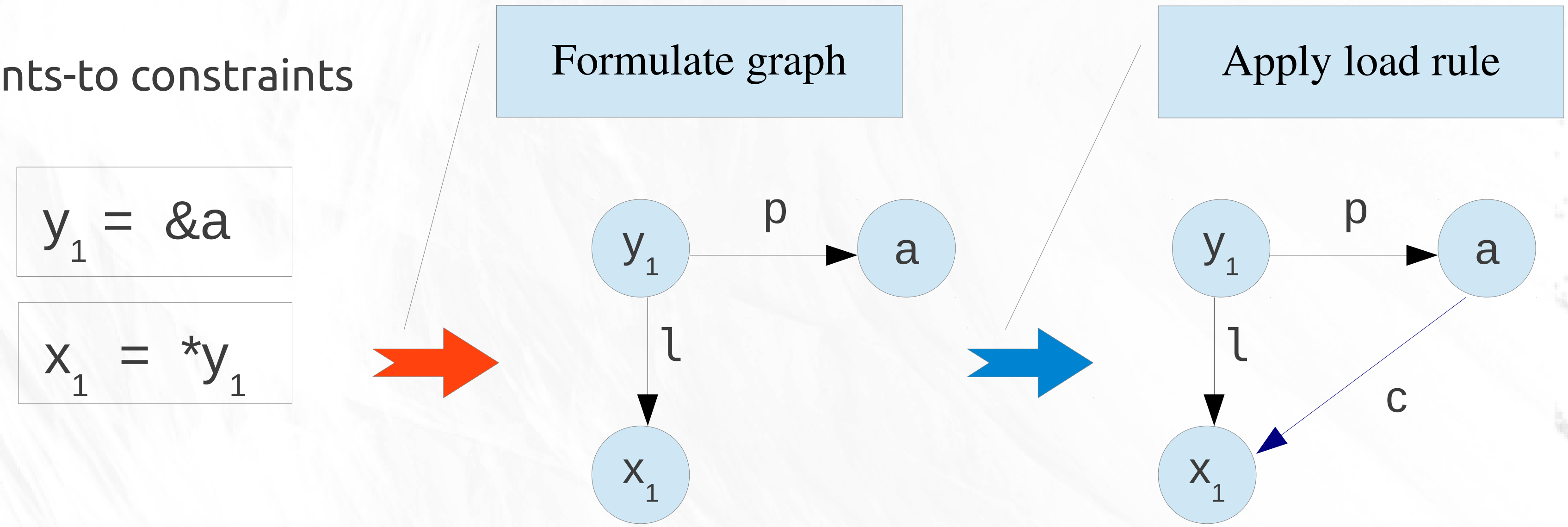
- Loss of precision when used for flow-sensitive analysis:



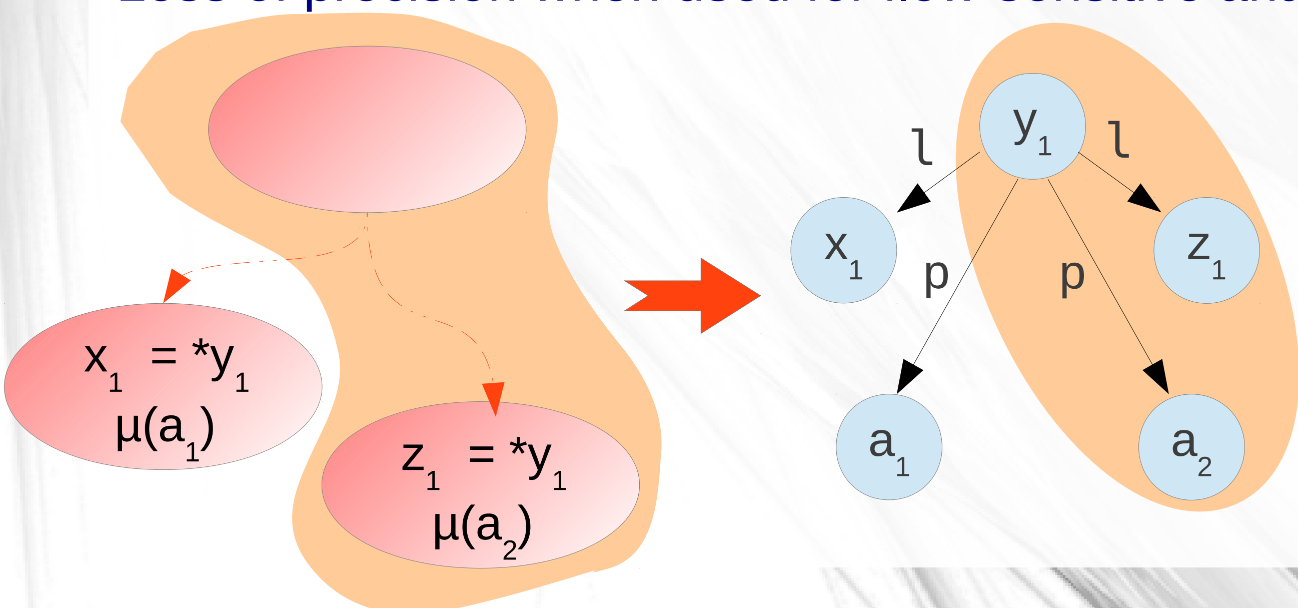
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints



- Loss of precision when used for flow-sensitive analysis:



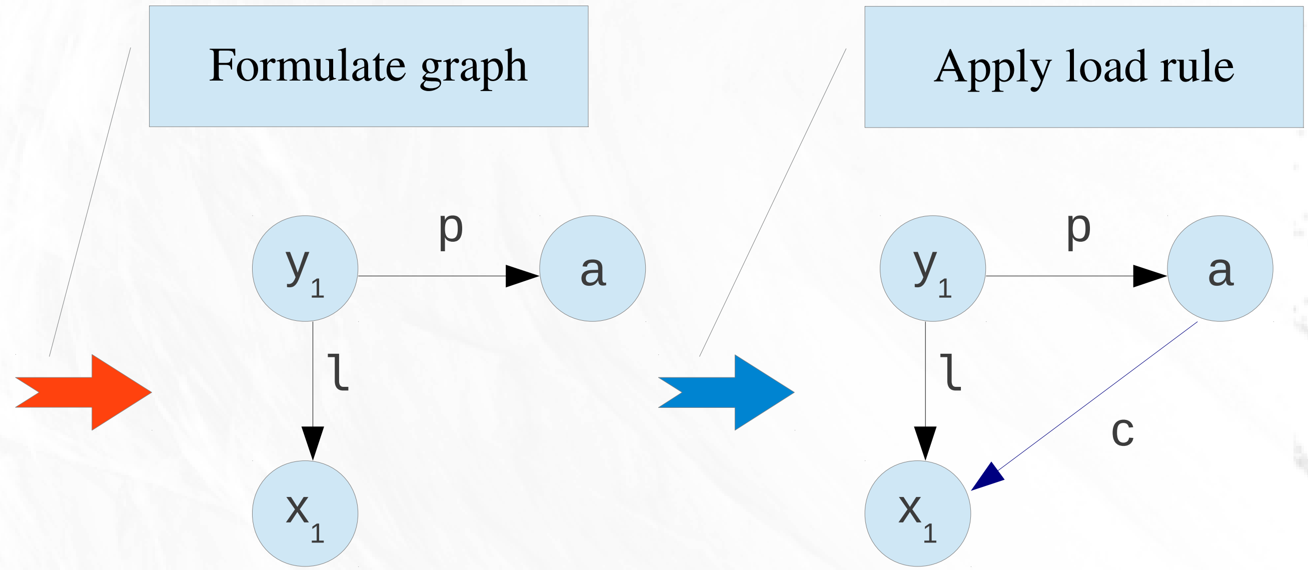
Spurious edges

- Load rule for flow-insensitive analysis:

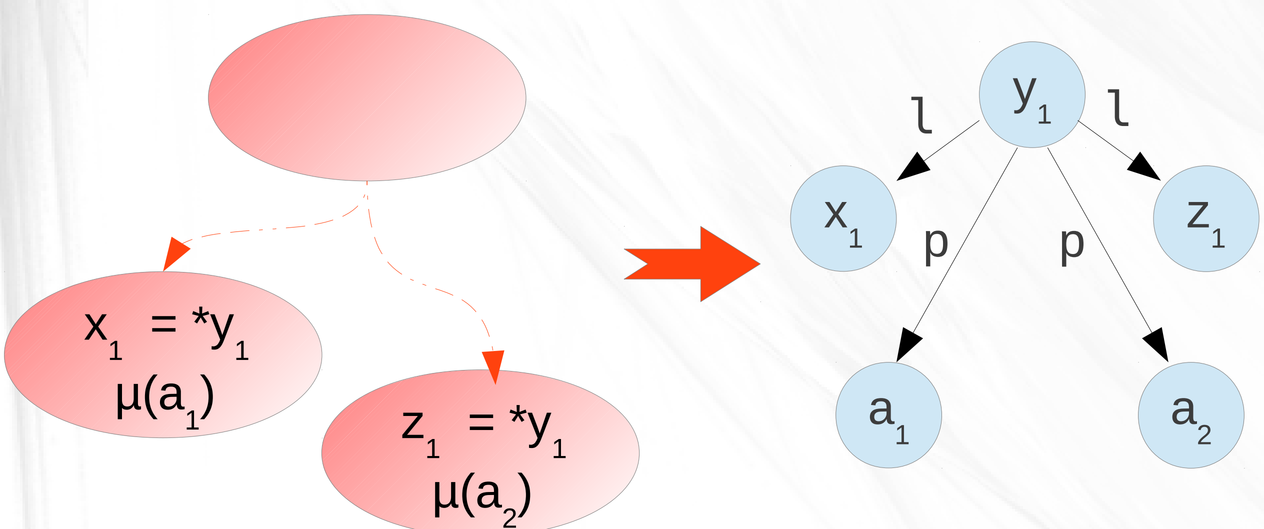
Points-to constraints

$y_1 = \&a$

$x_1 = *y_1$

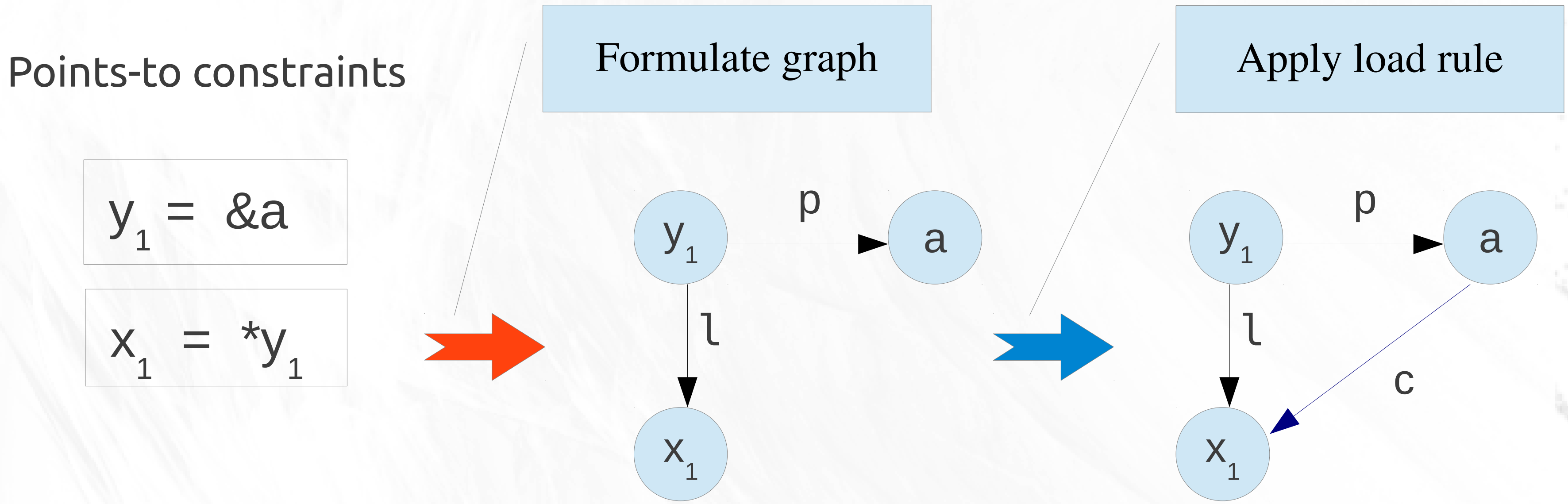


- Loss of precision when used for flow-sensitive analysis:

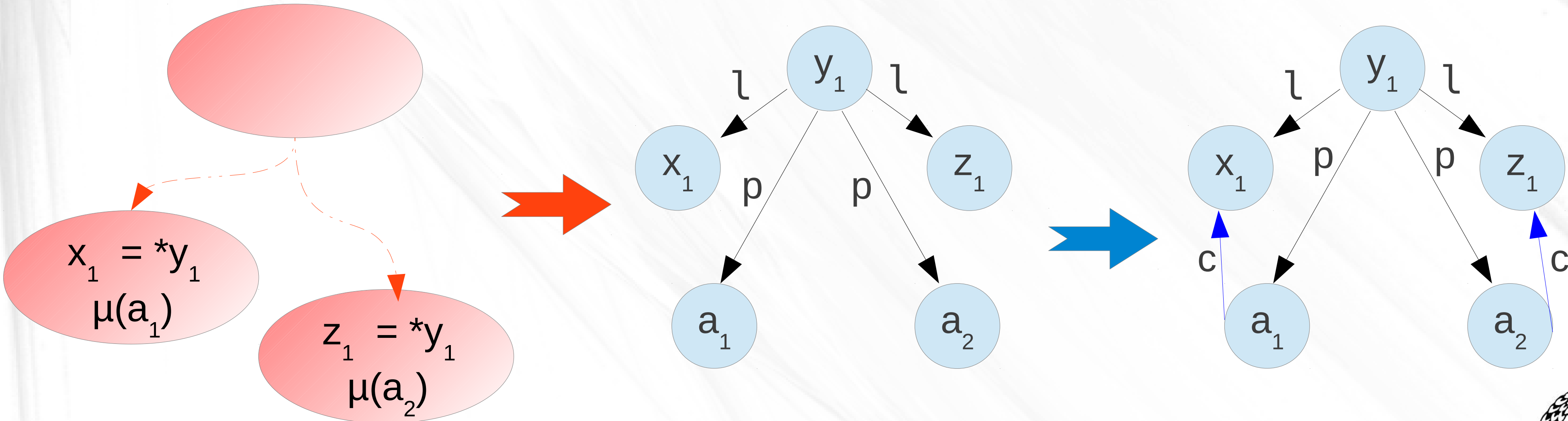


Spurious edges

- Load rule for flow-insensitive analysis:

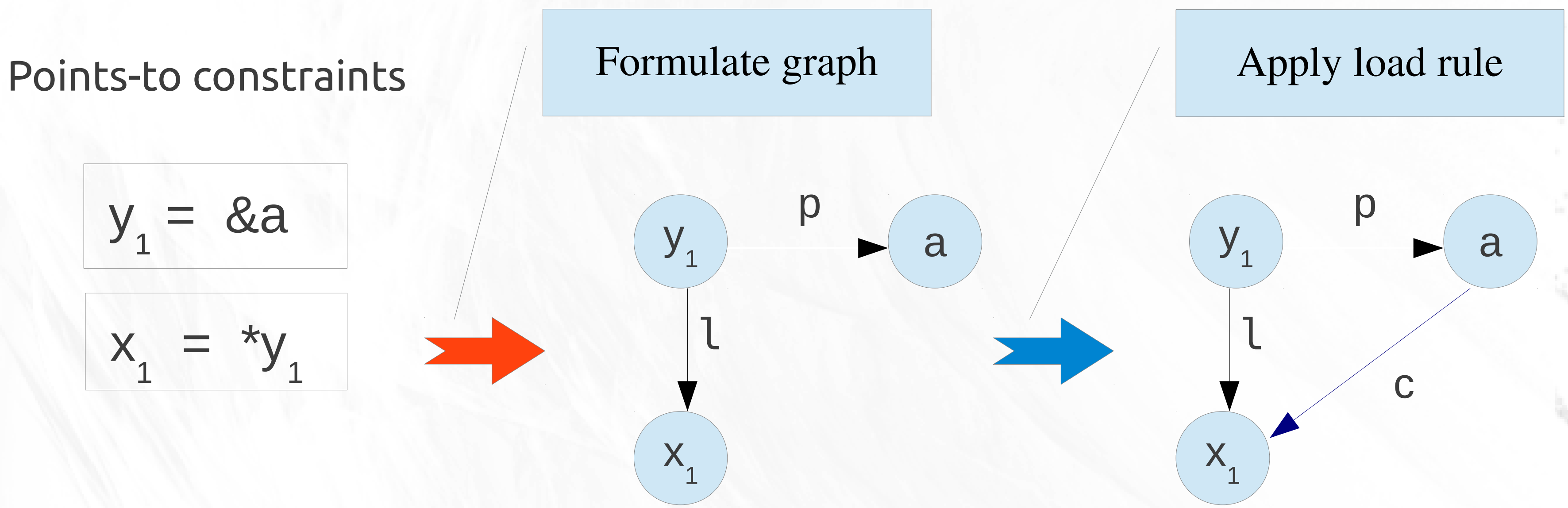


- Loss of precision when used for flow-sensitive analysis:

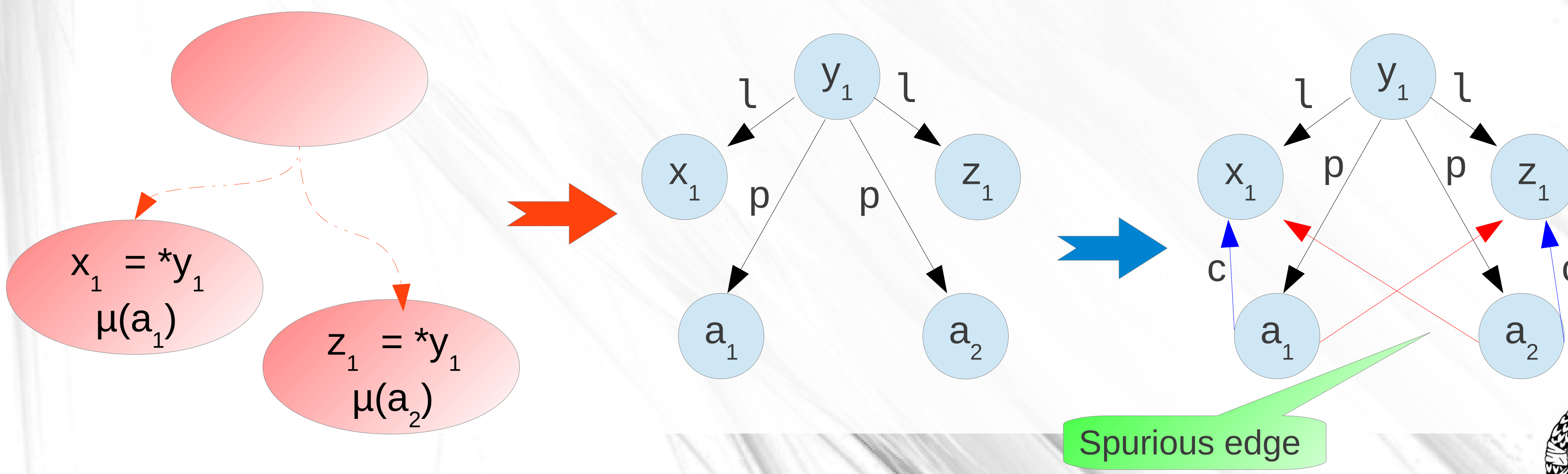


Spurious edges

- Load rule for flow-insensitive analysis:

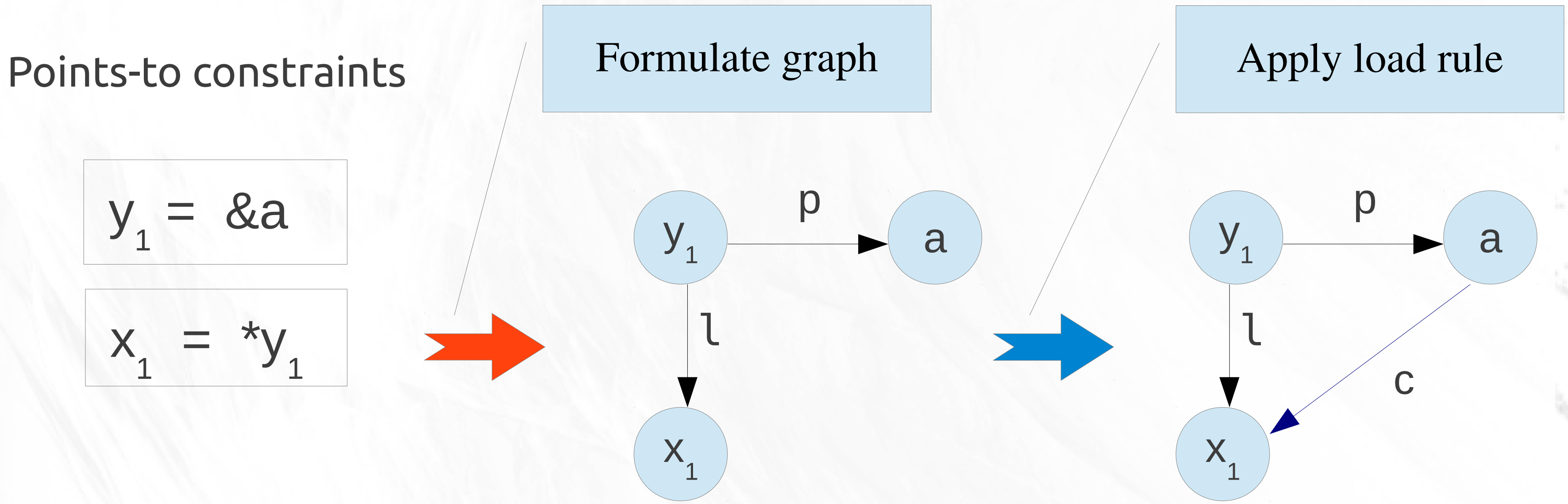


- Loss of precision when used for flow-sensitive analysis:

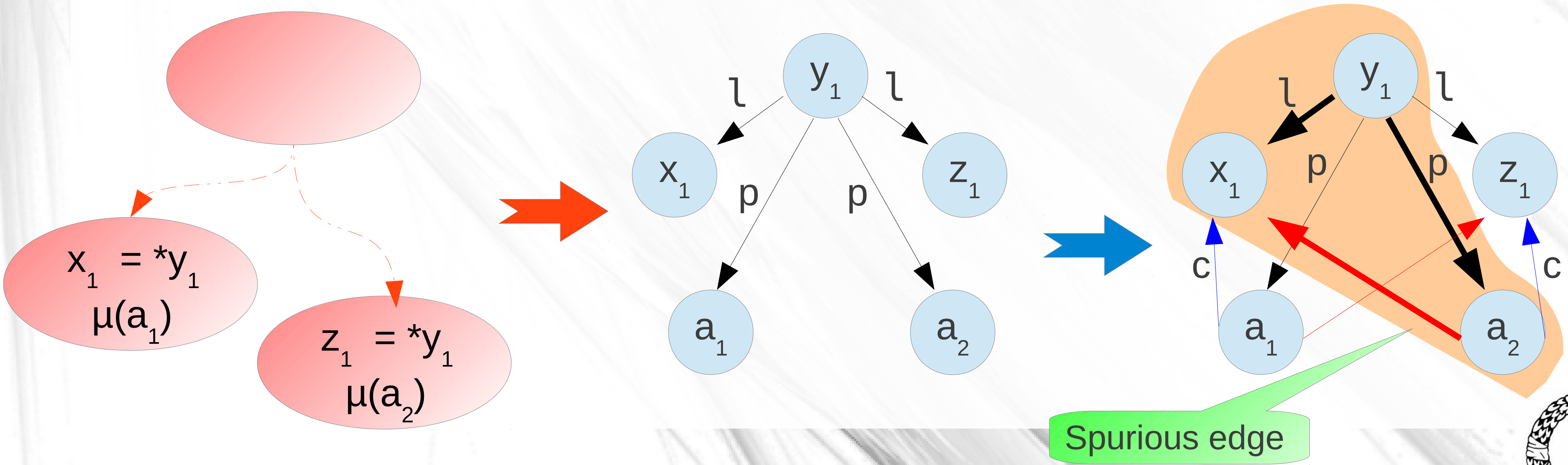


Spurious edges

- Load rule for flow-insensitive analysis:

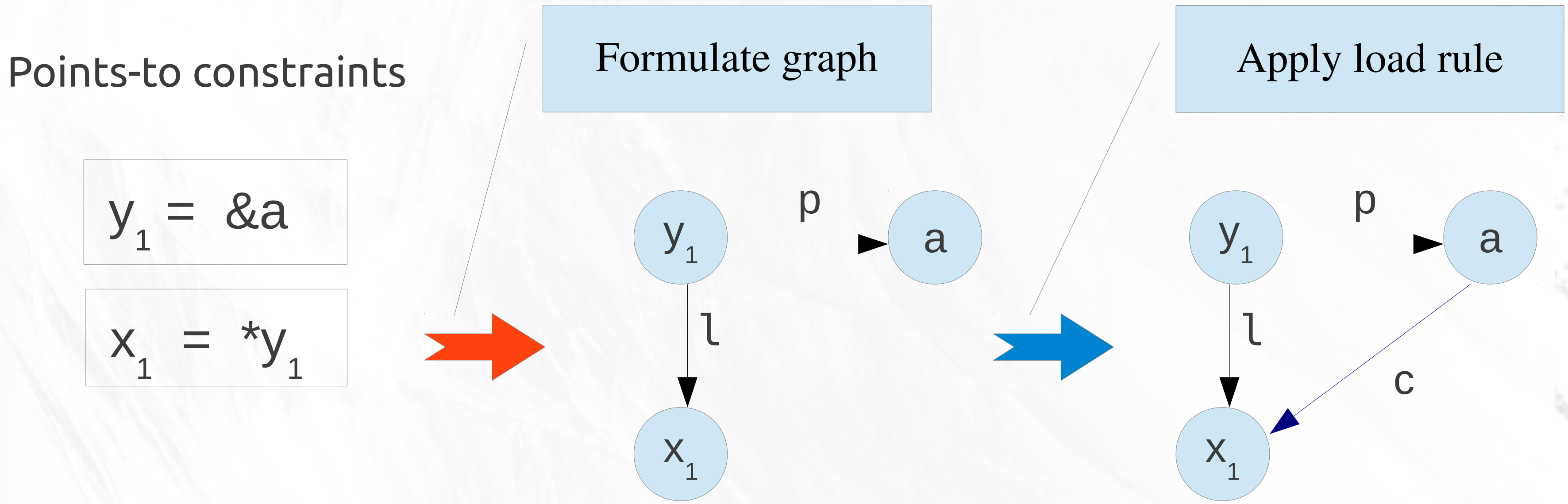


- Loss of precision when used for flow-sensitive analysis:

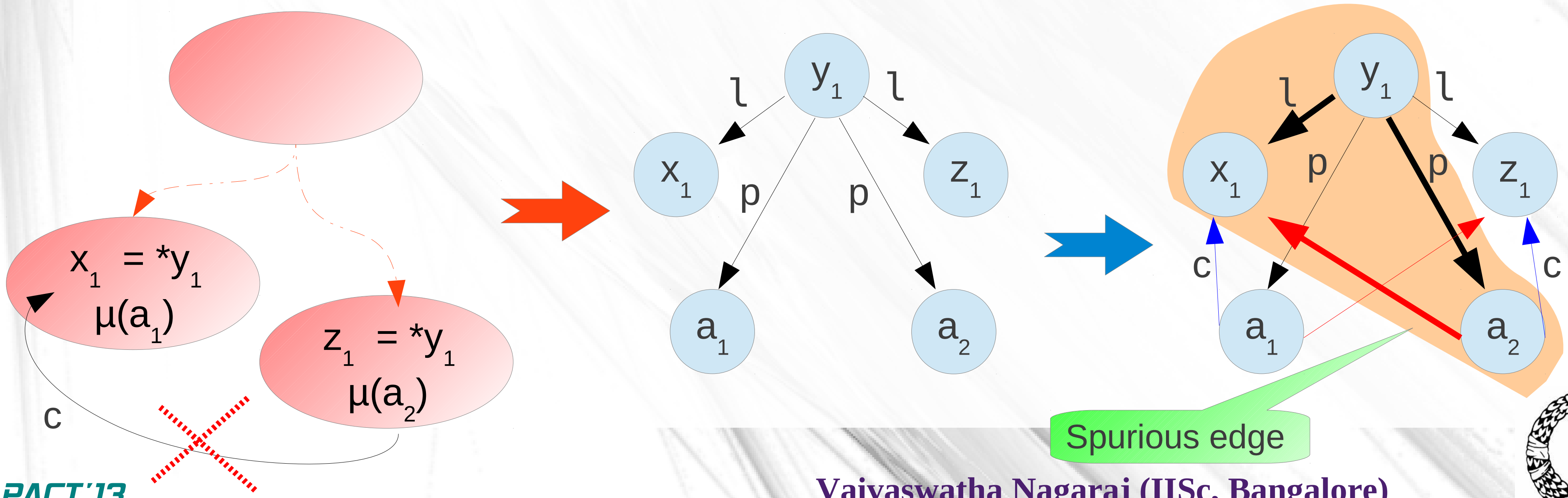


Spurious edges

- Load rule for flow-insensitive analysis:



- Loss of precision when used for flow-sensitive analysis:



Solution: Potential edges

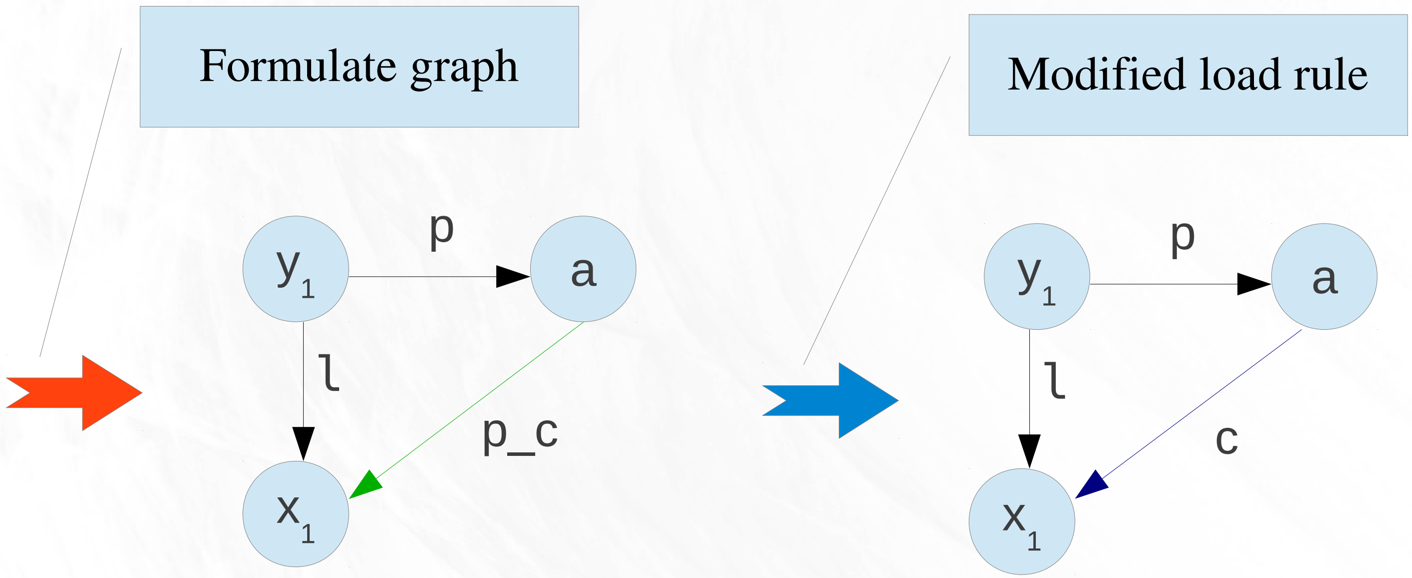
- A *potential* edge of type *t* means that, there could be an actual edge of type *t*, between the same two nodes
- *Potential* edges are added during initial graph construction
- Some rewrite rules are modified to look for potential edges

Solution: Potential edges

- Modified load rule:

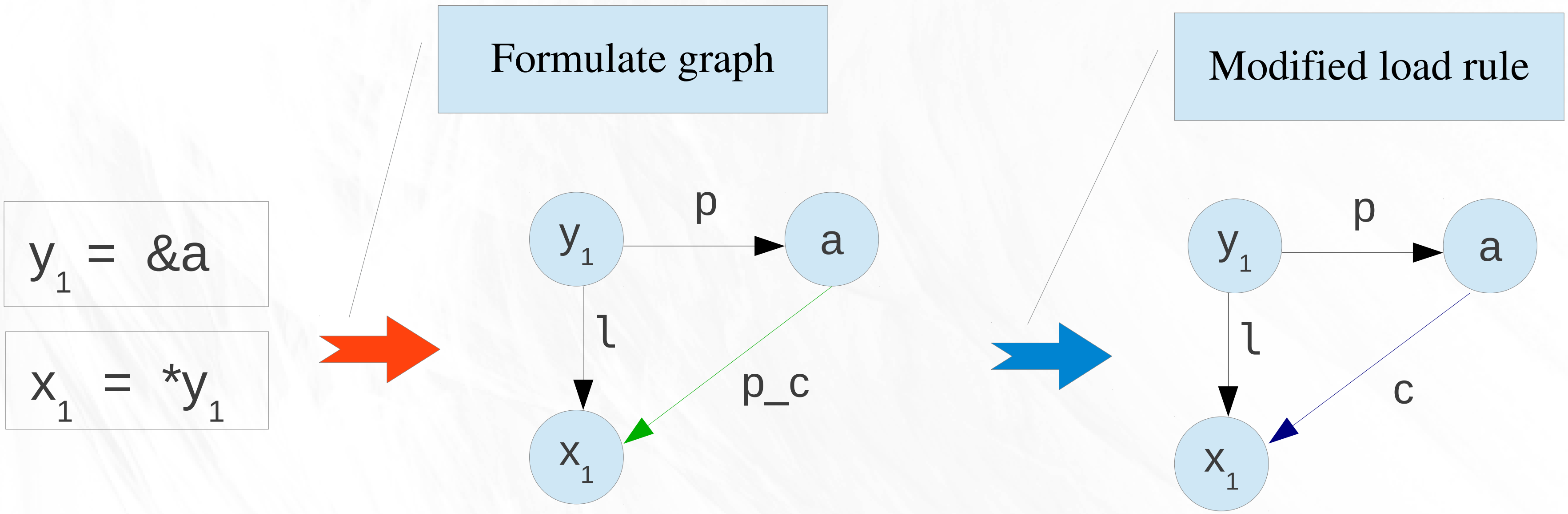
$y_1 = \&a$

$x_1 = *y_1$

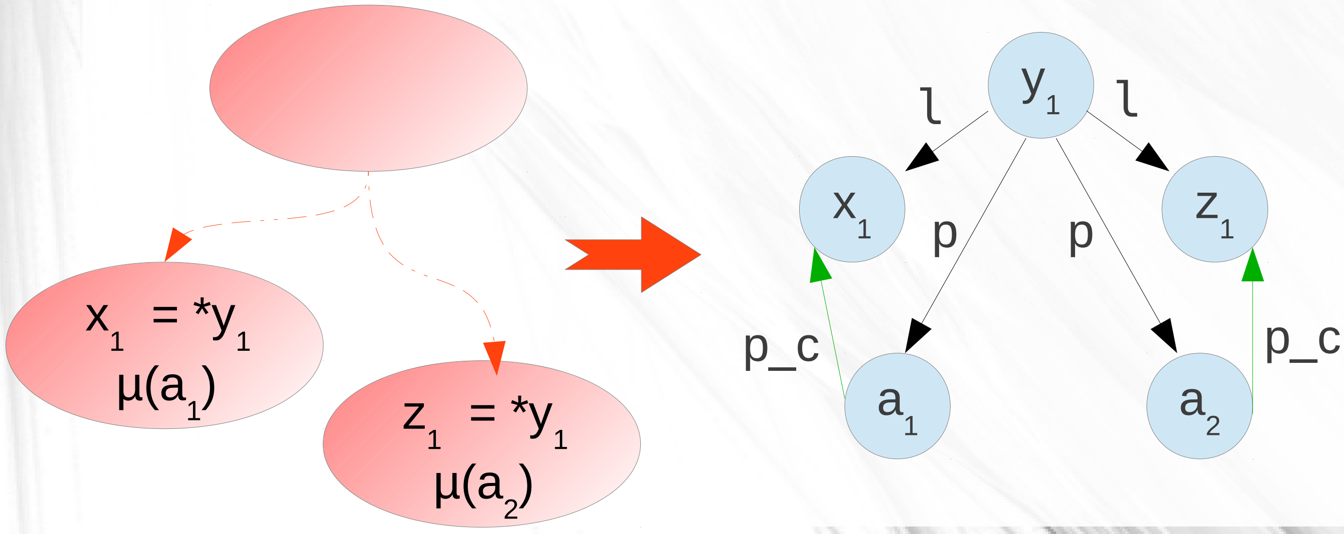


Solution: Potential edges

- Modified load rule:



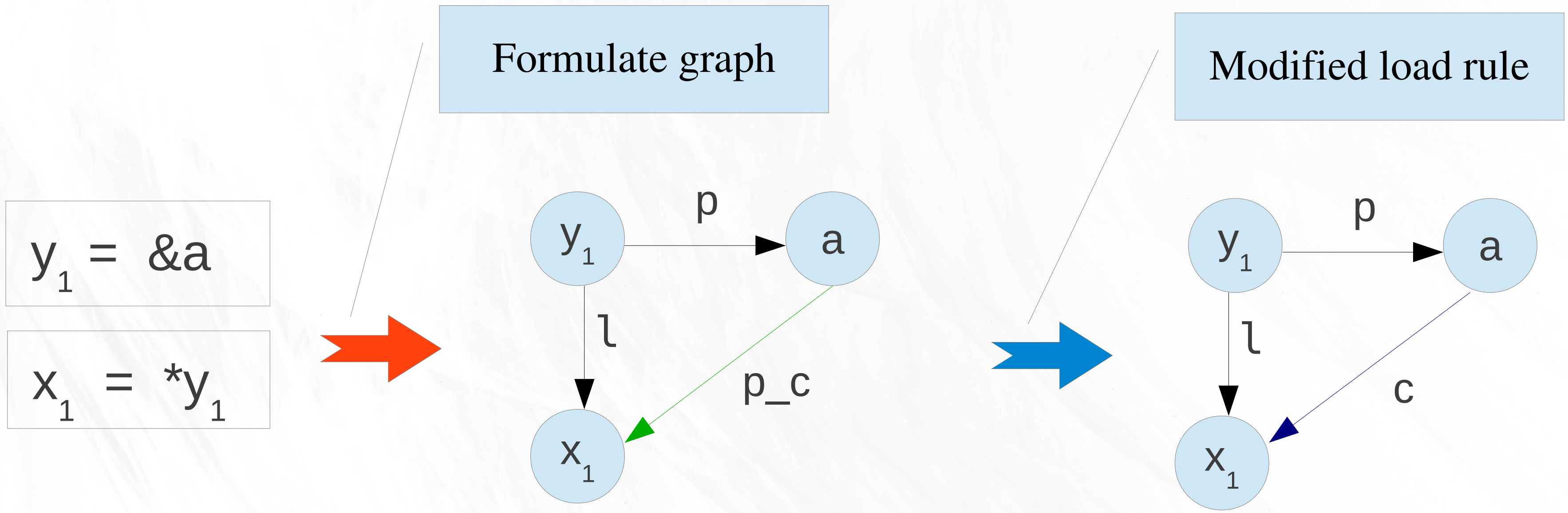
- Modified rule applied to flow-sensitive analysis:



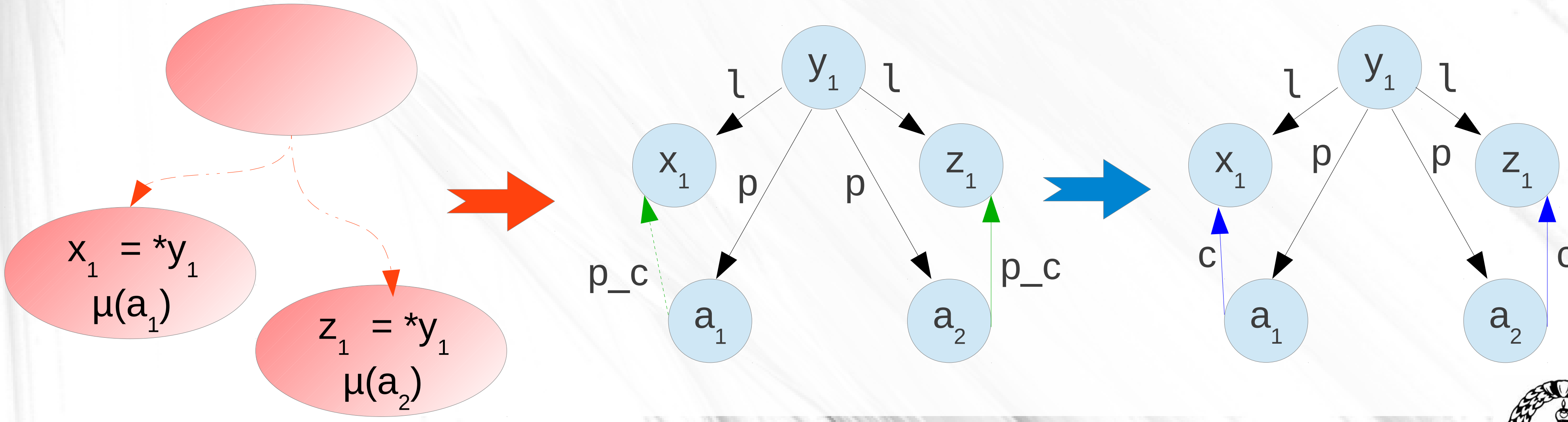
Vaivaswatha Nagaraj (IISc, Bangalore)

