Zygotes

Tyler Caraza-Harter

Outline

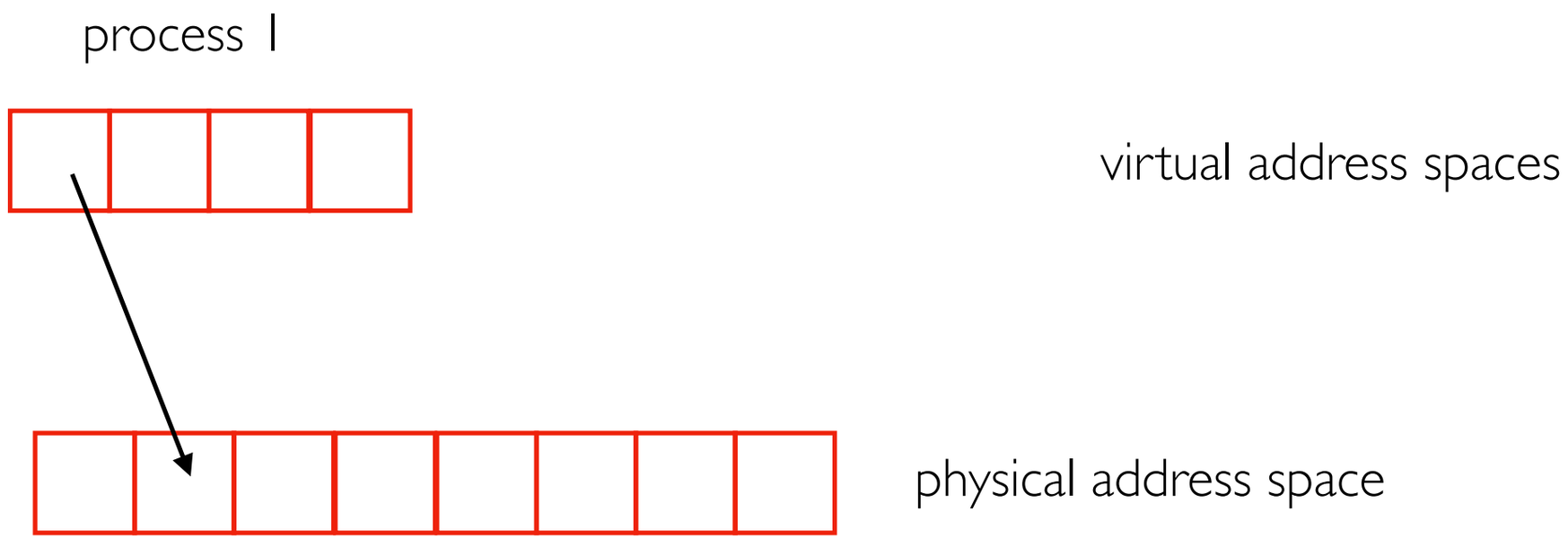
Zygote Background

Zygote Limitations

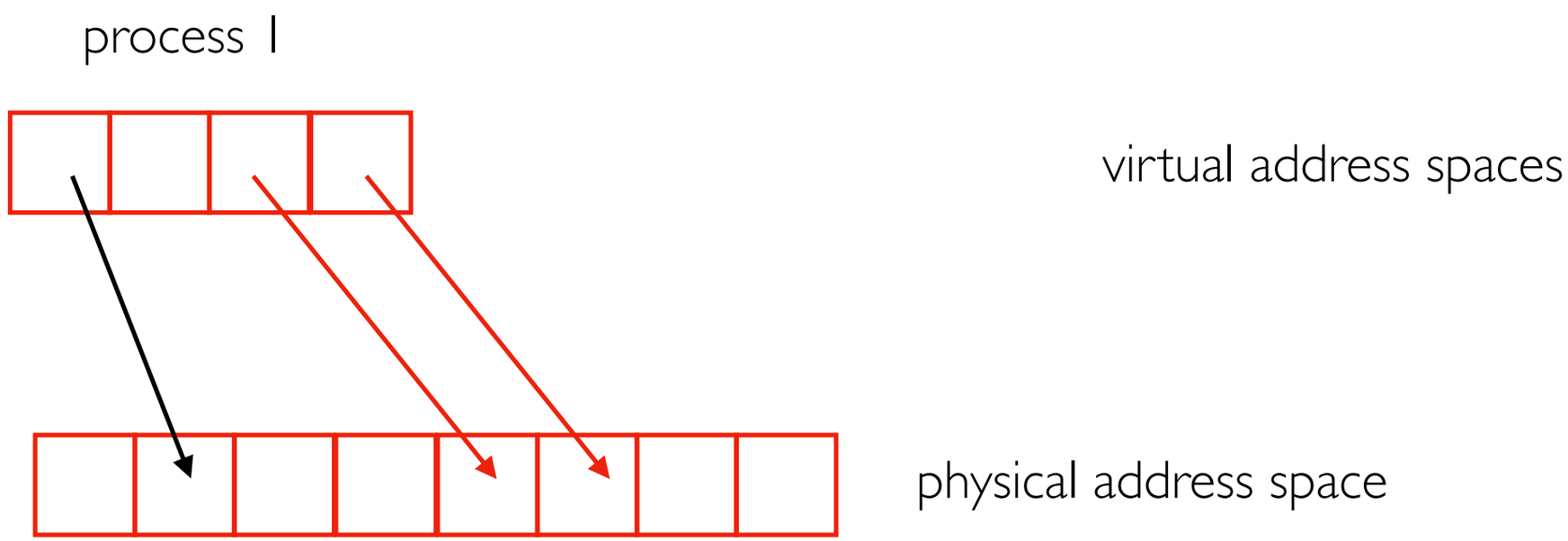
Sandboxed Zygotes

OpenLambda Zygotes: Policy

Single-Threaded Process

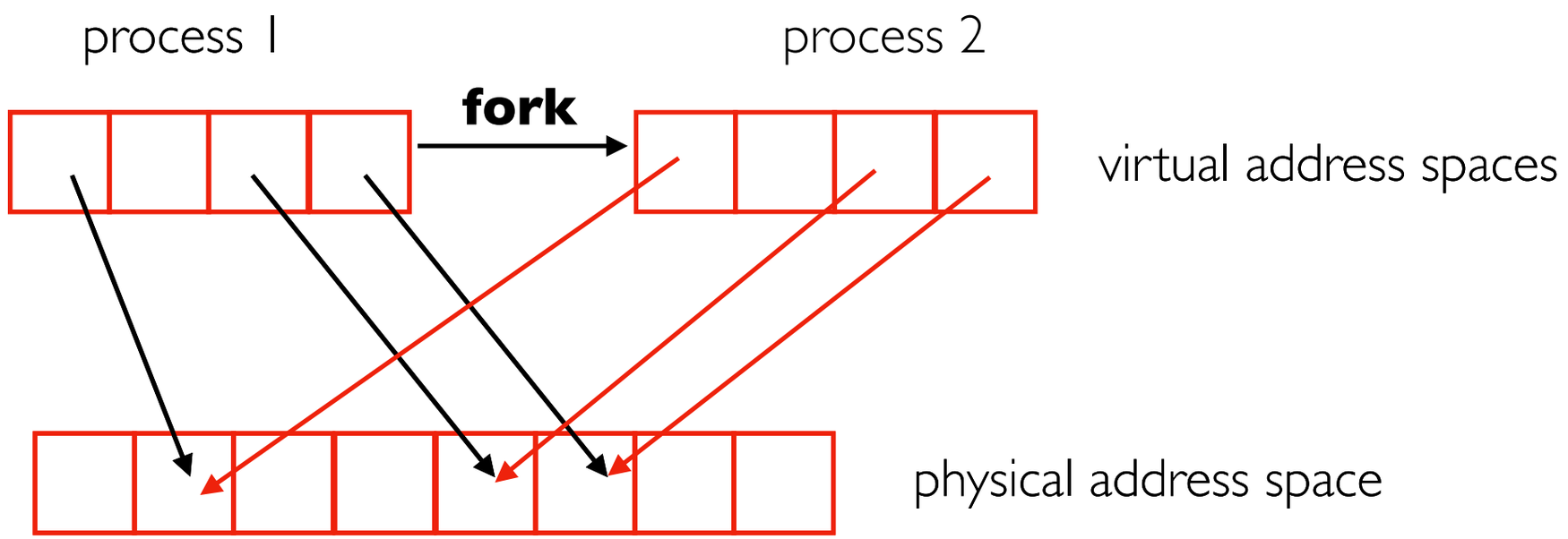


Single-Threaded Process



do some initialization (e.g., importing a module):
uses time and space

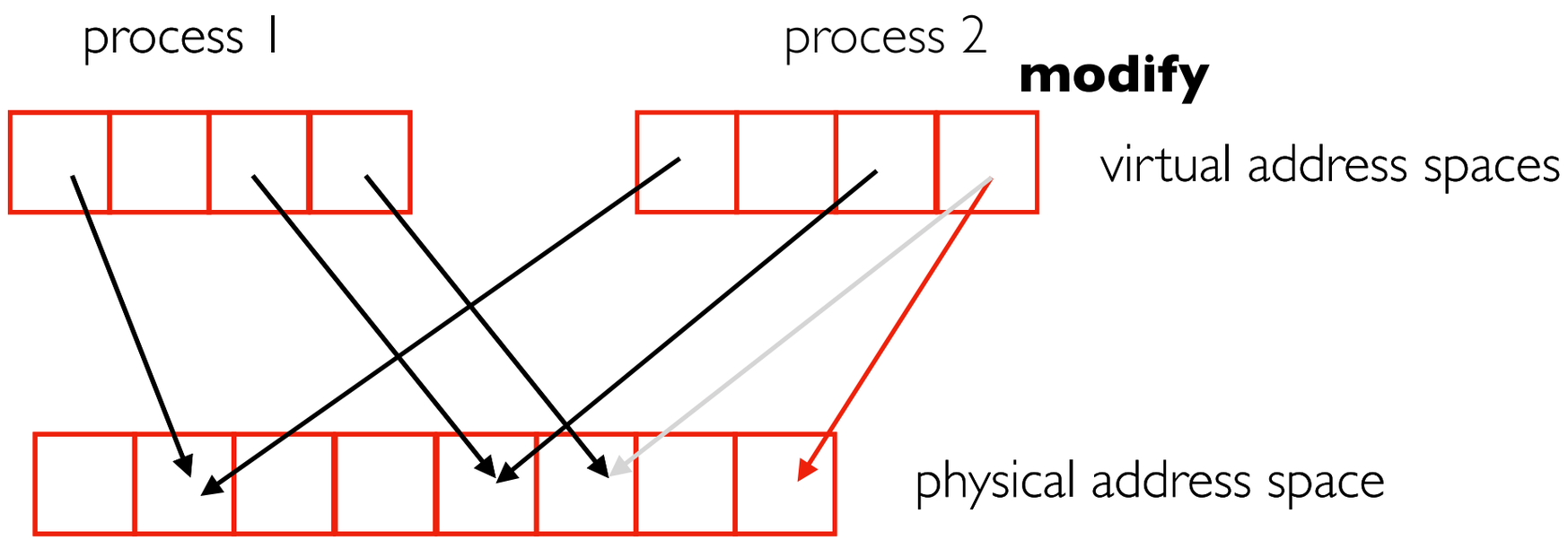
Single-Threaded Process



`fork()` creates a new process.
(note: fork is a wrapper around clone, which takes more args)

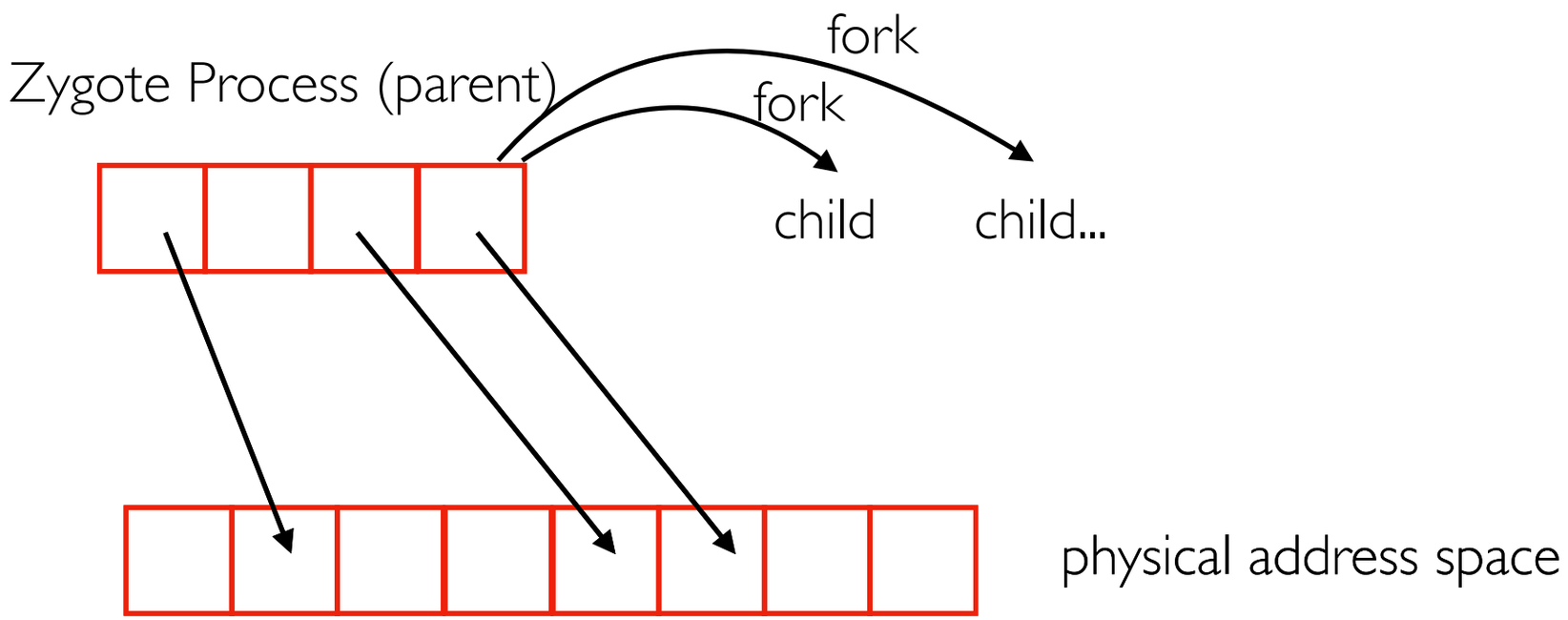
The two processes (parent and child) with have (initially)
identical address spaces, backed by the same physical memory

Single-Threaded Process



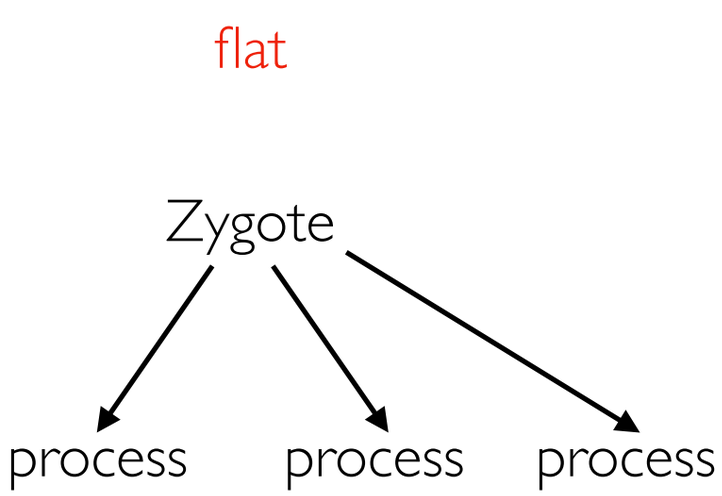
copy on write (COW):
if either of the processed write a page, a copy is made (so as not to interfere with the other process)

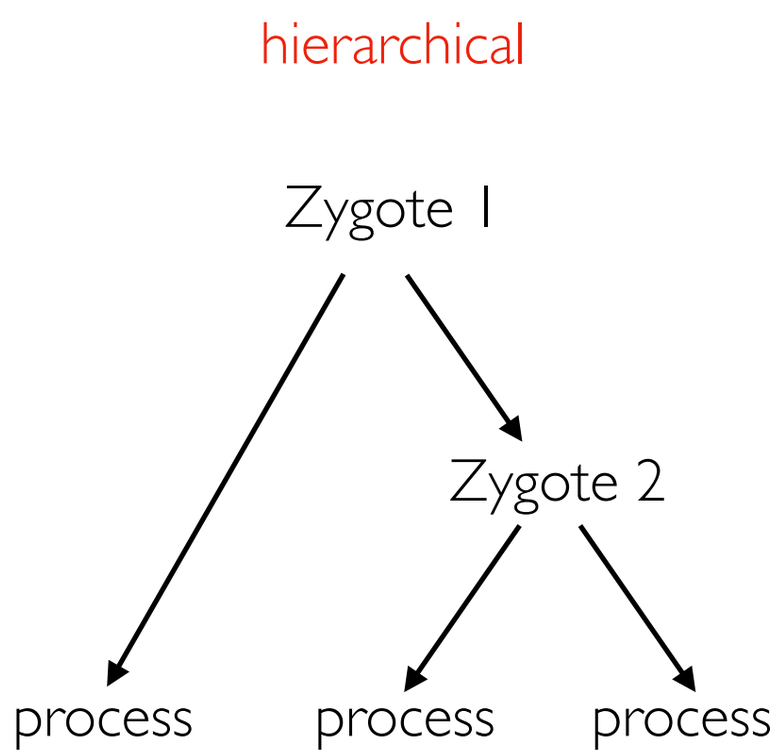
Zygotes



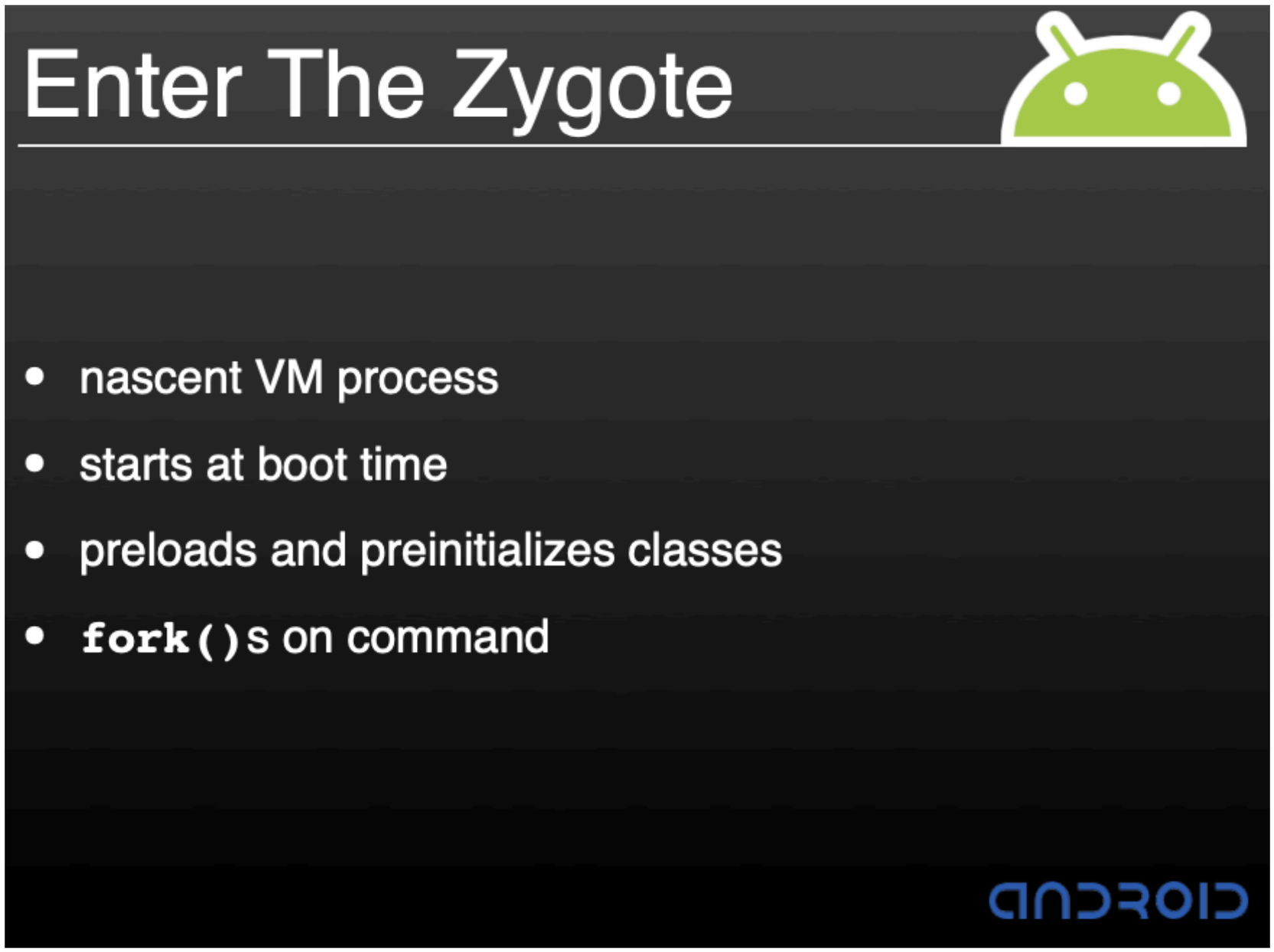
Zygote **definition:** a pre-initialized process that exists soley for the purpose of efficiently (with respect to space and time) forking new processes

Flat vs. Hierarchical





Zygotes in Android (first use?)



slide from 2008 Google I/O talk: Dalvik VM Internals

https://sites.google.com/site/io/dalvik-vm-internals

Outline

Zygote Background

Zygote Limitations

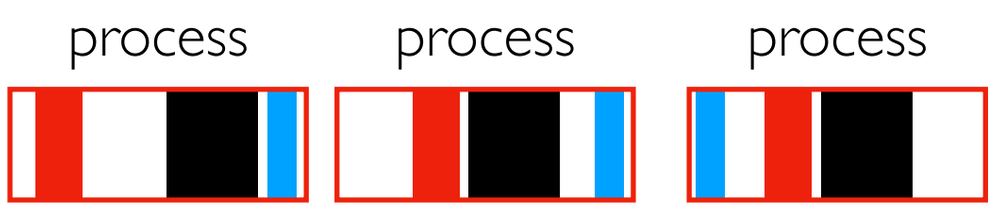
Sandboxed Zygotes

OpenLambda Zygotes: Policy

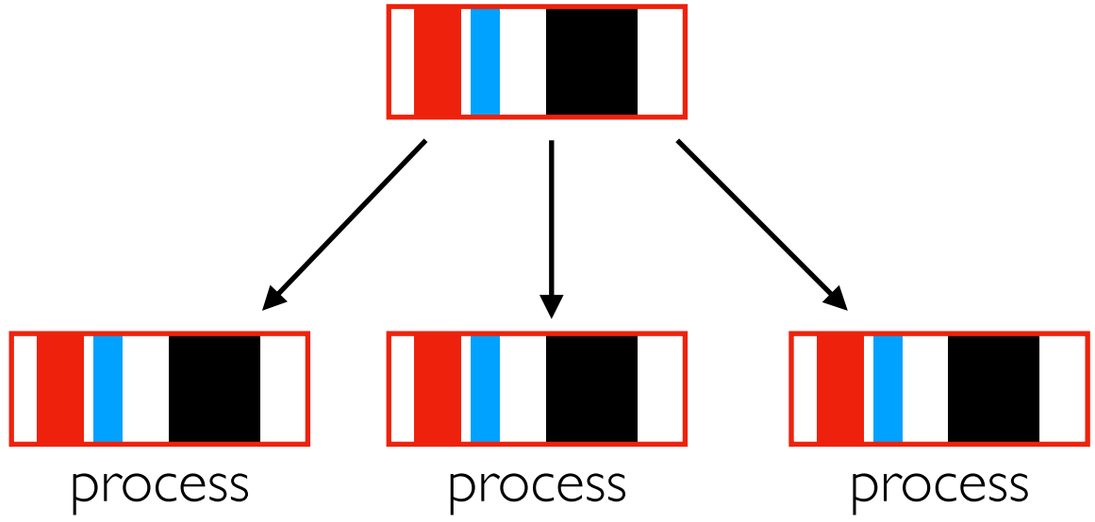
Issue 1: Address Space Randomization

Exploiting security bugs is easier if you know where things are (e.g., injecting a function addr as part of a buffer overflow).

Address space randomization helps.



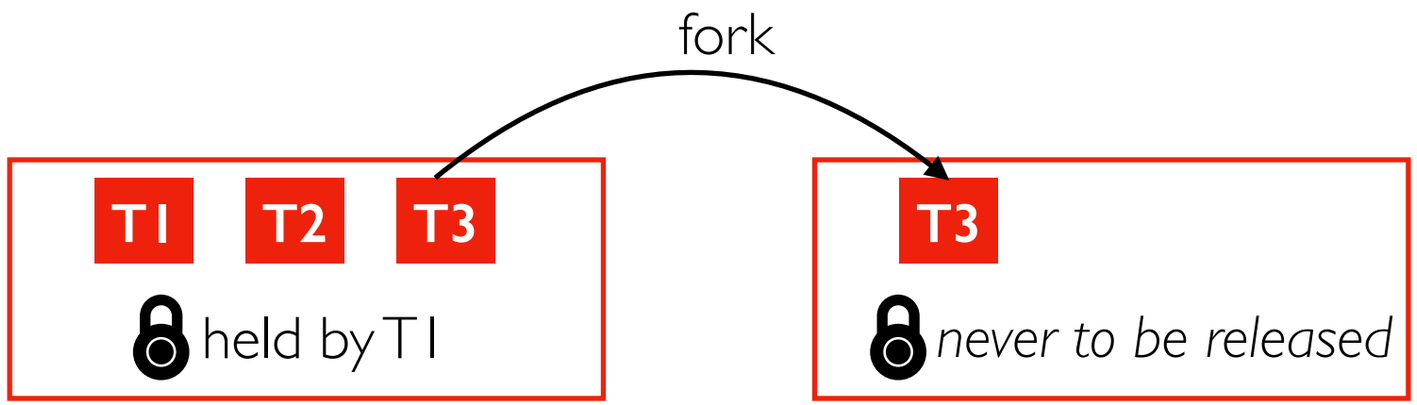
randomized layouts without Zygotes



Zygotes defeat randomization

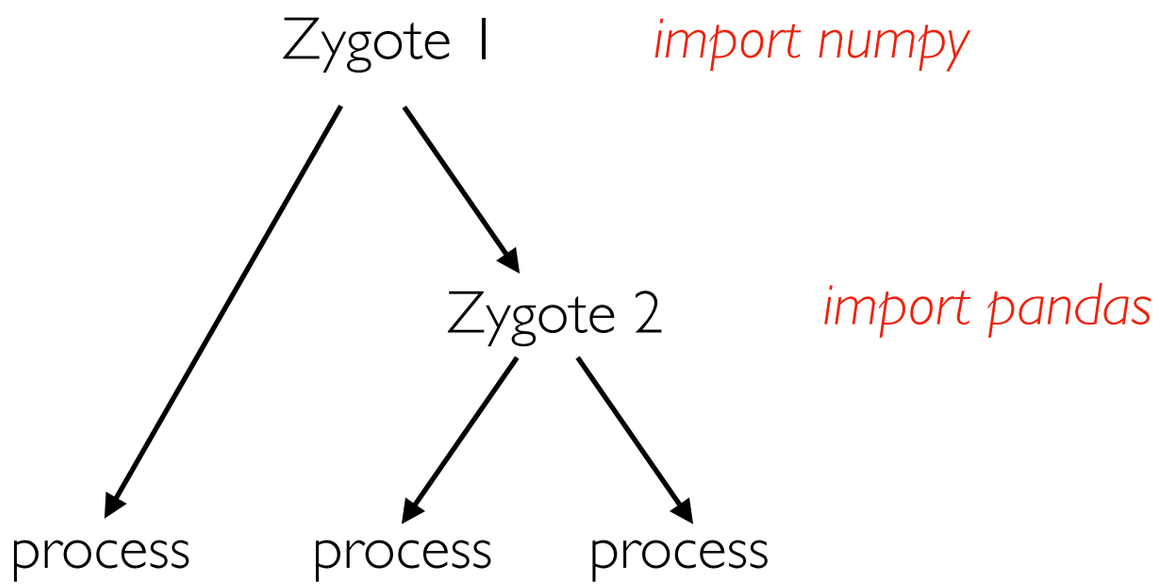
Issue 2: Threads

When a thread calls fork, only that thread goes. This can be problematic by itself, but is worse if locks are involved.



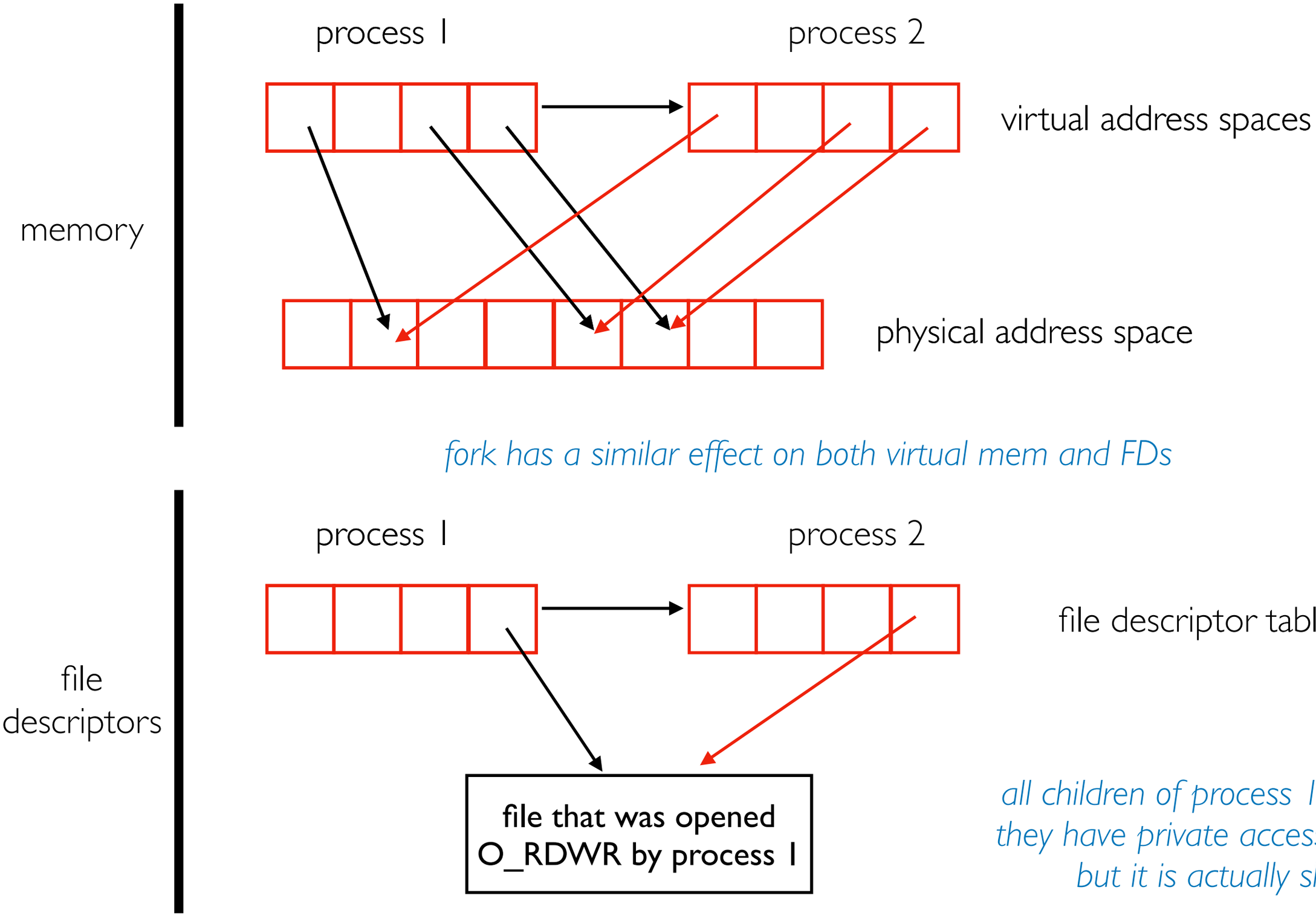
Issue 3: Incremental Init

Hierarchical Zygotes work when you can incrementally init more things mid-execution.



This wouldn't work if you're statically linking in all libraries (like in Rust or Go).

Issue 4: Unintentionally Shared Resources

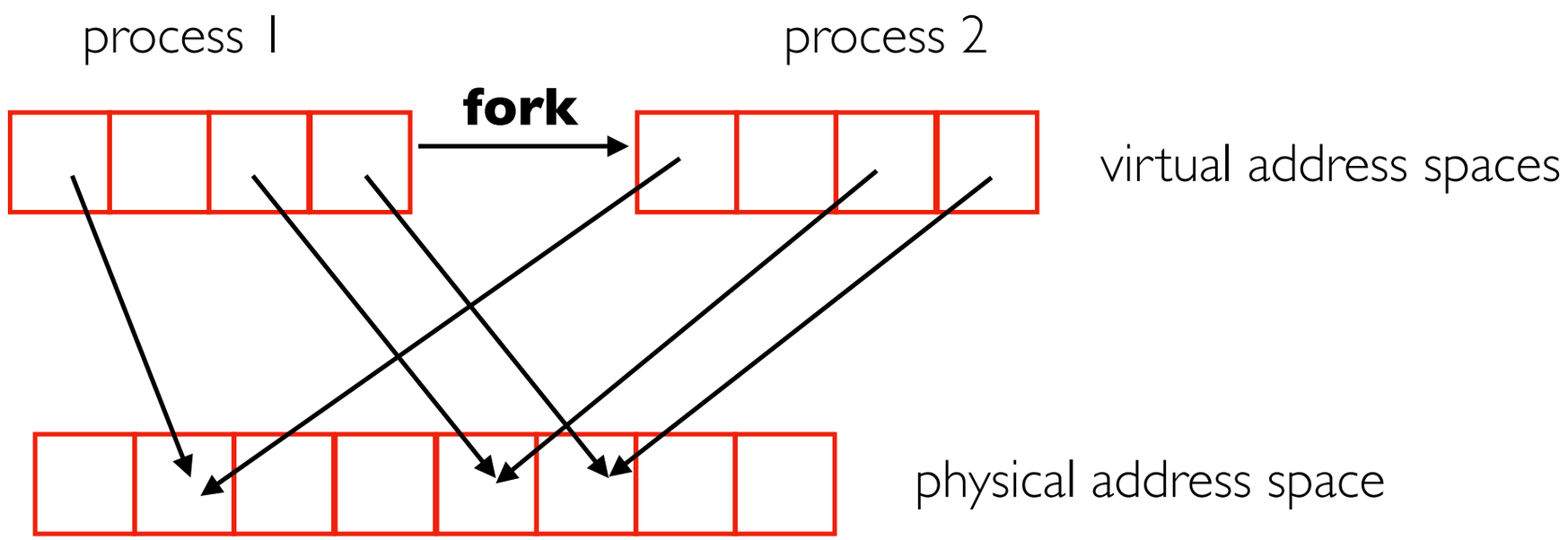


fork has a similar effect on both virtual mem and FDs

all children of process 1 may think they have private access to the file, but it is actually shared

worst case: keeping a dir FD open could let a child escape with chroot!

Issue 5: Accounting



We're using 12 KB (3 pages) of physical memory total. How should we do accounting across the two processes?

- 6 KB each? If process 2 stops, does the memory accounting for process 1 pop, leading to a kill?
- 12 KB for process 1, since it allocated it first? Why should process 2 have a free ride?