Solution: Potential edges

- Modified load rule:



- Modified rule applied to flow-sensitive analysis:



Vaivaswatha Nagaraj (IISc, Bangalore)

Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
  - Problem of spurious edges
  - **Handling strong and weak updates**
- Implementation and Results
- Conclusion

Strong and weak updates

As the flow-sensitive analysis progresses:

$$*x_1 = u_1$$

$$a_1 = \chi(a_0)$$

$$b_1 = \chi(b_0)$$

Strong and weak updates

As the flow-sensitive analysis progresses:

How to update a_1?

$$*x_1 = u_1$$

$$a_1 = \chi(a_0)$$

$$b_1 = \chi(b_0)$$

Strong and weak updates

As the flow-sensitive analysis progresses:

How to update a_1?

$*x_1 = u_1$

$a_1 = \chi(a_0)$

$b_1 = \chi(b_0)$

x_1 points-to $\{a_1\}$

Strong and weak updates

As the flow-sensitive analysis progresses:

How to update a_1?

$$*x_1 = u_1$$
$$a_1 = \chi(a_0)$$
$$b_1 = \chi(b_0)$$

Strong Update

x_1 points-to $\{a_1\}$

$a_1 \hookleftarrow u_1$

Strong and weak updates

As the flow-sensitive analysis progresses:

How to update a_1?

$$*x_1 = u_1$$
$$a_1 = \chi(a_0)$$
$$b_1 = \chi(b_0)$$

Strong Update

x_1 points-to $\{a_1\}$

$a_1 \hookleftarrow u_1$

x_1 points-to $\{a_1, b_1\}$

Strong and weak updates

As the flow-sensitive analysis progresses:

How to update a_1?

$$*x_1 = u_1$$

$$a_1 = \chi(a_0)$$
$$b_1 = \chi(b_0)$$

Strong Update

x_1 points-to $\{a_1\}$

$a_1 \leftarrow u_1$

Weak Update

x_1 points-to $\{a_1, b_1\}$

$a_1 \leftarrow u_1 \cup a_0$

Strong and weak updates

$*x_1 = u_1$

$a_1 = \chi(a_0)$

$b_1 = \chi(b_0)$

update a_1 *as*

if x_1 *points-to* a
then
    $a_1 \hookleftarrow u_1$
end if

if x_1 *points-to* b
then
    $a_1 \hookleftarrow a_0$
end if

Strong and weak updates

Points-to set of a_1 depends on whether x_1 points-to b_1

$*x_1 = u_1$

$a_1 = \chi(a_0)$

$b_1 = \chi(b_0)$

update a_1 as →

```
if x1 points-to a
then
     a1 ← u1
end if

if x1 points-to b
then
     a1 ← a0
end if
```

Strong and weak updates

Points-to set of $\mathbf{a_1}$ depends on whether x_1 points-to $\mathbf{b_1}$

$$*x_1 = u_1$$

$$a_1 = \chi(a_0)$$

$$b_1 = \chi(b_0)$$

Connect $\mathbf{a_1}$ and $\mathbf{b_1}$

update $\mathbf{a_1}$ *as*

```
if x₁ points-to a
then
     a₁ ← u₁
end if

if x₁ points-to b
then
     a₁ ← a₀
end if
```

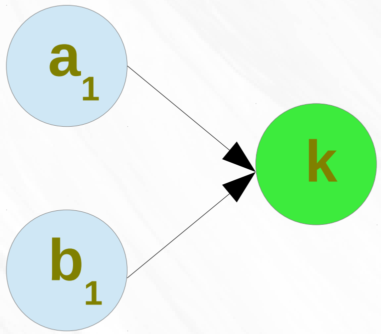
Strong and weak updates

$$*x_1 = u_1$$

$$a_1 = \chi(a_0)$$

$$b_1 = \chi(b_0)$$

Solution: Connect all variables on the LHS of χ, within a *store*. Use a *klique* node as the common connection.



klique node $\overset{\text{1-1}}{\Leftrightarrow}$ store constraint

Strong and weak updates

$*x_1 = u_1$

$a_1 = \chi(a_0)$

$b_1 = \chi(b_0)$

update a_1 *as*

```
if x₁ points-to a
then
    a₁ ↵ u₁
end if
```

```
if x₁ points-to b
then
    a₁ ↵ a₀
end if
```

Strong and weak updates

$$*x_1 = u_1$$

$$a_1 = \chi(a_0)$$

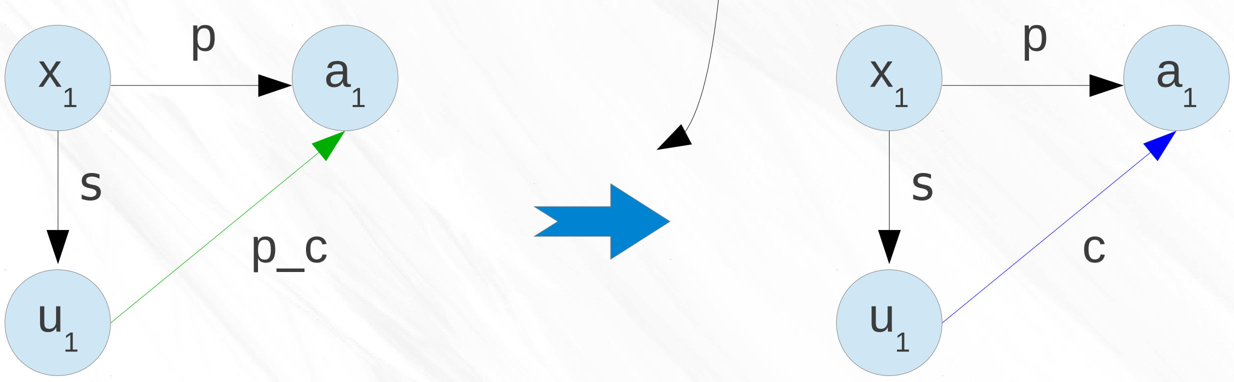
$$b_1 = \chi(b_0)$$

update a_1 *as*

```
if x_1 points-to a
then
     a_1 ← u_1
end if

if x_1 points-to b
then
     a_1 ← a_0
end if
```

Rewrite rule to update a_1:



Strong and weak updates

$$*x_1 = u_1$$
$$a_1 = \chi(a_0)$$
$$b_1 = \chi(b_0)$$

update a_1 *as*

```
if x1 points-to a
then
    a1 ← u1
end if

if x1 points-to b
then
    a1 ← a0
end if
```

Strong and weak updates

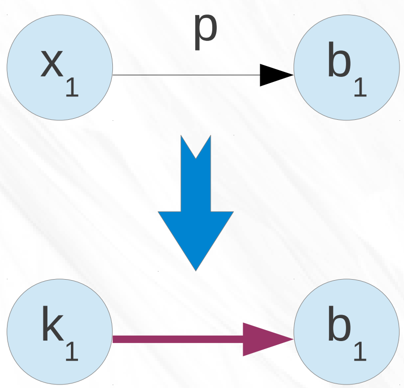
$$*x_1 = u_1$$
$$a_1 = \chi(a_0)$$
$$b_1 = \chi(b_0)$$

update a_1 *as*

```
if x1 points-to a
then
     a1 ← u1
end if
```

```
if x1 points-to b
then
     a1 ← a0
end if
```

Rewrite rule to update a_1: (2 steps)



Note: The rewrite rule shown here is not complete.