Outline

Zygote Background

Zygote Limitations

Sandboxed Zygotes

OpenLambda Zygotes: Policy

Peeking Behind the Curtains of Serverless Platforms

Liang Wang, *UW-Madison;* Mengyuan Li and Yinqian Zhang, *The Ohio State University;*
Thomas Ristenpart, *Cornell Tech;* Michael Swift, *UW-Madison*

https://www.usenix.org/conference/atc18/presentation/wang-liang

1. AWS Lambda Instance Coldstart Latency >250ms

2. If N instances of a given lambda are idle,
AWS evicts N/2 instances every 5 minutes

Observation: we need to improve cold start latency
and/or make Lambda instances VERY memory efficient

SOCK: Rapid Task Provisioning with Serverless-Optimized Containers

Edward Oakes, Leon Yang, Dennis Zhou, and Kevin Houck, *University of Wisconsin-Madison;* Tyler Harter, *Microsoft, GSL;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin-Madison*

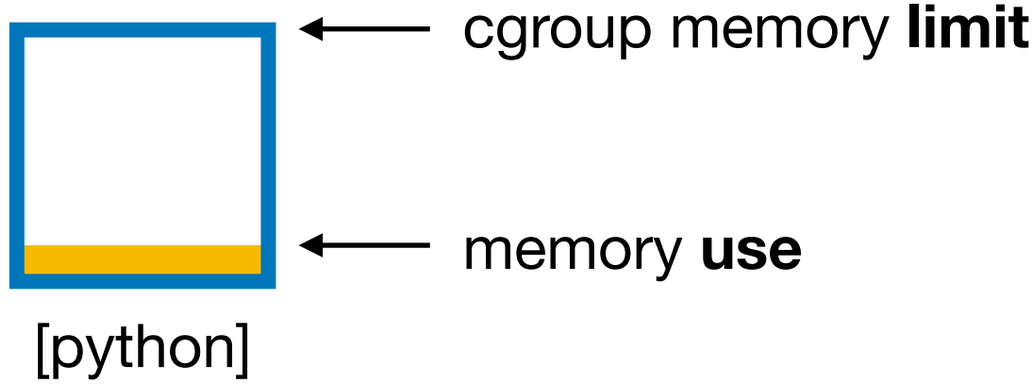
https://www.usenix.org/conference/atc18/presentation/oakes

Key Idea of OpenLambda/SOCK: start Lambdas by forking containerized Zygotes that have already imported certain dependencies (basic Zygote idea used by Android with JVM)

1. Improve startup time (pay import time only once)

2. Improve memory efficiency (copy-on-write sharing in kernel)

Parent Sandbox



Virtual

Physical

very short background on SOCK Zygotes

Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes

Parent Sandbox

Virtual

import pandas

Physical

Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes

Parent Sandbox

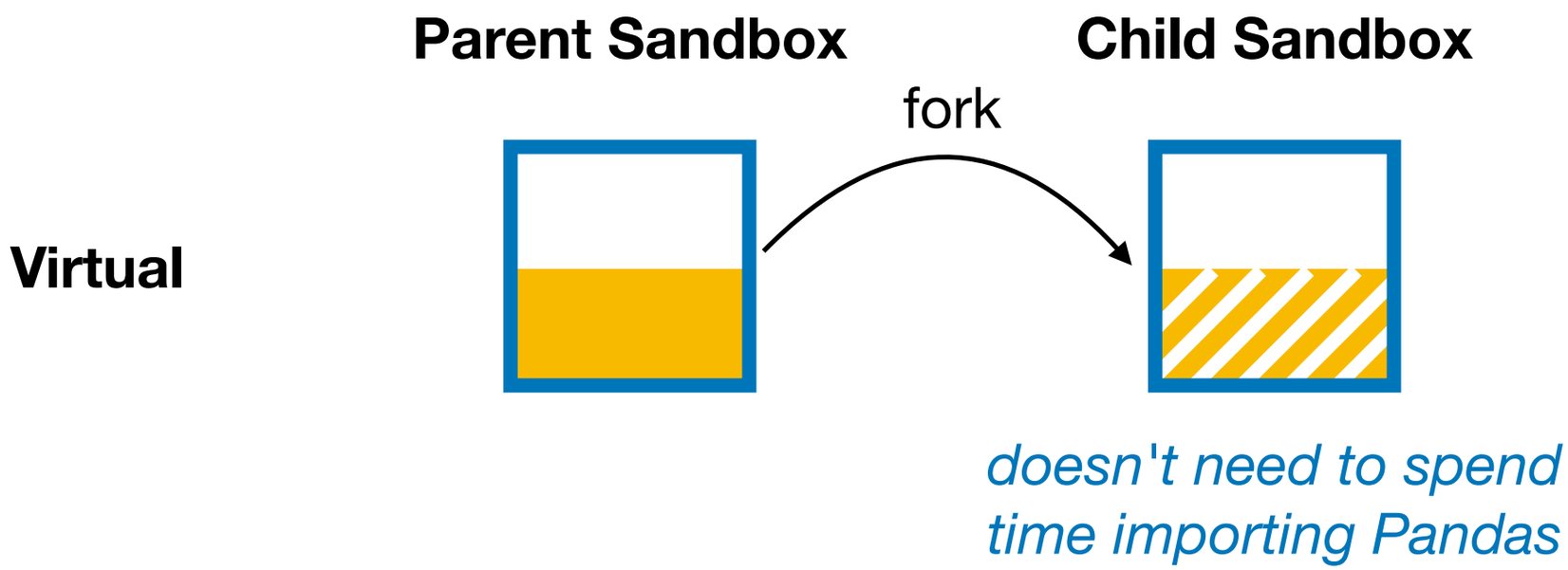
Virtual

import pandas

Physical

Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes



Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes

Parent Sandbox

Child Sandbox

Virtual

cgroup memory **limit**

memory **use**

didn't allocate any more memory

Physical

Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes

Parent Sandbox

Child Sandbox

Virtual

import sklearn

Physical

Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes

Parent Sandbox

Child Sandbox

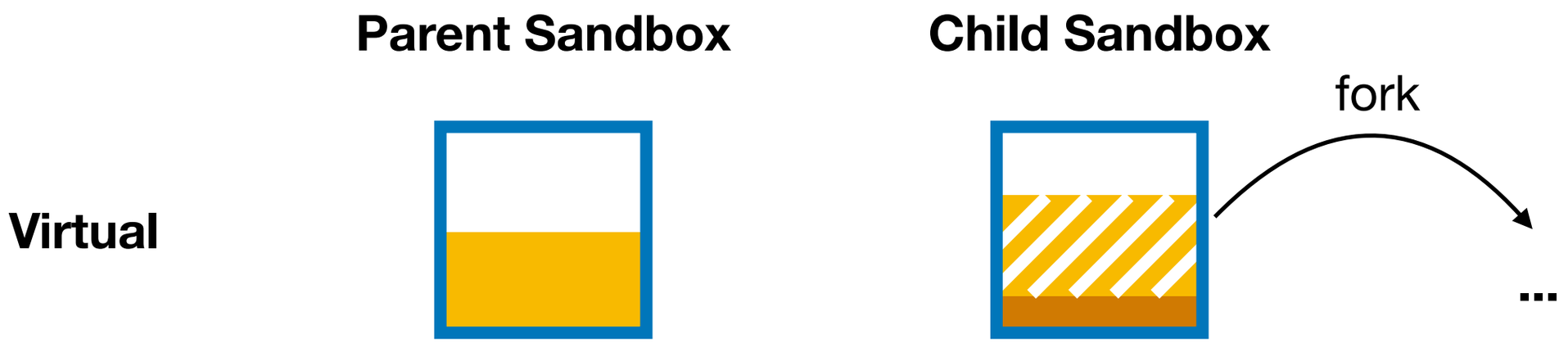
Virtual

import sklearn

Physical

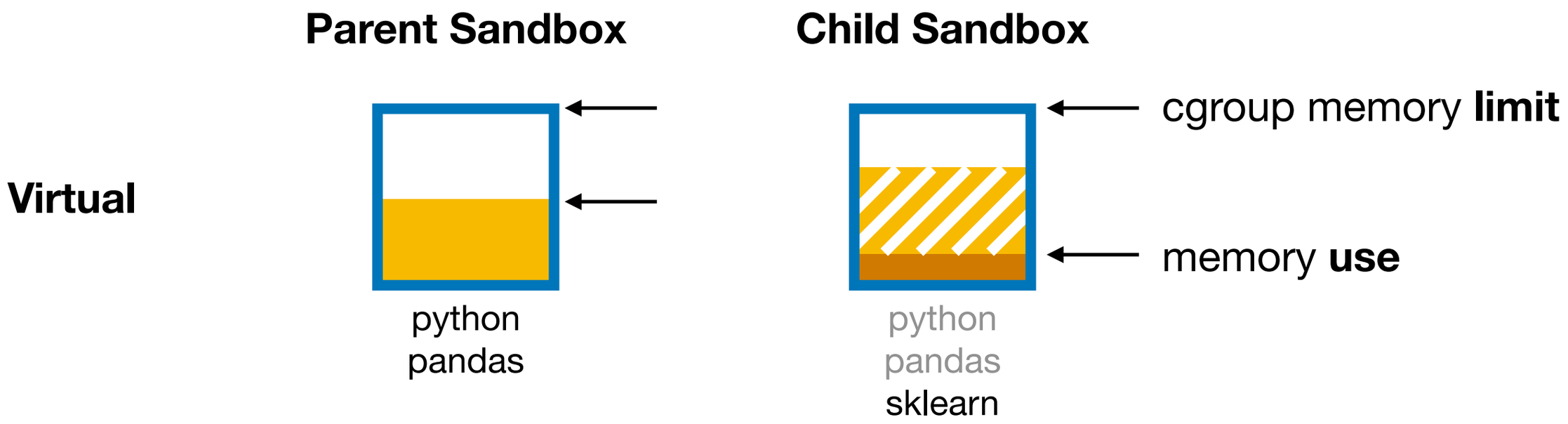
Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes



Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes



Mechanism: SOCK containers support sandbox forking

Policy: import cache maintains pool of "Zygote" containers with various pre-imports, providing parents for new sandboxes

Outline

Zygote Background

Zygote Limitations

Sandboxed Zygotes

OpenLambda Zygotes: Policy

Previous Cache Policy

Policy in SOCK paper was dynamic, but shortsighted:

- If a Zygote has exactly the packages needed by lambda, use it
- Else:
    1. find a Zygote ZA with "greatest" subset of packages
    2. create a Zygote ZB from ZA, adding any lacking packages
    3. use ZB to create Sandbox for lambda

Problem: cannot take advantage of certain patterns obvious across many different lambdas. Pathological case:

- there are N packages, and N*(N-1)/2 lambdas
- each lambda imports a different pair of packages
- a new Zygote will be created each time, but it will never be used more than once!

Inspiration for a better Zygote tree policy

1

Cache-aware load balancing of data center applications

Aaron Archer
Google
New York, New York
aarcher@google.com

Kevin Aydin
Google
New York, New York
kaydin@google.com

MohammadHossein Bateni
Google
New York, New York
bateni@google.com

Vahab Mirrokni
Google
New York, New York
mirrokni@google.com

Aaron Schild
UC Berkeley
Berkeley, California
aschild@berkeley.edu

Ray Yang
Google
New York, New York
rayy@google.com

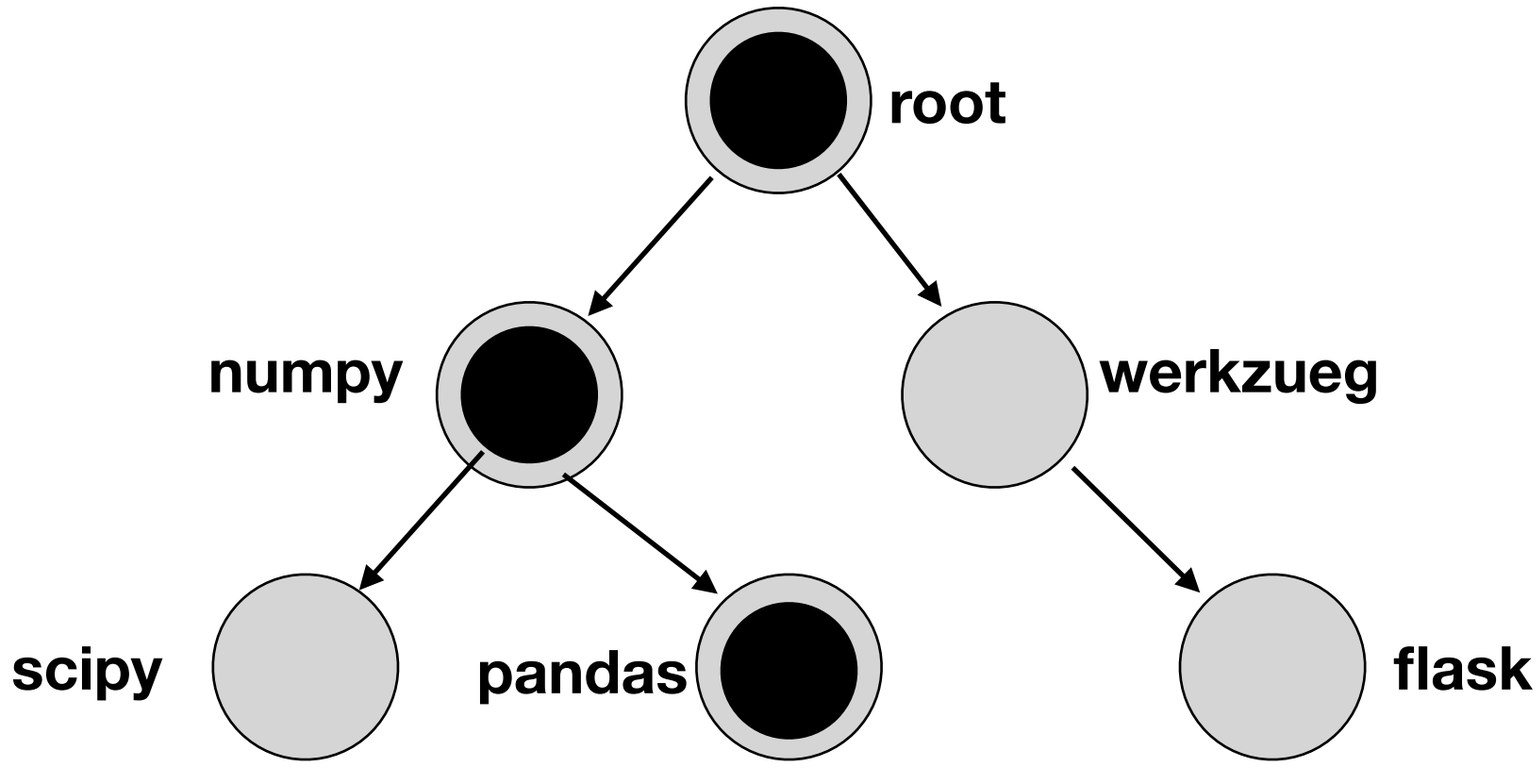
Idea: statically compute the cache structure offline (e.g., from last night's traces); tweak to make it somewhat dynamic

2 **Decision Trees**

Idea: greedily build a tree by starting with one node, specifying a metric for the quality of the tree, and greedily split nodes to improve quality the most

Inspiration for a better Zygote tree policy

1



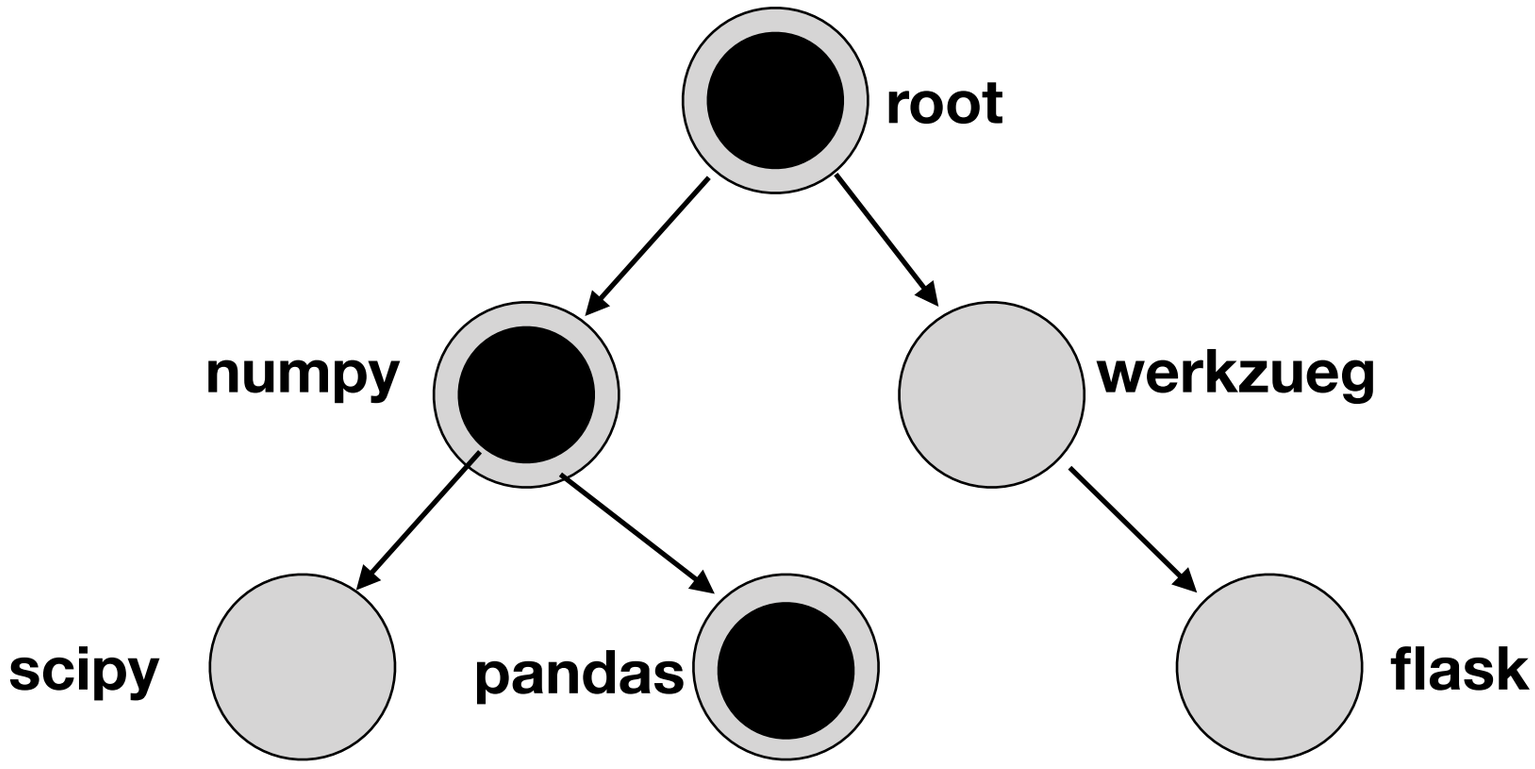
compute gray structure offline, create/evict black Zygotes based on workload

2 **Decision Trees**

Idea: greedily build a tree by starting with one node, specifying a metric for the quality of the tree, and greedily split nodes to improve quality the most

Inspiration for a better Zygote tree policy

1



compute gray structure offline, create/evict black Zygotes based on workload

2 Decision Trees

Idea: greedily build a tree by starting with one node, specifying a metric for the quality of the tree, and greedily split nodes to improve quality the most

Formulating Lambda Trace for Decision Tree

Matrix constructed
from last night's trace

Root Node

	A	B	C	D	Y
fn1	1	1	0	0	**1**
fn2	1	0	1	0	**1**
fn3	1	0	0	1	**1**
fn4	0	1	0	1	**0**
fn5	0	0	1	1	**1**
fn6	0	0	0	1	**0**
fn7	0	1	0	0	**0**
fn8	0	0	0	0	**0**

don't overthink what the "Y"
is that we're trying to predict
(it will go away soon)

Lambda function fn5 imports packages C and D

Formulating Lambda Trace for Decision Tree

Root Node

	A	B	C	D	Y
fn1	1	1	0	0	**1**
fn2	1	0	1	0	**1**
fn3	1	0	0	1	**1**
fn4	0	1	0	1	**0**
fn5	0	0	1	1	**1**
fn6	0	0	0	1	**0**
fn7	0	1	0	0	**0**
fn8	0	0	0	0	**0**

This one-node tree has a high *impurity score(*)*, because there is a 50/50 mix of Y=0 and Y=1 in the same node (the worst case)

*common impurity metrics: Gini, entropy, variance

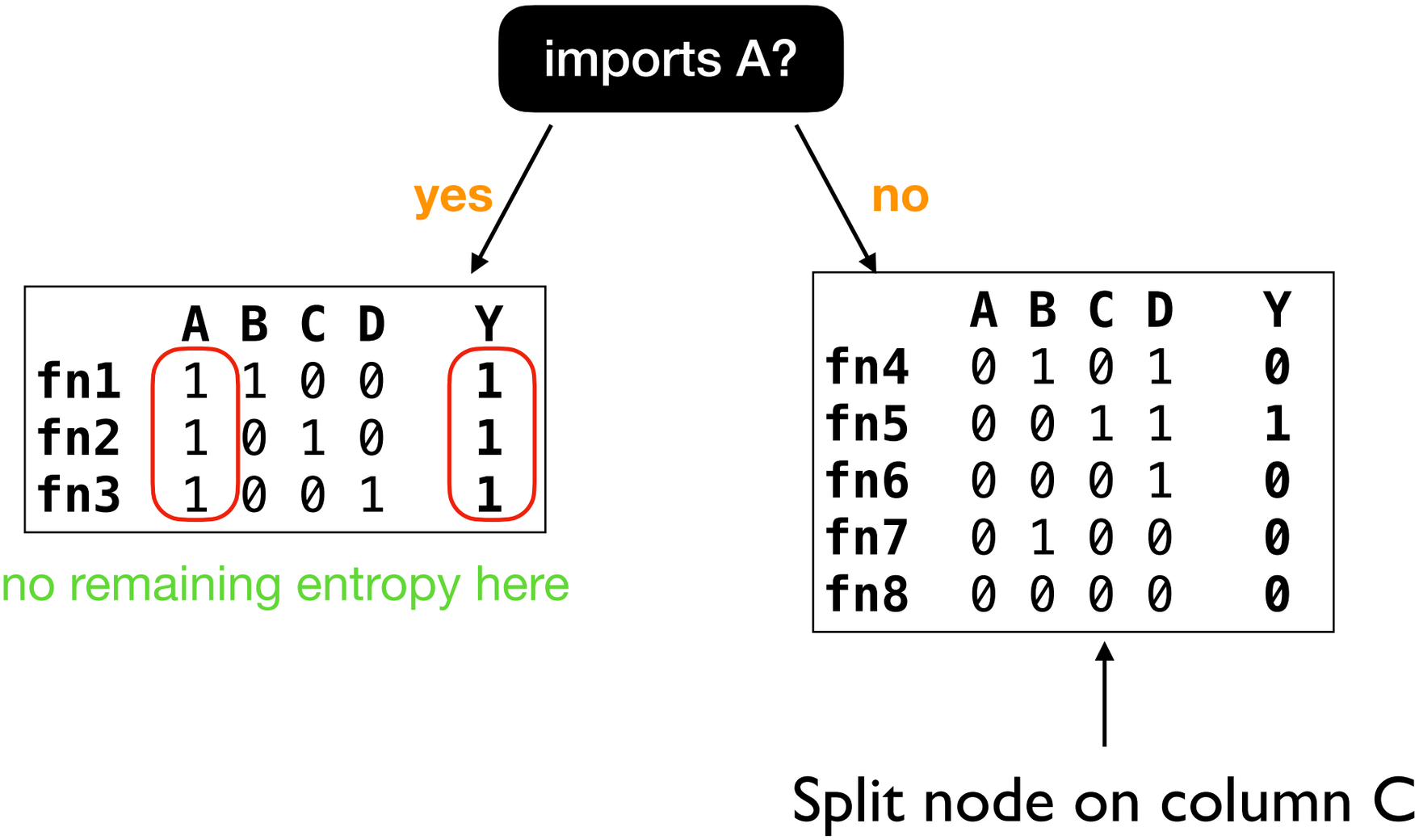
Formulating Lambda Trace for Decision Tree

Root Node

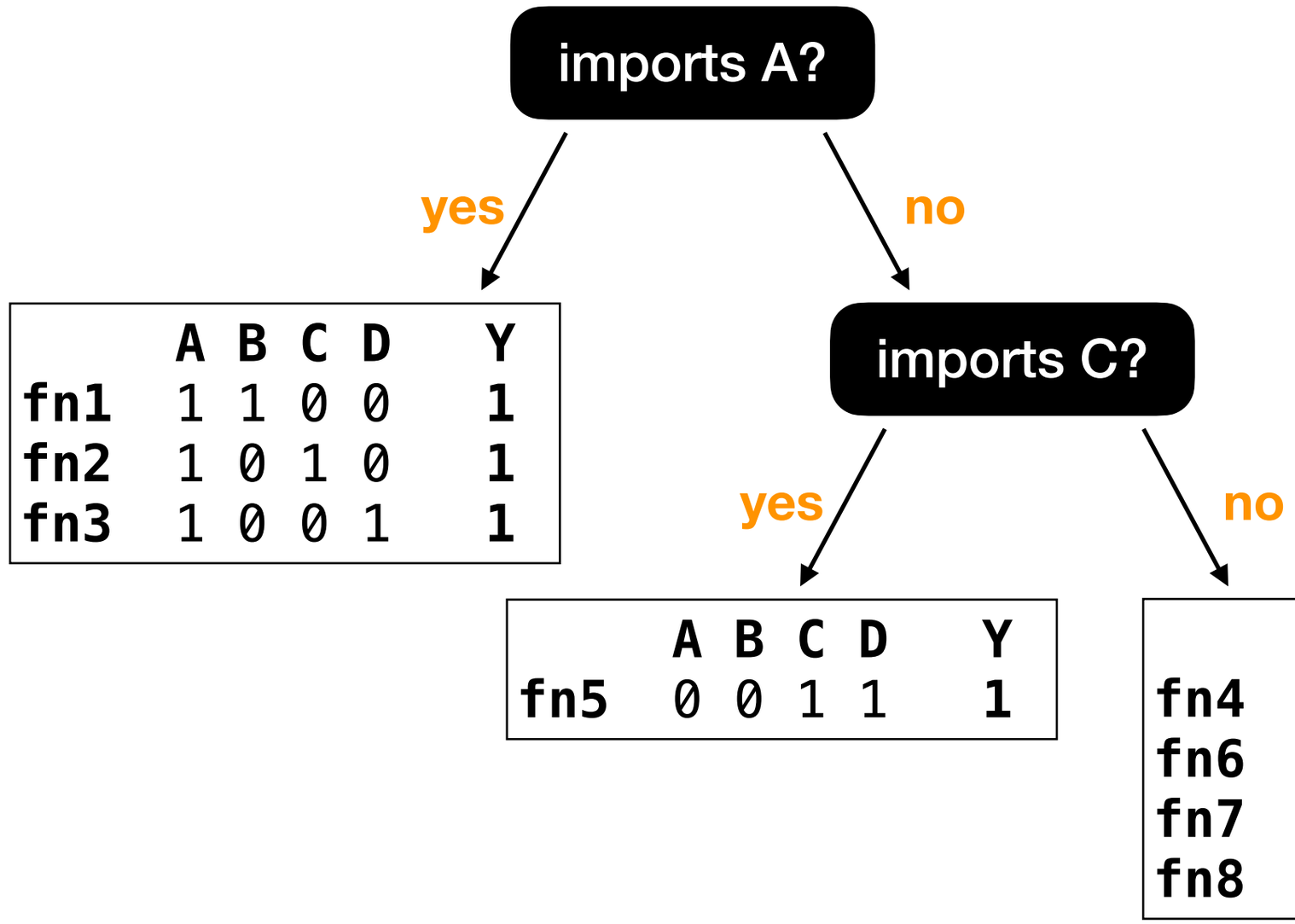
	A	B	C	D	Y
fn1	1	1	0	0	**1**
fn2	1	0	1	0	**1**
fn3	1	0	0	1	**1**
fn4	0	1	0	1	**0**
fn5	0	0	1	1	**1**
fn6	0	0	0	1	**0**
fn7	0	1	0	0	**0**
fn8	0	0	0	0	**0**

Split node on column A in root node,
because that will organize Y the most

Formulating Lambda Trace for Decision Tree



Formulating Lambda Trace for Decision Tree



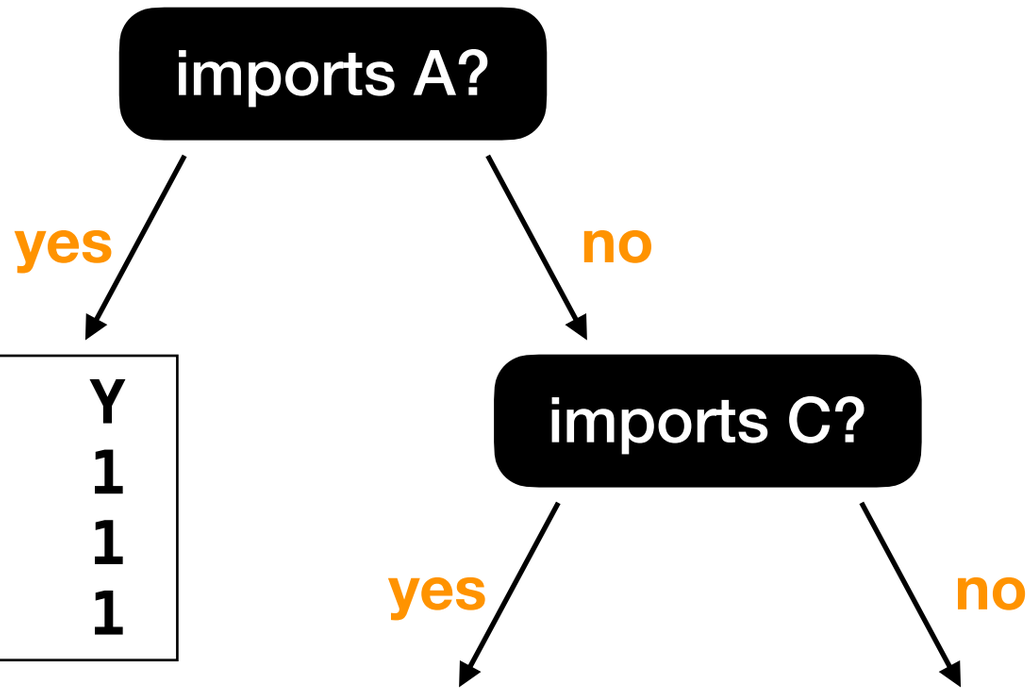
	A	B	C	D	Y
fn1	1	1	0	0	1
fn2	1	0	1	0	1
fn3	1	0	0	1	1

	A	B	C	D	Y
fn5	0	0	1	1	1

	A	B	C	D	Y
fn4	0	1	0	1	0
fn6	0	0	0	1	0
fn7	0	1	0	0	0
fn8	0	0	0	0	0

This perfectly fits the training data!

Formulating Lambda Trace for Decision Tree



	A	B	C	D	Y
fn1	1	1	0	0	1
fn2	1	0	1	0	1
fn3	1	0	0	1	1

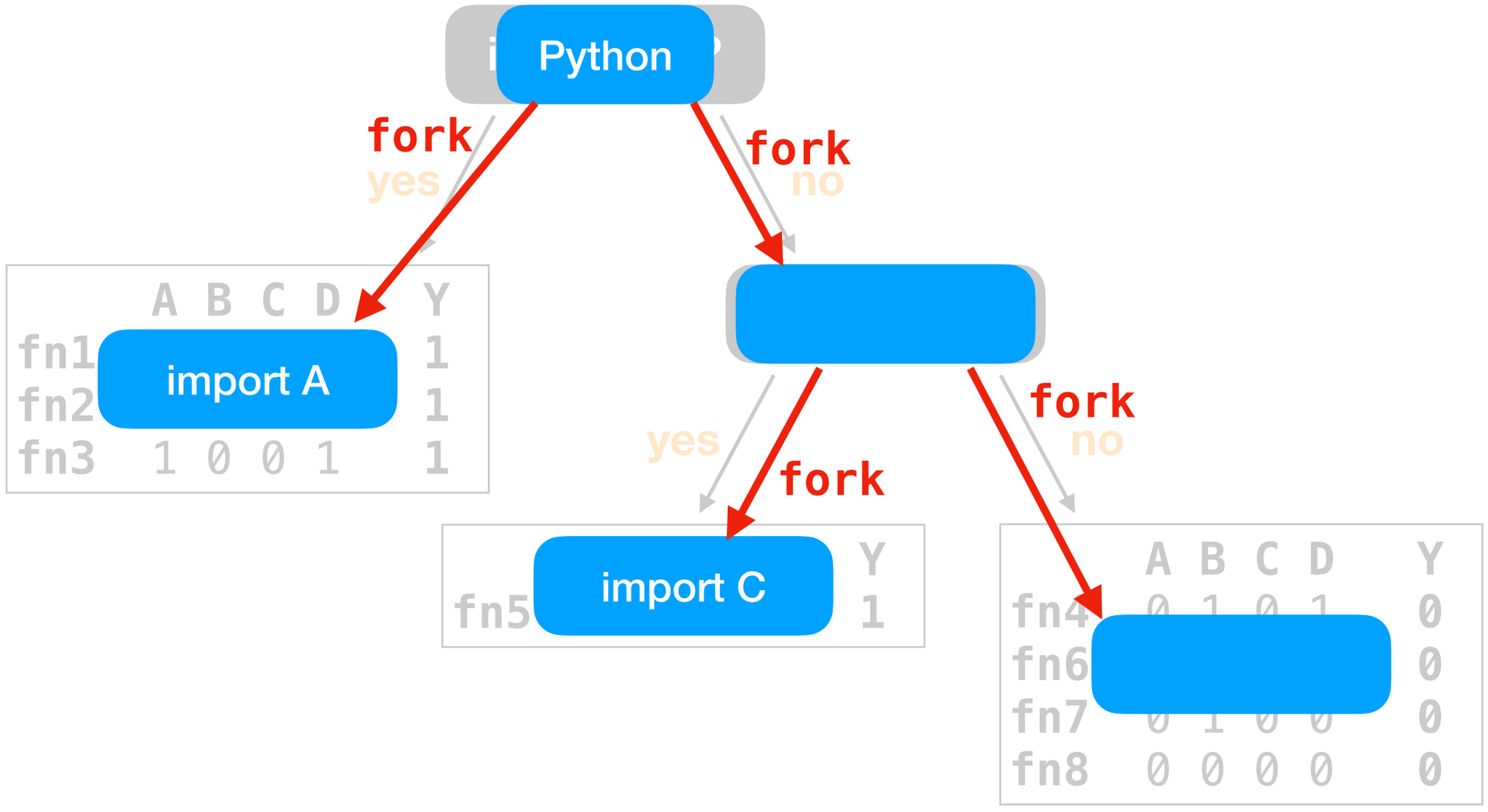
	A	B	C	D	Y
fn5	0	0	1	1	1

	A	B	C	D	Y
fn4	0	1	0	1	0
fn6	0	0	0	1	0
fn7	0	1	0	0	0
fn8	0	0	0	0	0

This perfectly fits the training data!

Note: we *could* also construct a 5-node Zygote tree from this decision tree

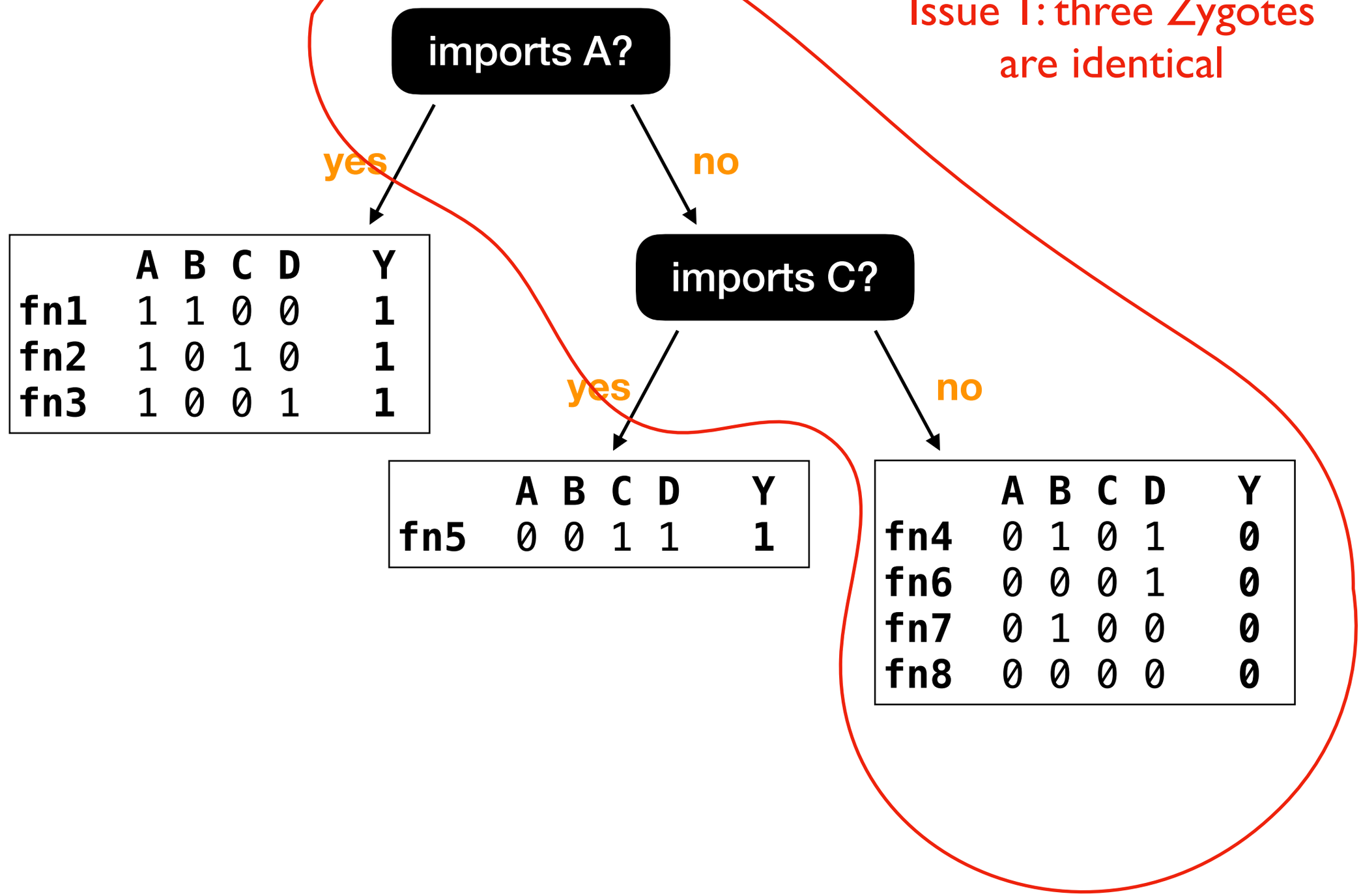
Formulating Lambda Trace for Decision Tree