Strong and weak updates

$*x_1 = u_1$

$a_1 = \chi(a_0)$

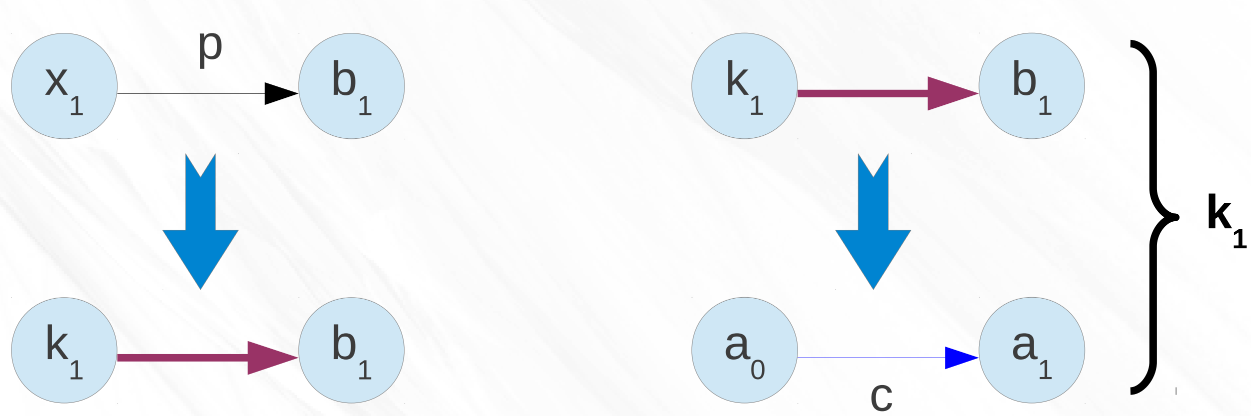
$b_1 = \chi(b_0)$

update a_1 *as*

```
if x1 points-to a
then
     a1 ← u1
end if

if x1 points-to b
then
     a1 ← a0
end if
```

Rewrite rule to update a_1: (2 steps)



Note: The rewrite rule shown here is not complete.

Strong and weak updates

$*x_1 = u_1$

$a_1 = \chi(a_0)$

$b_1 = \chi(b_0)$

update a_1 *as*

```
if x_1 points-to a
then
     a_1 ← u_1
end if

if x_1 points-to b
then
     a_1 ← a_0
end if
```

Handled

Applying the rewrite rules

- Rewrite rules can be applied in **any order**, and to **any node**
- Rewrite rules are applied until fixed point
- At any time, there may be multiple nodes ready for rewrite rule application, allowing **parallel** application of rewrite rules

Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- **Implementation and Results**
- Conclusion

Implementation details

- Intel threading building blocks (TBB) used to manage parallel workload
- A dual-worklist based approach to keep track of nodes for processing
- A hash table based concurrent data structure (`concurrent_unordered_set`), provided by TBB used to represent graph edges

Machine configuration

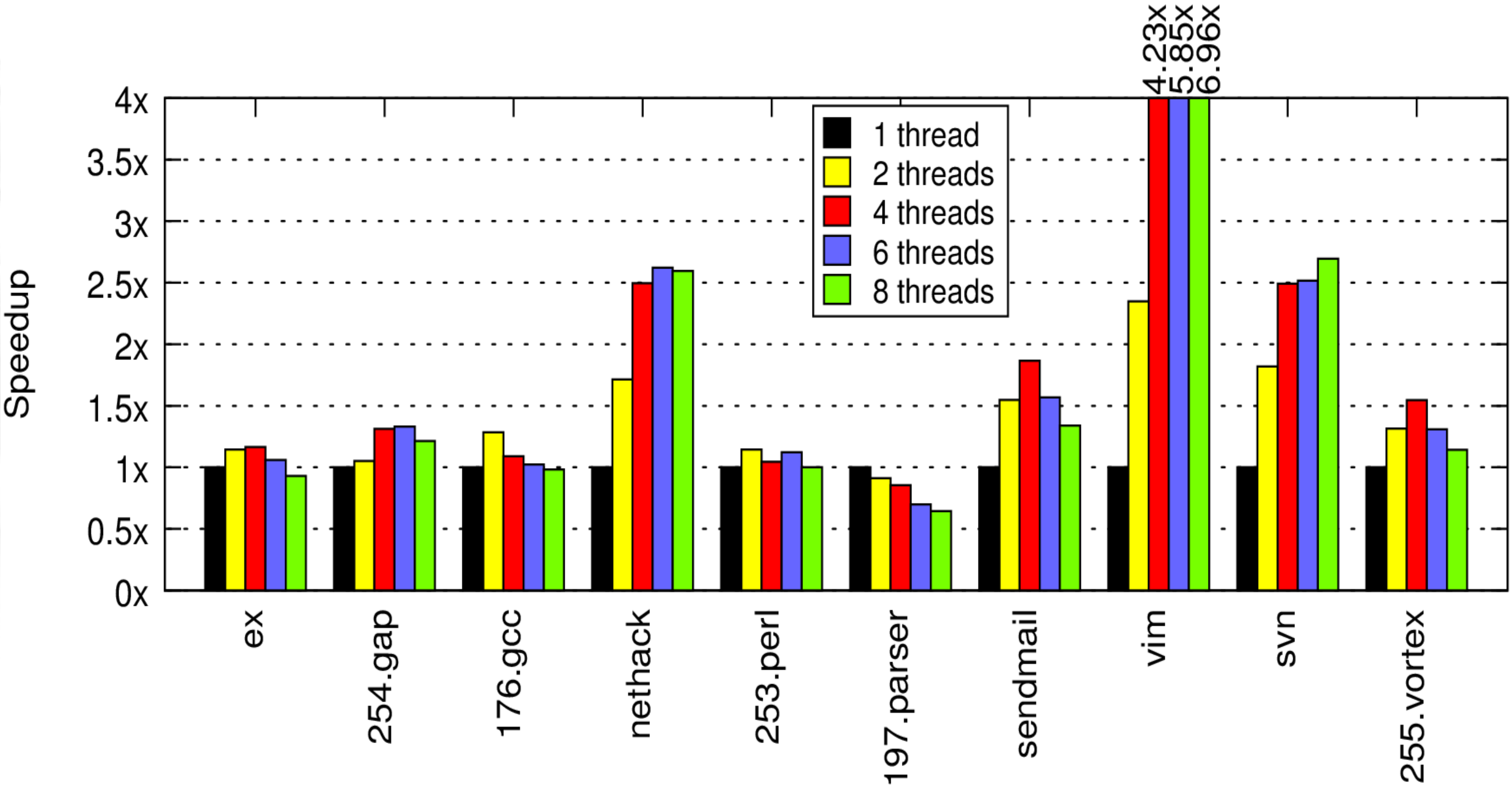
- 4-socket machine
- 2.0 GHz, 8-core processor on each socket
- 64GB memory
- Debian GNU/Linux 6.0
- Intel Threading Building Blocks – 4.0

Benchmarks

Benchmark	Number of graph nodes
Larger (lines of code) programs from SPEC2006	13k - 414k
Ex – text processor	19k
Nethack – text based game	222k
Sendmail – email server	71k
SVN – revision control system	6439k
Vim – text editor	1265k

All of these have been used in previous pointer analysis experiments

Scaling


Speedup
4x
3.5x
3x
2.5x
2x
1.5x
1x
0.5x
0x
4.23x
5.85x
6.96x
1 thread
2 threads
4 threads
6 threads
8 threads
ex
254.gap
176.gcc
nethack
253.perl
197.parser
sendmail
vim
svn
255.vortex

Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion

Conclusion

- First parallelization of precise flow-sensitive pointer analysis
- Flow-sensitive pointer analysis as a graph-rewriting problem – easy to take advantage of amorphous data parallelism
- Scaling of up to 6.9x, for 8 threads shown