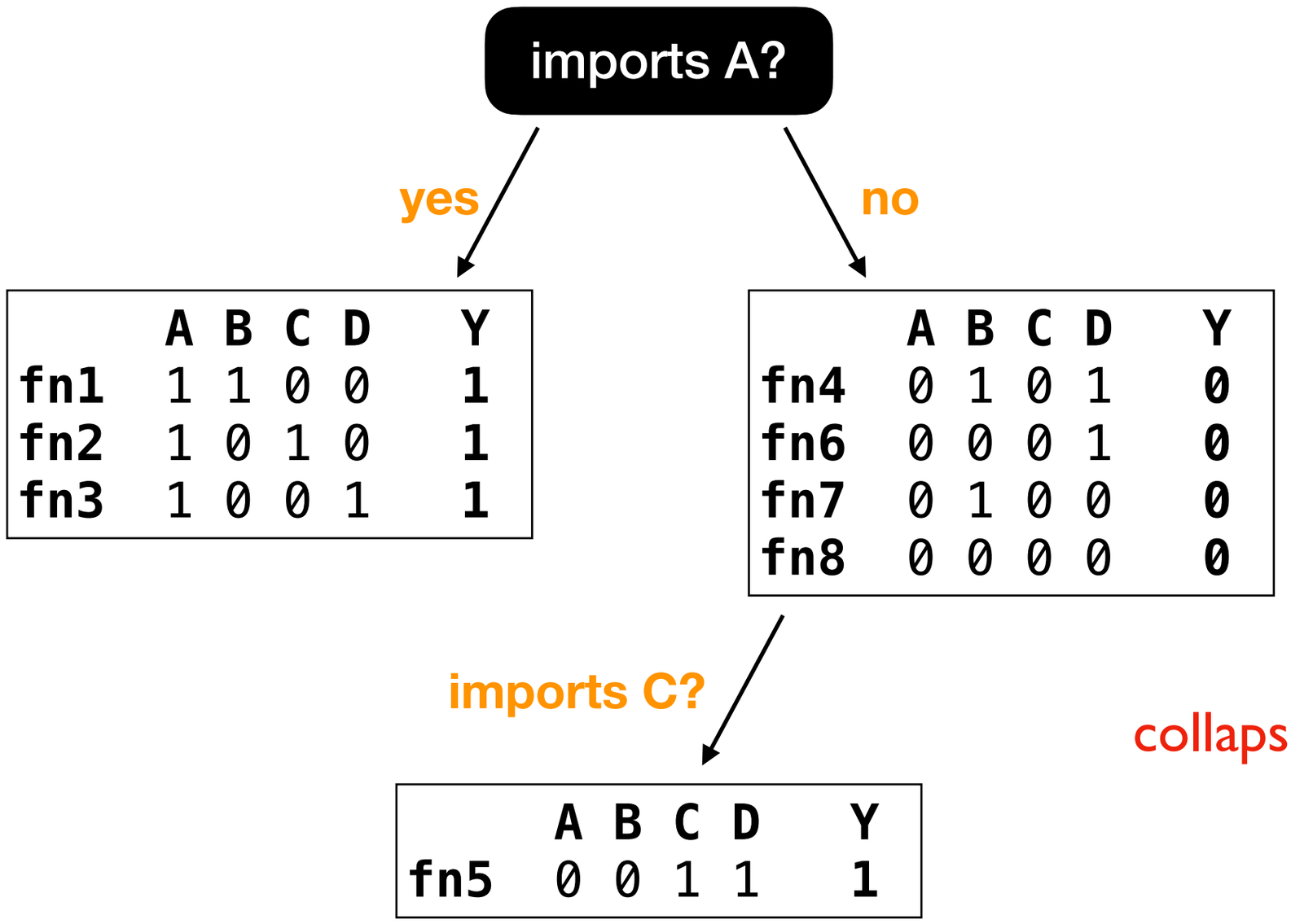
This perfectly fits the training data!

Note: we *could* also construct a 5-node Zygote tree from this decision tree

Formulating Lambda Trace for Decision Tree



Formulating Lambda Trace for Decision Tree



collapse nodes...

Formulating Lambda Trace for Decision Tree

	A	B	C	D	Y
fn4	0	1	0	1	0
fn6	0	0	0	1	0
fn7	0	1	0	0	0
fn8	0	0	0	0	0

imports A?

imports C?

	A	B	C	D	Y
fn1	1	1	0	0	1
fn2	1	0	1	0	1
fn3	1	0	0	1	1

	A	B	C	D	Y
fn5	0	0	1	1	1

collapse nodes...

Formulating Lambda Trace for Decision Tree

	A	B	C	D	Y
fn4	0	1	0	1	0
fn6	0	0	0	1	0
fn7	0	1	0	0	0
fn8	0	0	0	0	0

to match original behavior, evaluate children left to right

imports A?

imports C?

	A	B	C	D	Y
fn1	1	1	0	0	1
fn2	1	0	1	0	1
fn3	1	0	0	1	1

	A	B	C	D	Y
fn5	0	0	1	1	1

like replacing *nested if/else* with chain of *else ifs*

Formulating Lambda Trace for Decision Tree

	A	B	C	D	Y
fn4	0	1	0	1	0
fn6	0	0	0	1	0
fn7	0	1	0	0	0
fn8	0	0	0	0	0

imports A?

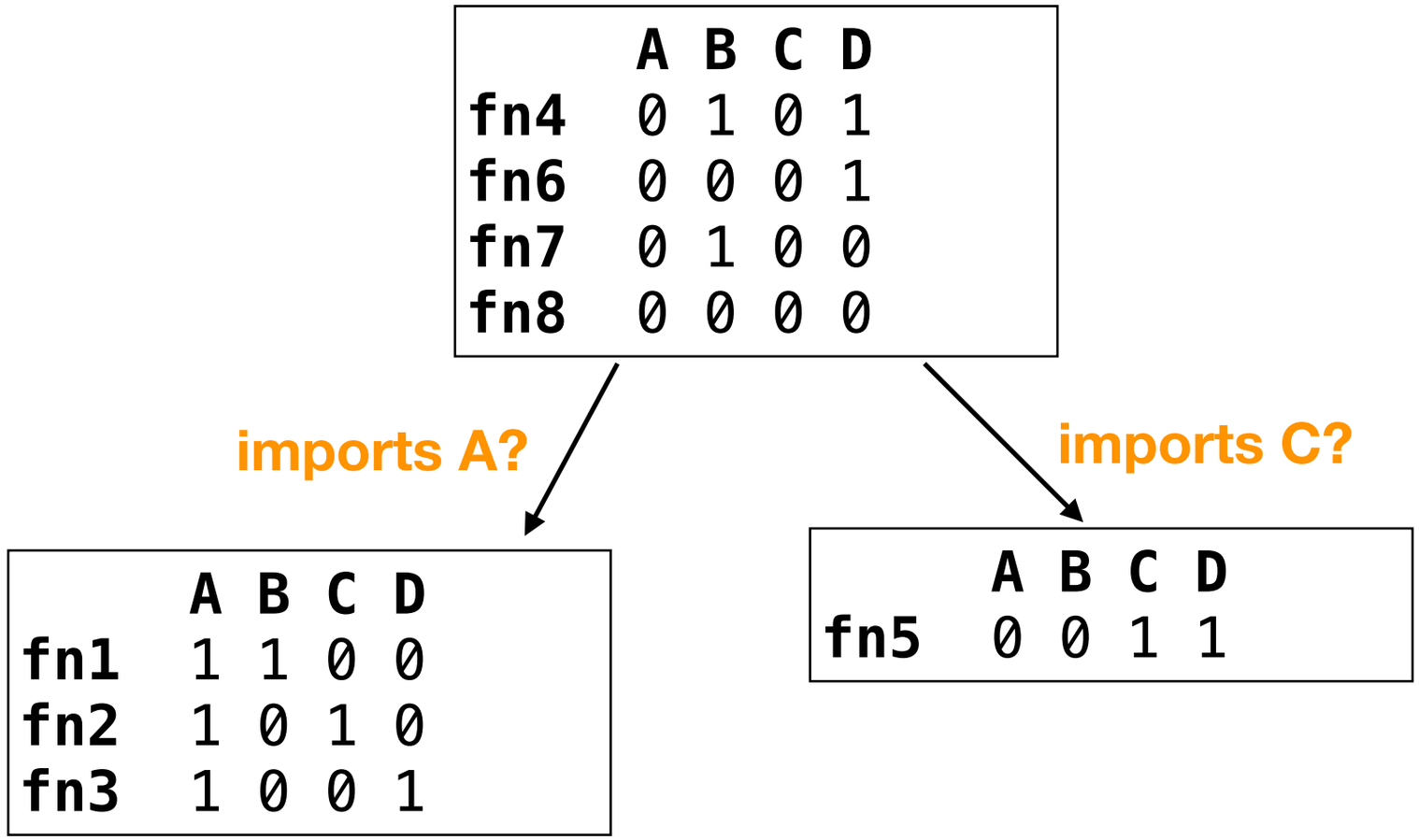
imports C?

	A	B	C	D	Y
fn1	1	1	0	0	1
fn2	1	0	1	0	1
fn3	1	0	0	1	1

	A	B	C	D	Y
fn5	0	0	1	1	1

Issue 2: what is Y anyway?

Formulating Lambda Trace for Decision Tree

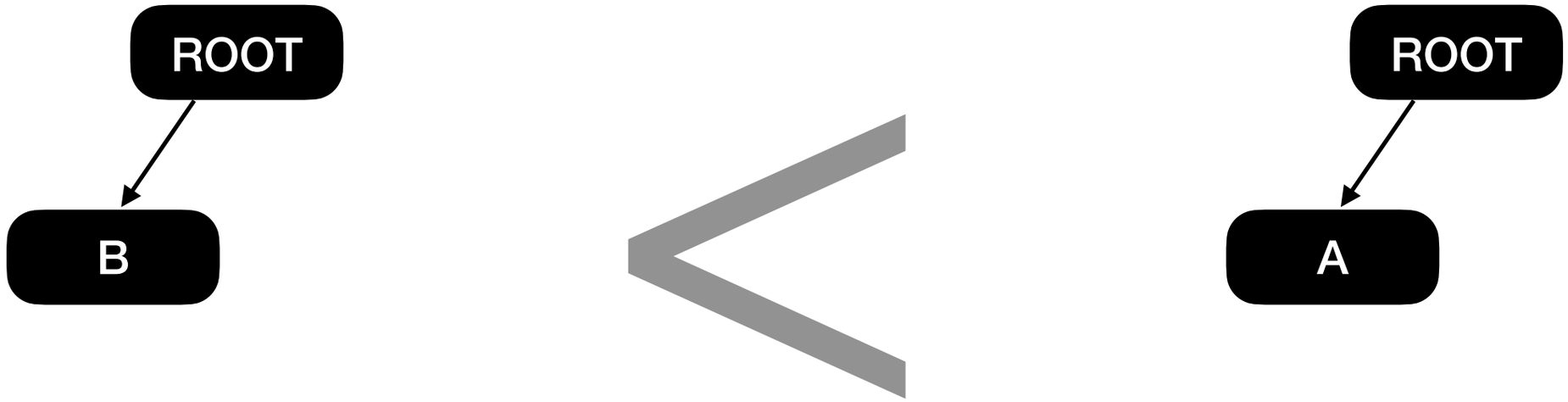


Issue 2: what is Y anyway?

We don't need it if we invent our own perf-oriented impurity measure.
We're not trying to classify anything!

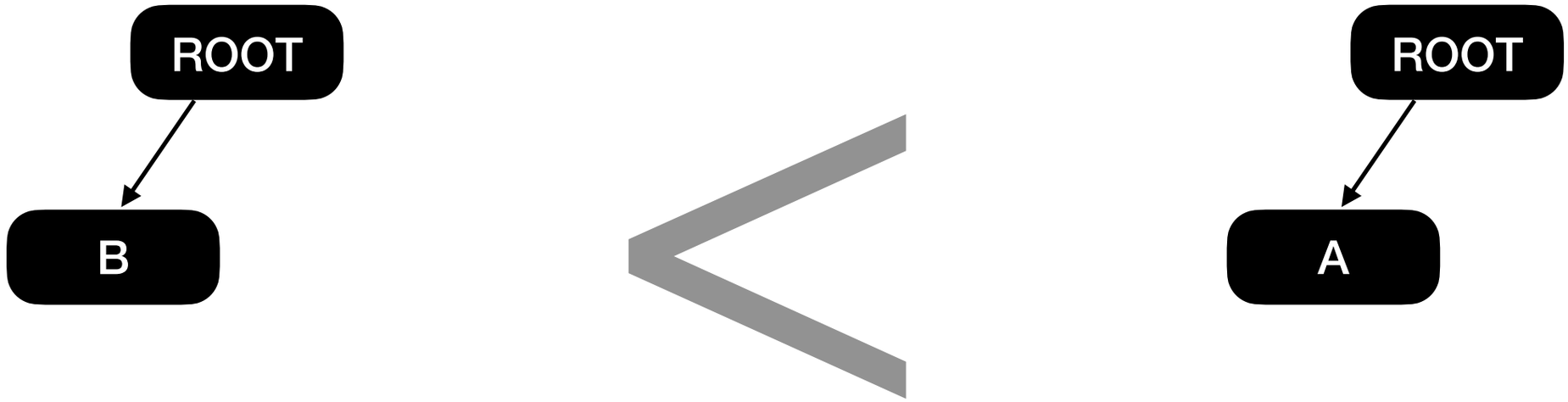
we'll propose 5 components for such a metric

Rule 0: Prioritize common packages



	pkg A	pkg B
fn 1	**1**	**1**
fn 2	**1**	**0**
fn 3	**1**	**0**
fn 4	**0**	**0**

Rule 0: Prioritize common packages



	pkg A	pkg B	"Y"
fn 1	**1**	**1**	1
fn 2	**1**	**0**	0
fn 3	**1**	**0**	0
fn 4	**0**	**0**	1

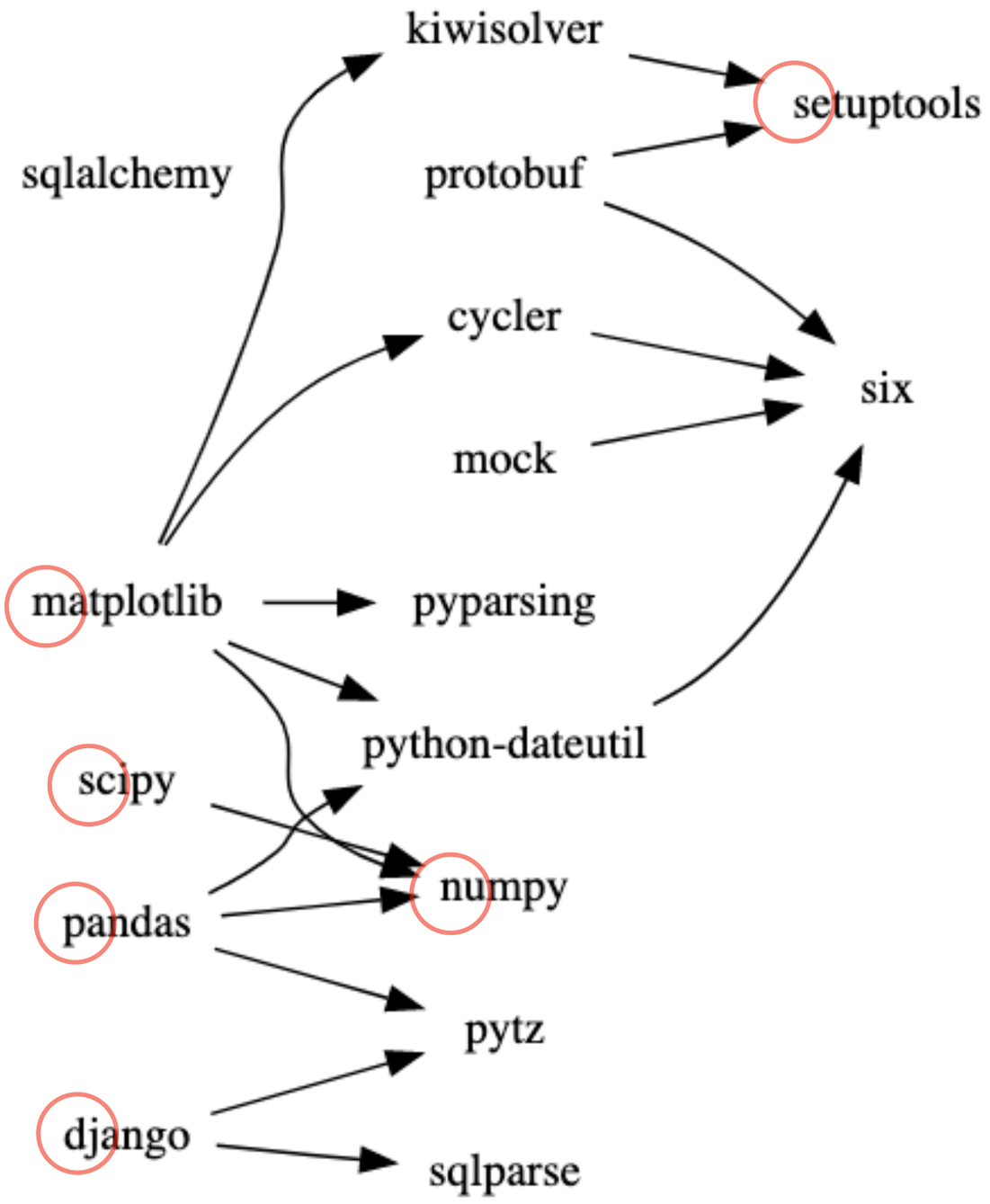
if we *were* classifying, note that we wouldn't have any bias for/against 1 or 0

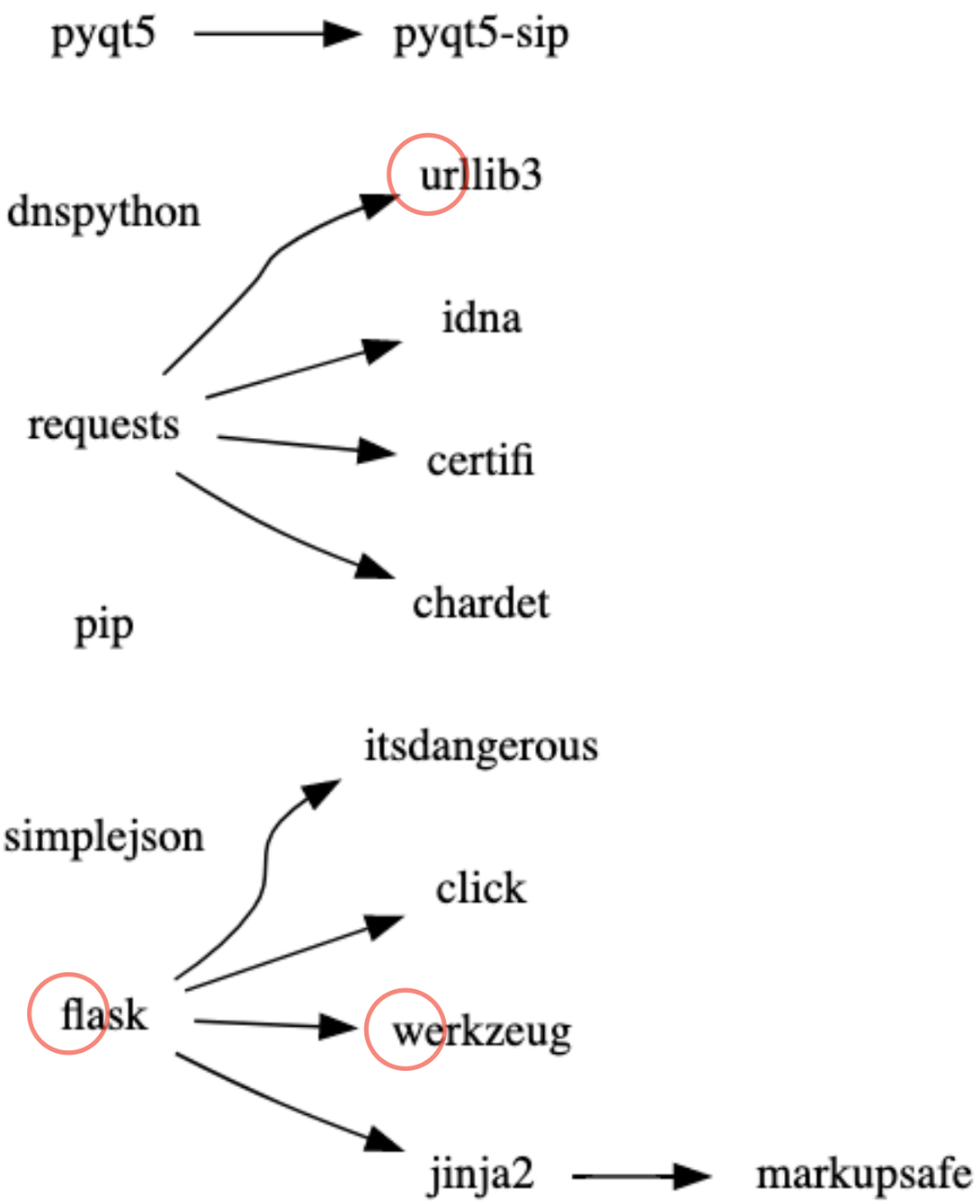
Evaluation: Workload Generator

32 Real Packages

- Start with 20 top packages (SOCK paper)
- Eliminate 2 that don't work with OpenLambda currently
- Add 14 dependencies

Benchmark Packages and Dependencies





Evaluation: Workload Generator

32 Real Packages

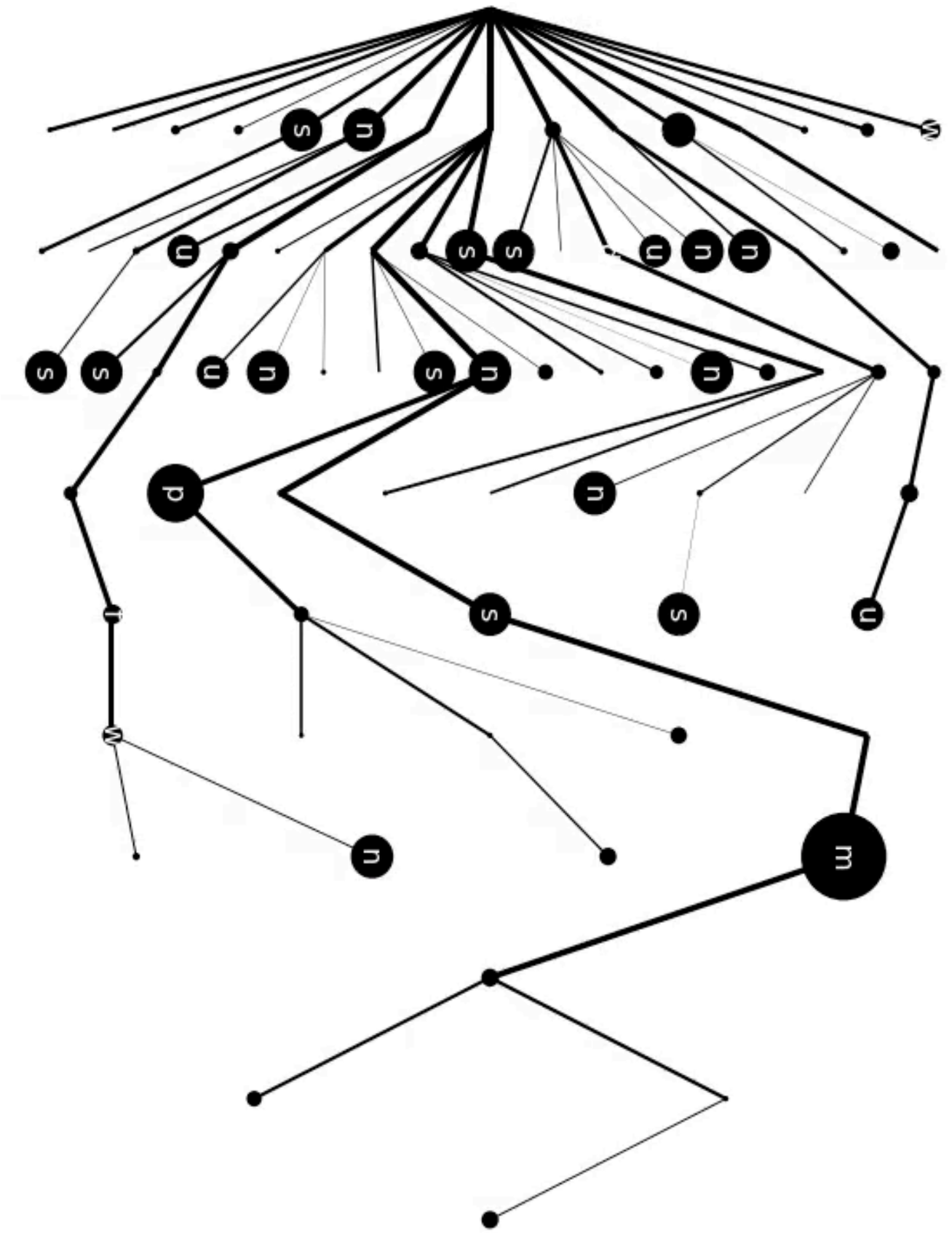
- Start with 20 top packages (SOCK paper)
- Eliminate 2 that don't work with OpenLambda currently
- Add 14 dependencies

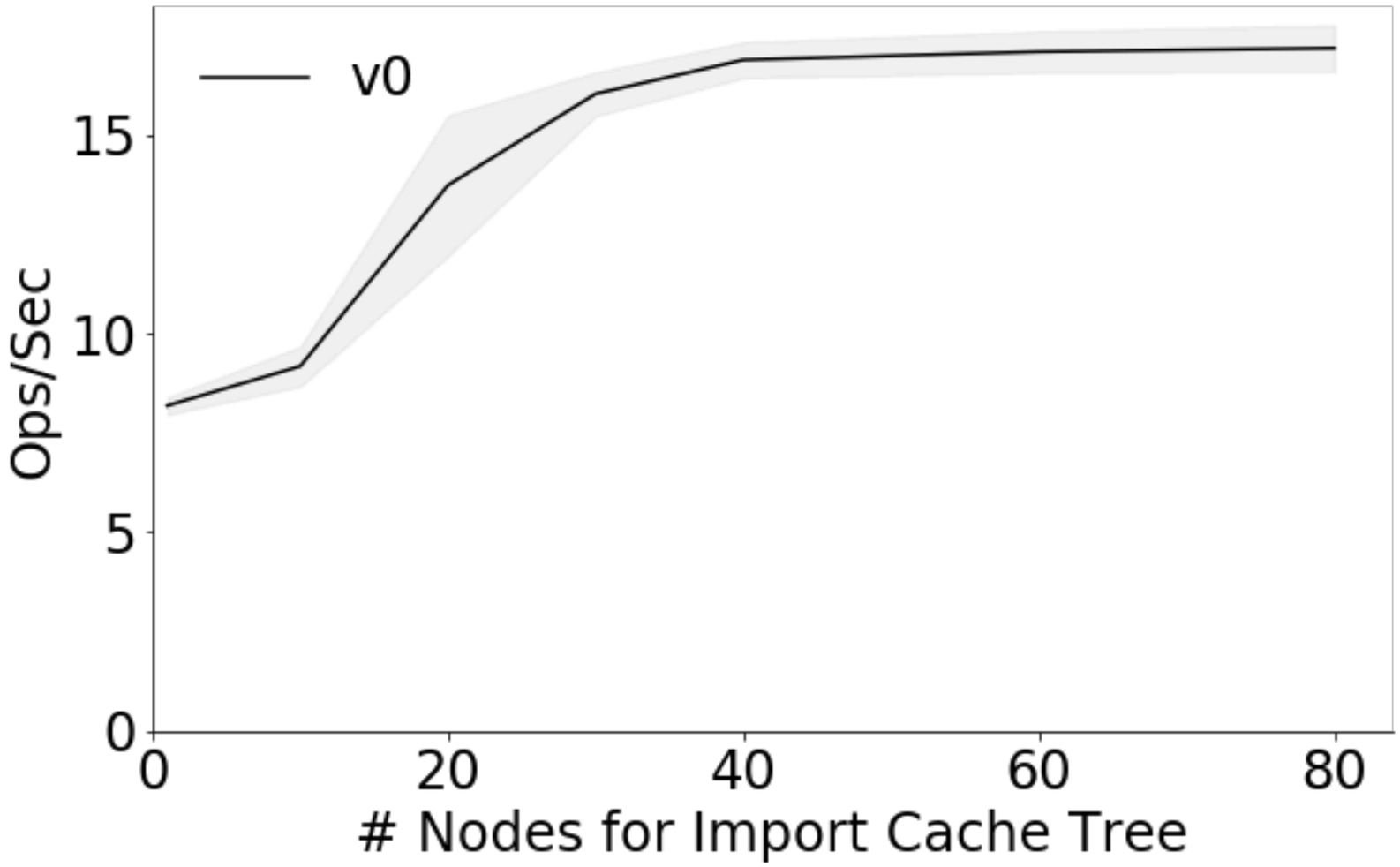
1000 Synthetic Lambdas

- For each, pick 2 of the 32 packages, uniformly at random (the 2 picks are independent)
- Import those 2 (this usually entails importing deps too)
- By importing a package, we mean importing it's top-level modules
- Do nothing else (no-op)

Client:

- Invoke each Lambda once
- 5 concurrent invocations
- Measure throughput





Observation: up to ~40 Zygote nodes are useful, and gives us ~16 invocations/sec (this is on a 2-core, 4-GB VM on my Macbook)

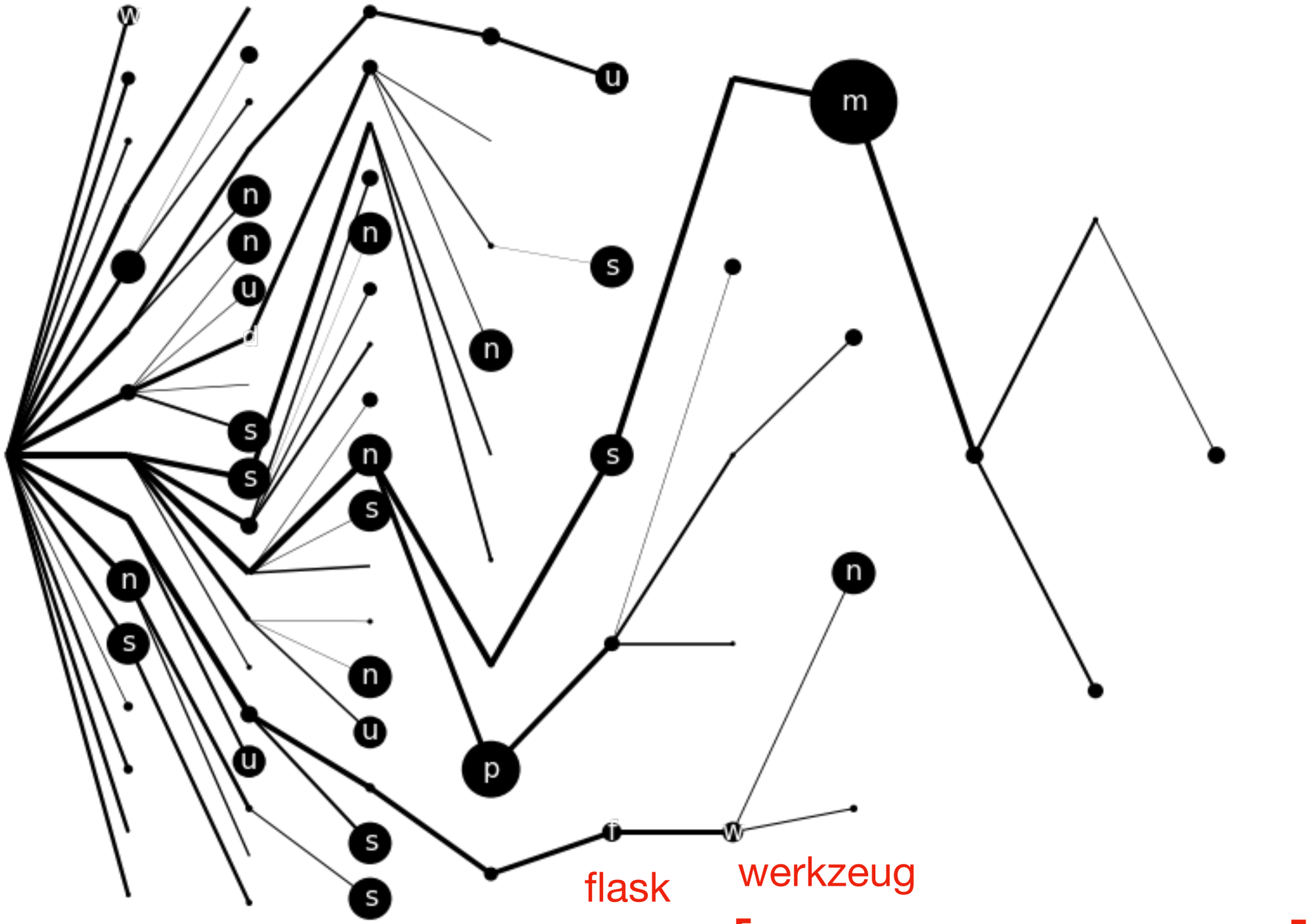
More Methodology Details (previous and subsequent plots)

Overfitting

- Generate pairs of train/test traces
- Fit Zygote tree to train trace
- Measure throughput of that tree on test trace