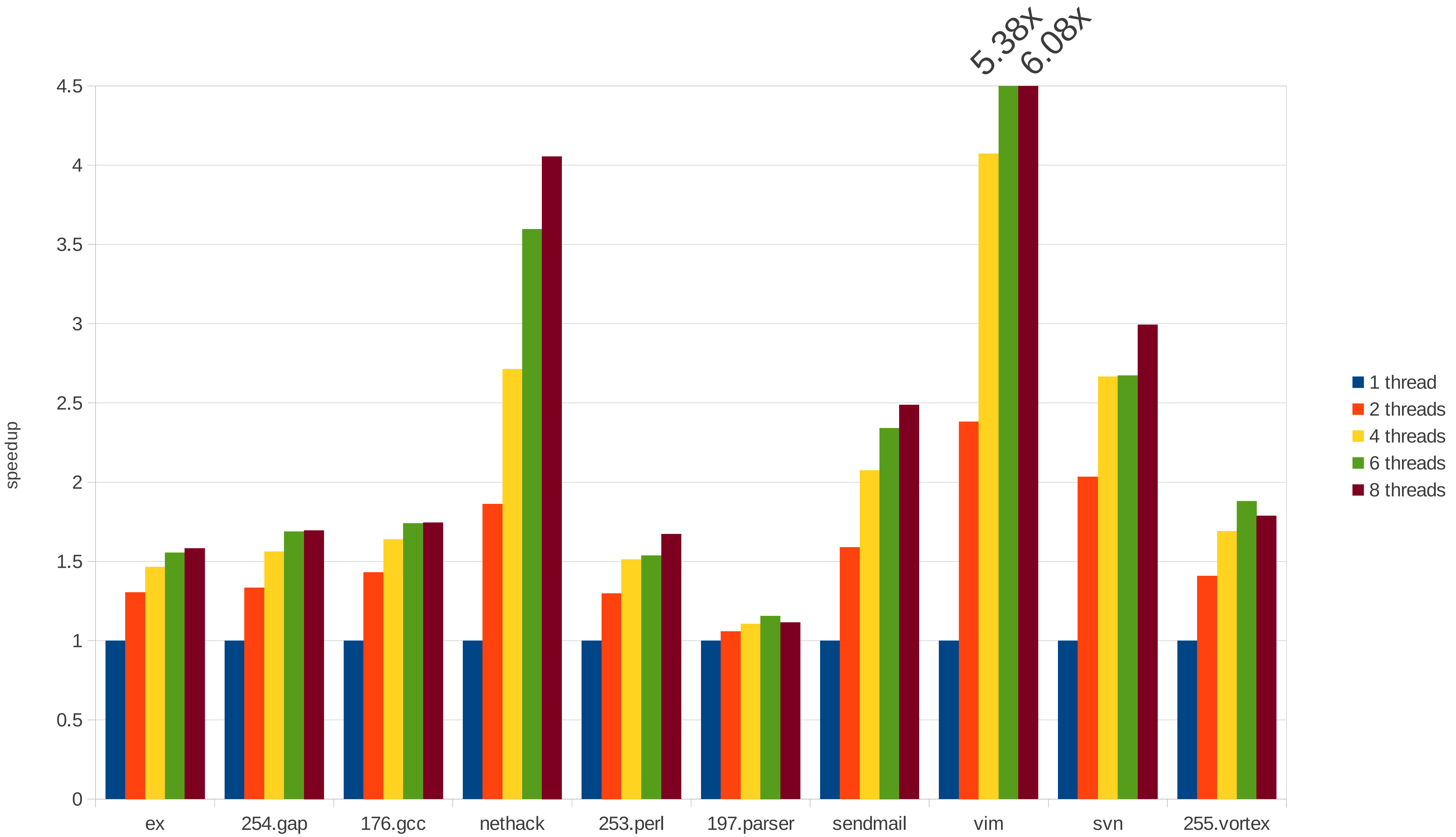
Acknowledgement

- PACT student travel grant
- GARP student grant - IISc
- Ben Hardekopf and Mario Mendez-Lojo
- Rupesh Nasre
- HPC lab members – SERC – IISc

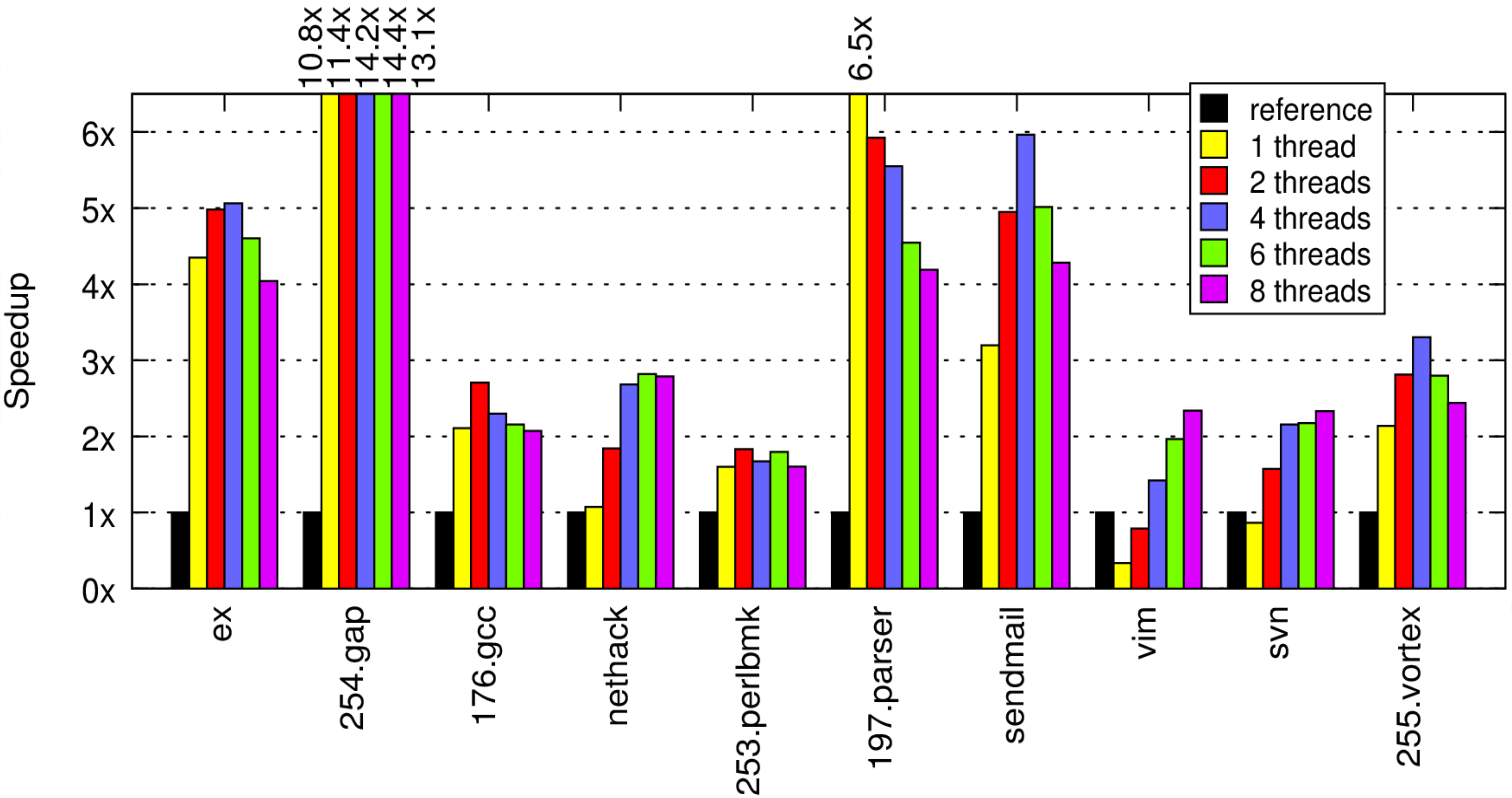
Thank you

Questions?

Backup: recent results – combine worklists



Backup: – Comparison with SFS



Backup: Program size

TABLE I: Number of nodes of each type

benchmark	TopLvl	NonSSA	AddrTakenSSA	Klique
ex	9852	1229	7953	148
254.gap	45946	2393	635639	469
176.gcc	114994	5908	255774	1770
nethack	85994	11977	124448	223
197.parser	8514	1020	3631	129
253.perlbmk	50538	2829	270833	543
sendmail	45155	4136	22220	347
svn	99181	8740	6328901	2738
vim	238031	8935	1017678	724
255.vortex	17910	3304	12138	107

Backup: – Average worklist size

TABLE I: Average number of nodes in each worklist

benchmark	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
ex	86	12	11	11	13	11	16	6
254.gap	335	23	20	20	23	20	5	27
176.gcc	220	76	27	27	76	27	23	72
nethack	1147	270	23	23	275	23	278	956
197.parser	57	8	16	16	9	16	4	13
253.perlbmk	106	15	9	9	16	9	10	14
sendmail	225	31	17	17	32	17	31	22
svn	713	180	82	82	180	82	315	26
vim	4638	81	34	34	85	34	625	66
255.vortex	260	40	10	10	40	10	151	64

(a) .. (h) are worklists for each rewrite rule

Backup: - Future work

- Explore different orders of applying rewrite rules
- Reordering nodes for locality might give better scaling
- Extend for context-sensitivity
- Using concurrent sparse-bit-vectors for representing edges may improve performance