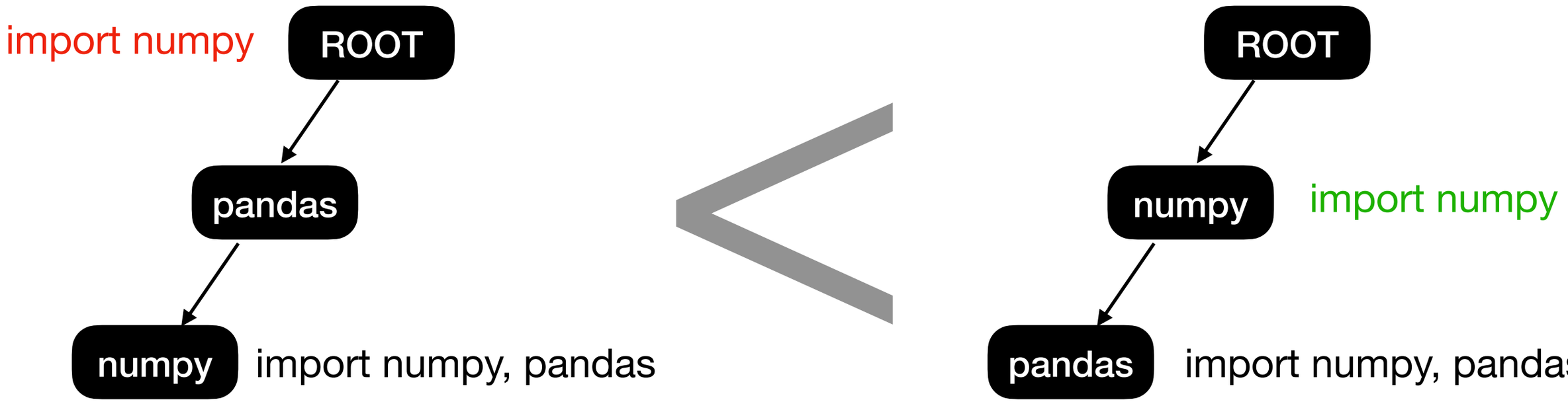
Variance

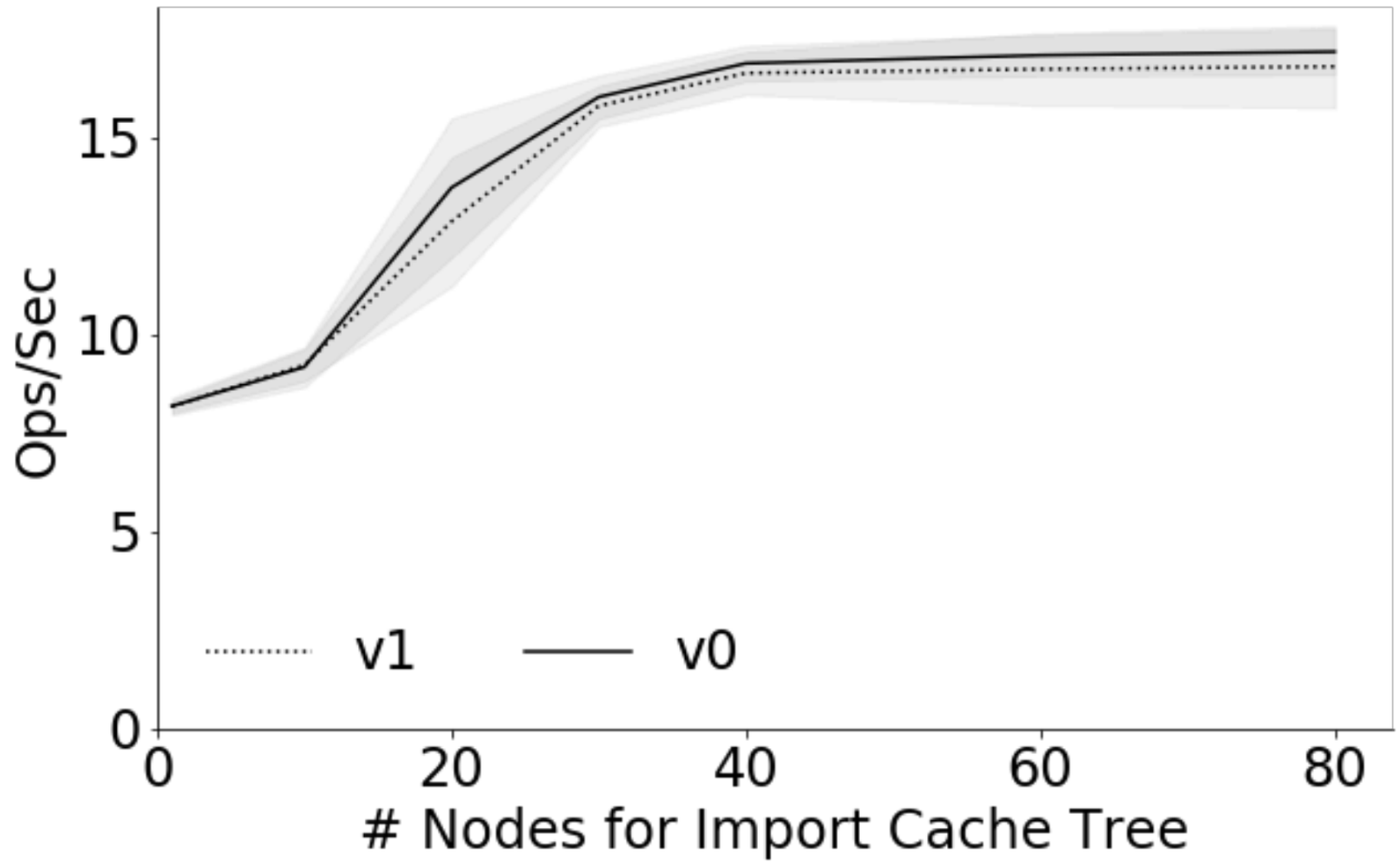
- Generate 5 train/test pairs
- Gray regions represent standard deviation
- Each test trace is executed for various tree sizes, so the same sample of 5 is reused along the x-axis



[this Zygote is never used because
flask depends on werkzeug]

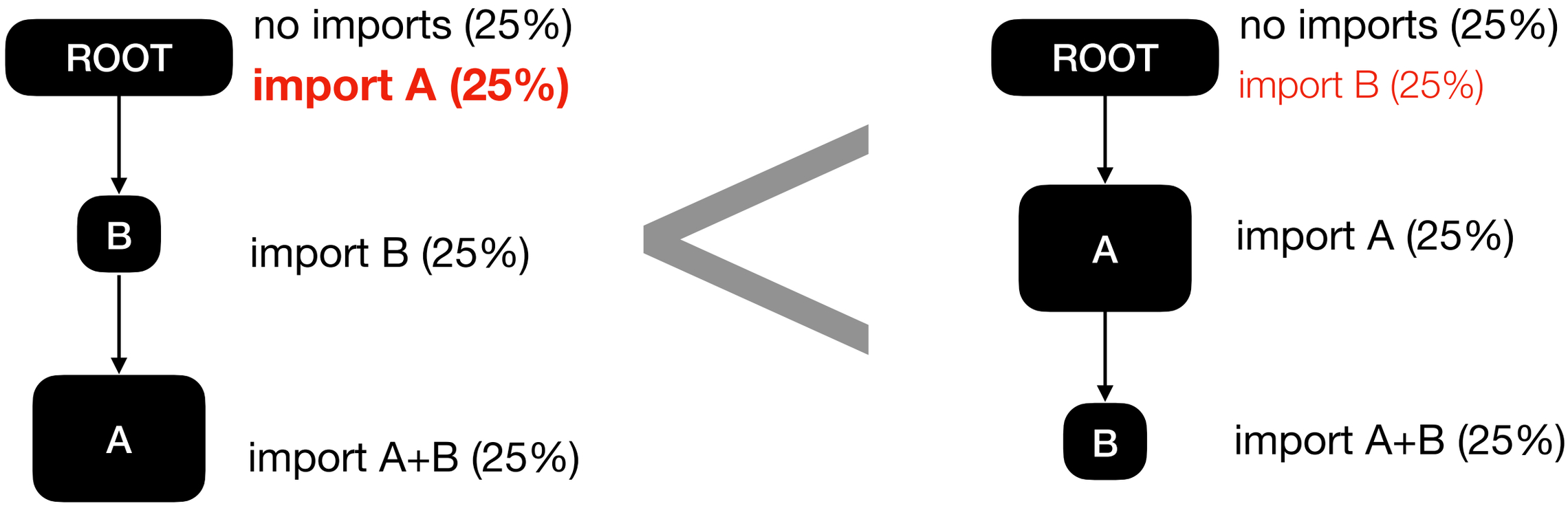
Rule 1: Put Prereqs Closer to Root



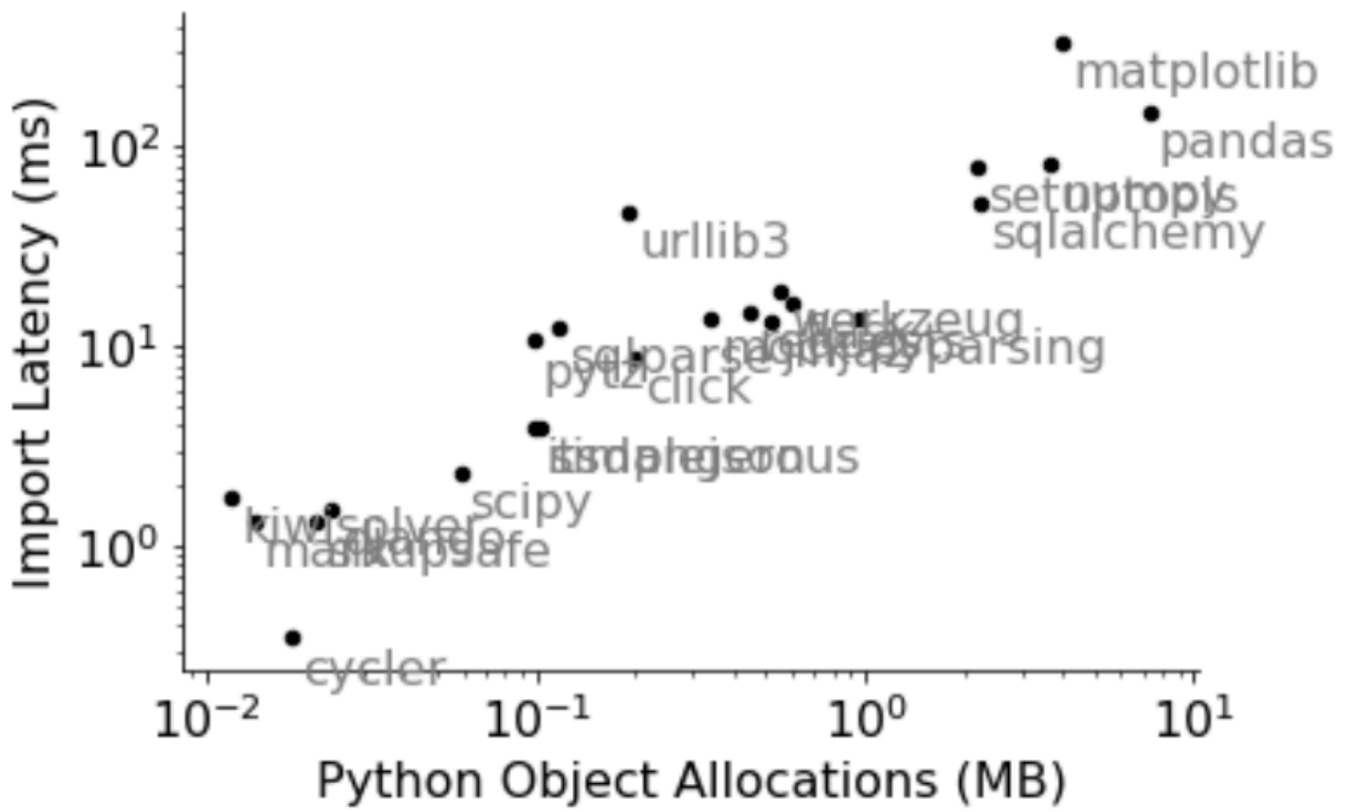


Observation: enforcing pre-reqs didn't help, perhaps because the workload tends to make prereqs be used more often (so they'll usually be near the root anyway)

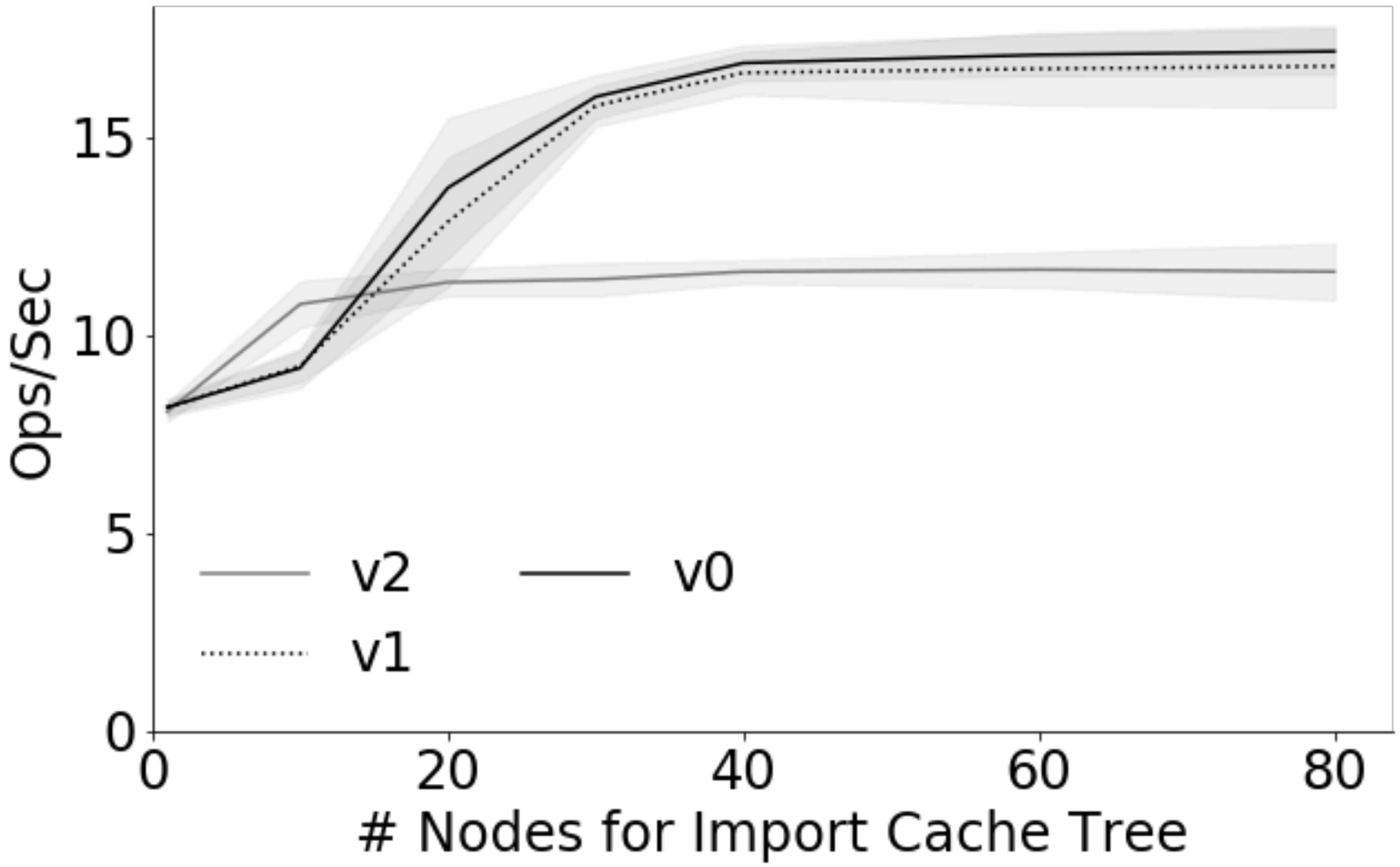
Rule 2: Put Heavy Modules Near Root



Assume there are no package deps between A and B

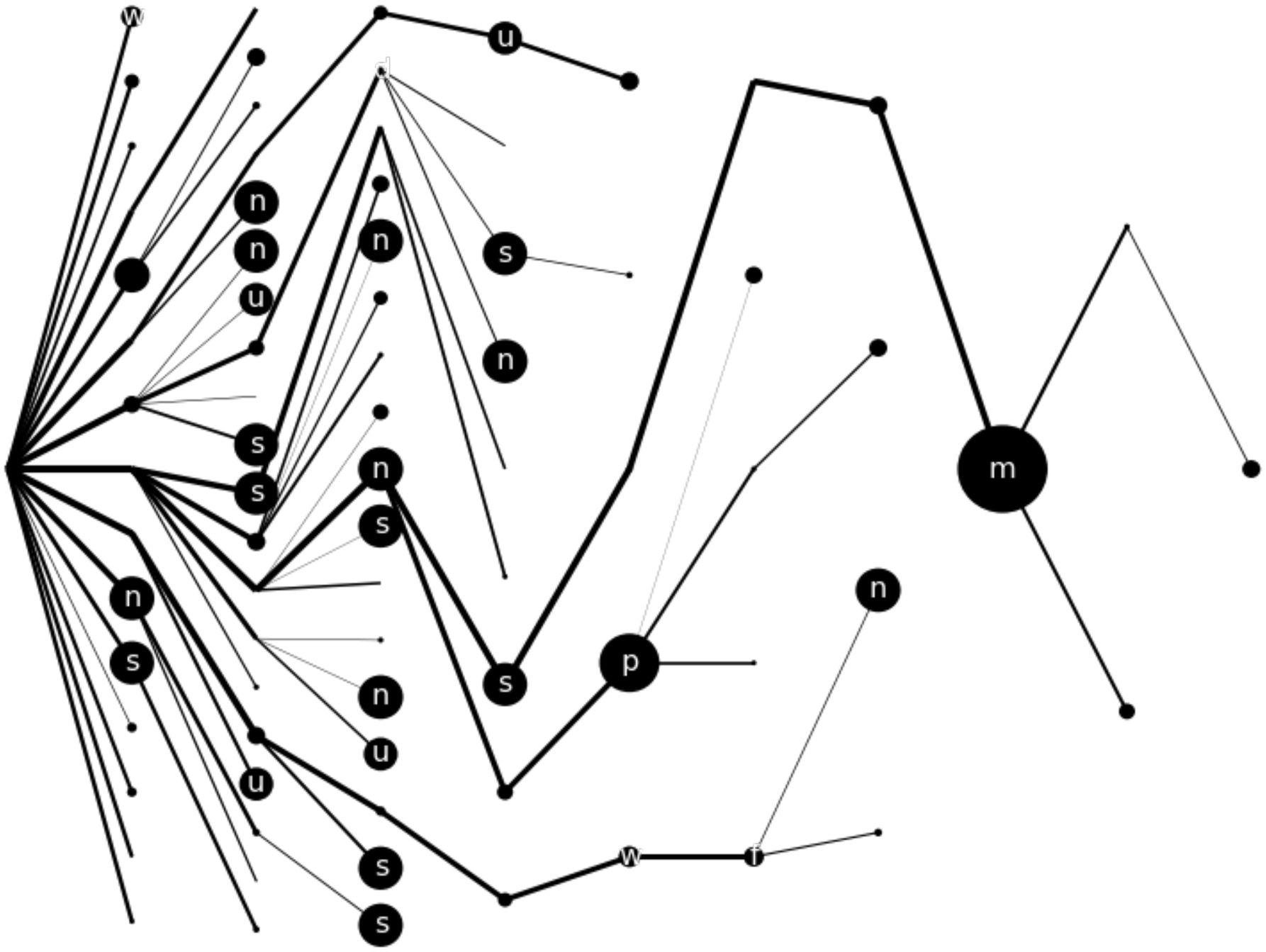


We use import time as the weight/cost of a package
(this is somewhat correlated with the alternative metric, mem footprint)

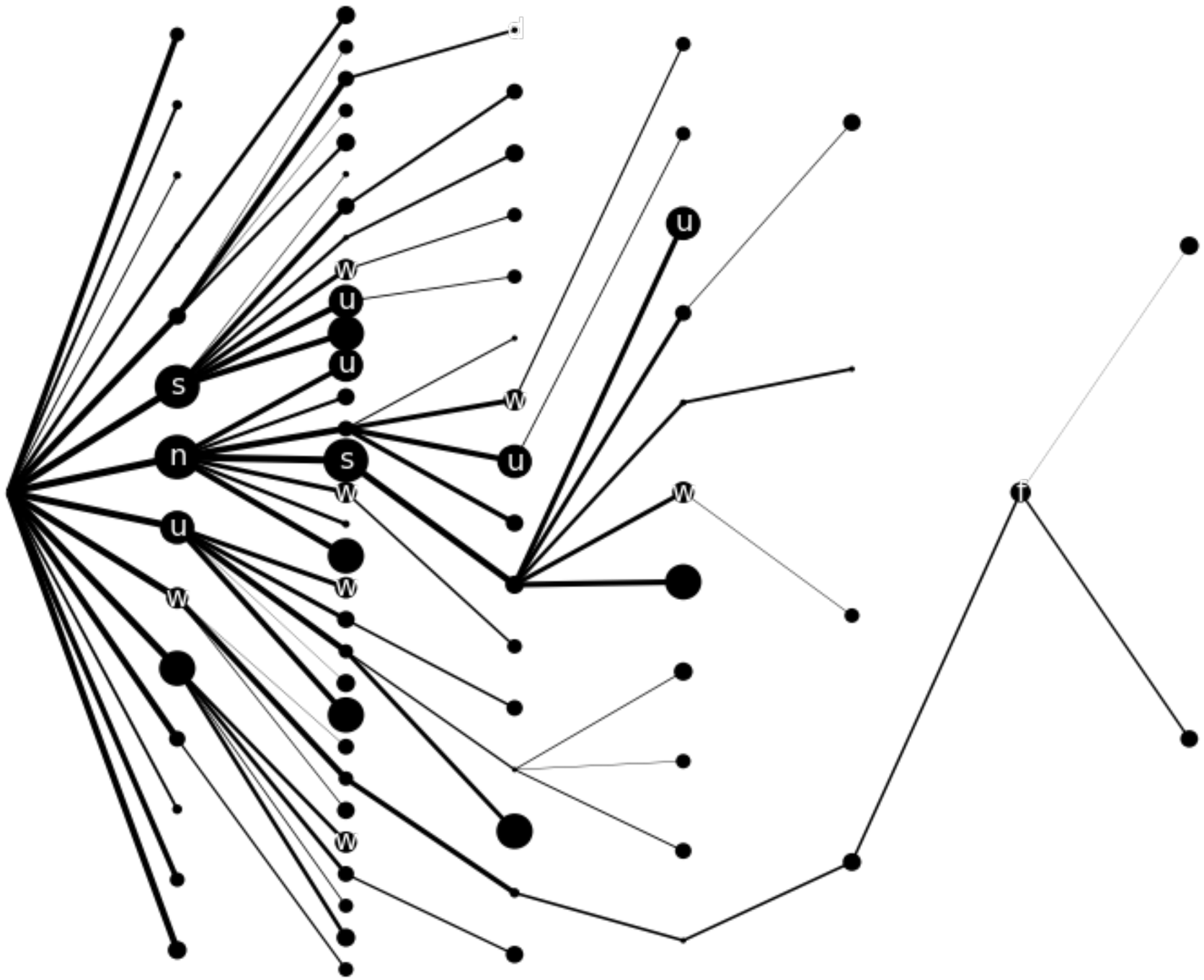


Observation: this rule really hurts perf! Why?

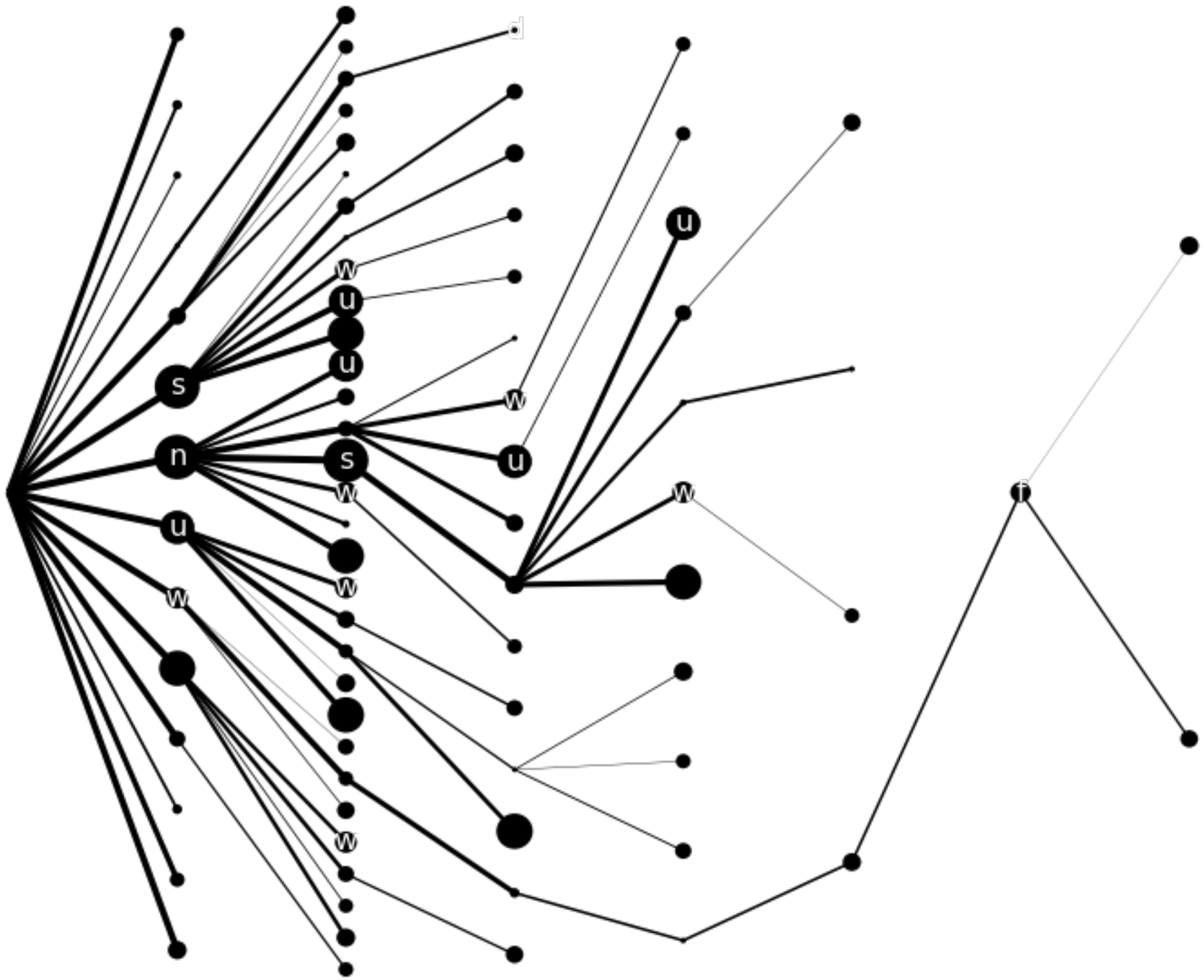
Without Rule 2



With Rule 2

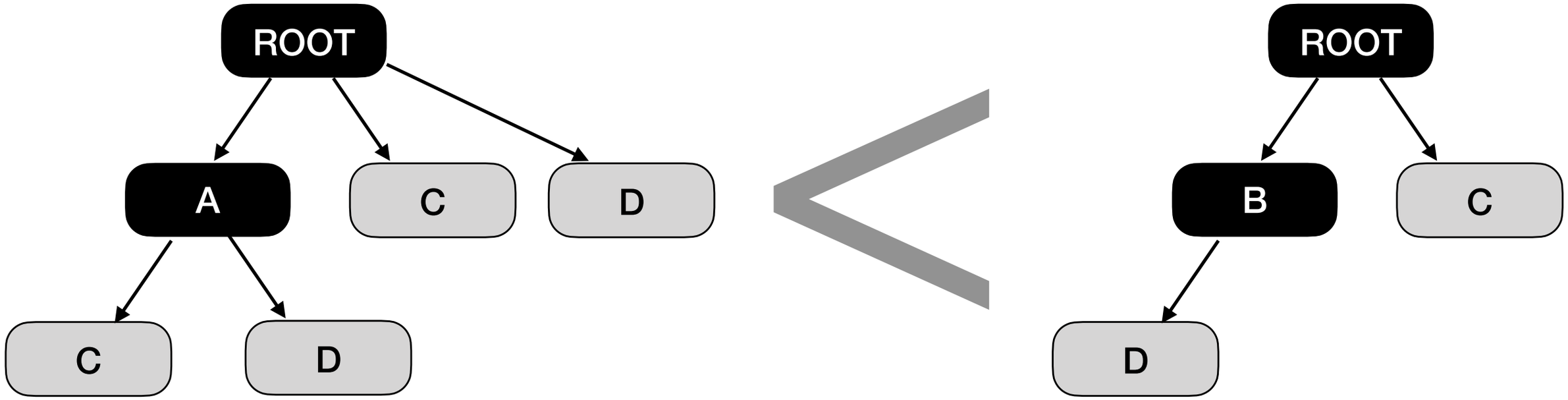


With Rule 2

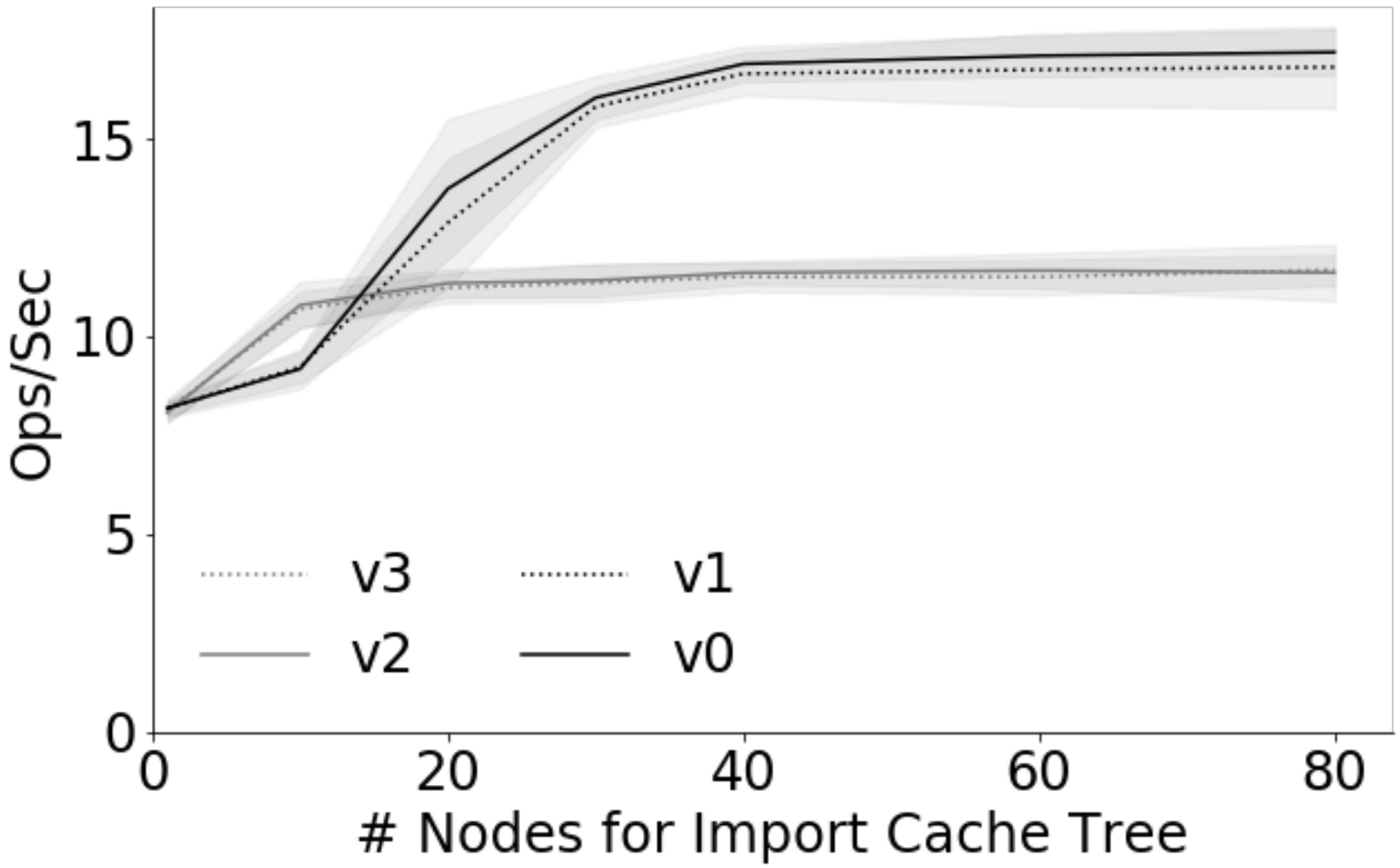


Observation: the algorithm is greedy. We're not getting to very heavy packages (e.g., pandas + matplotlib) that depend on very light packages.

Rule 3: Split to Minimize Entropy

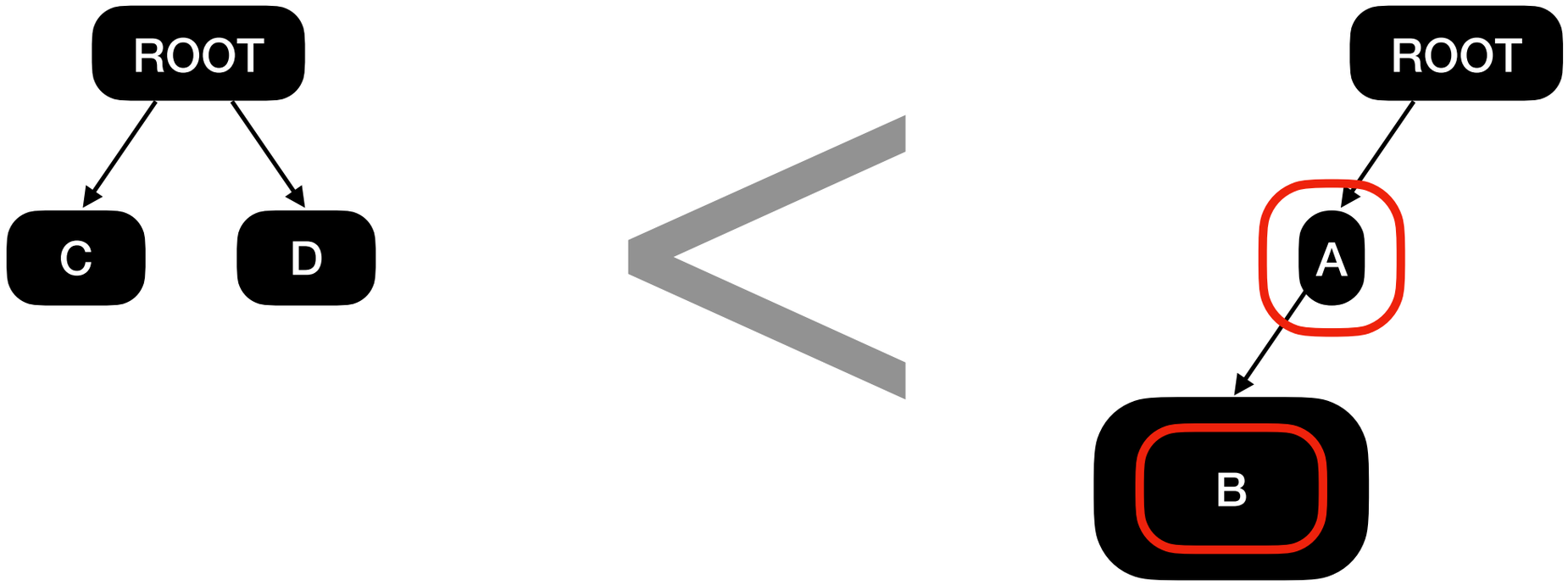


	pkg A	pkg B	pkg C	pkg D
fn 1	**1**	**1**	**0**	**1**
fn 2	**1**	**0**	**1**	**0**
fn 3	**0**	**1**	**0**	**1**
fn 4	**0**	**0**	**1**	**0**



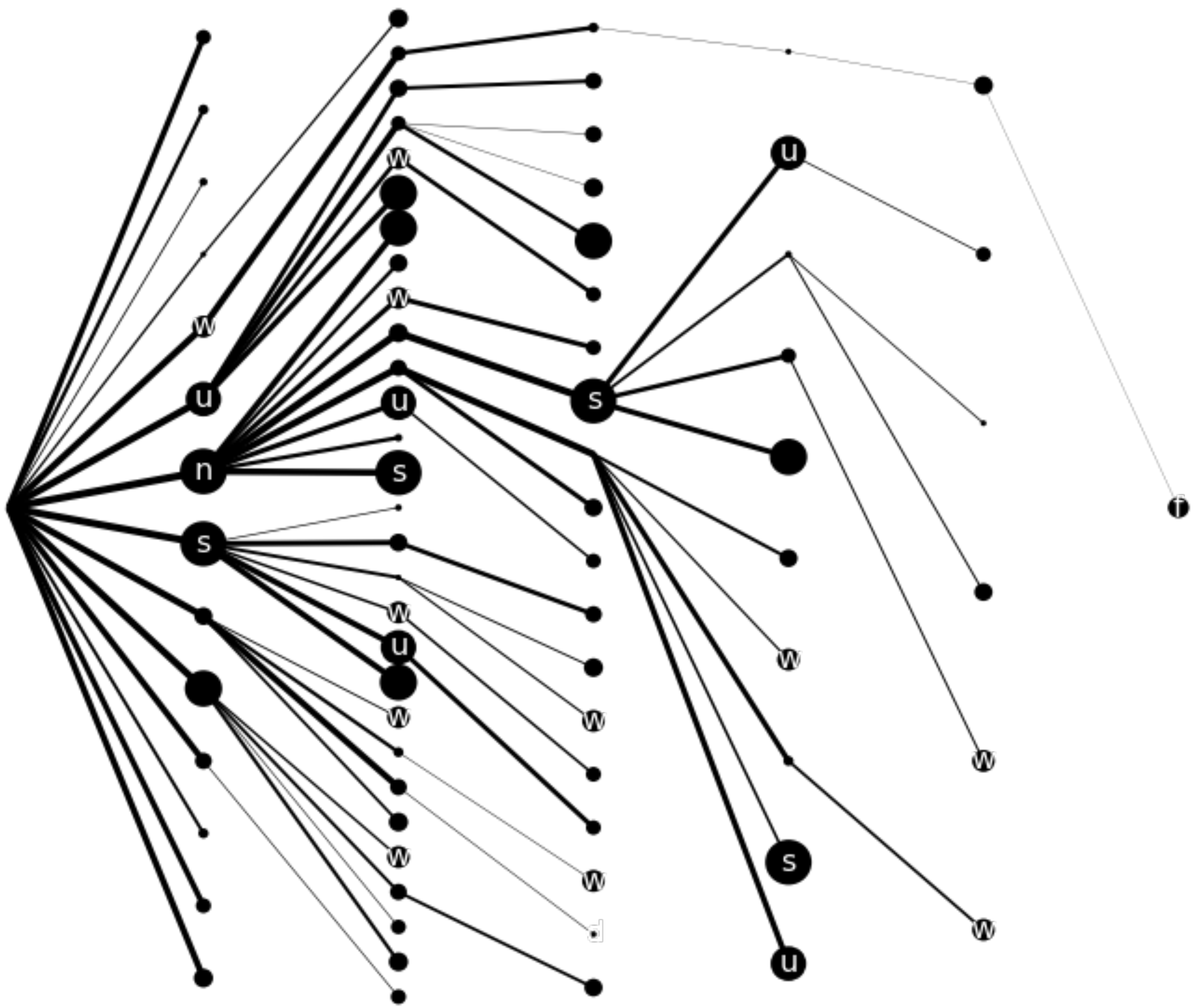
Observation: this addresses one kind of shortsighted greediness, but doesn't solve the priority inversion problem.

Rule 4: Distribute weights to dependencies (like "priority inheritance")

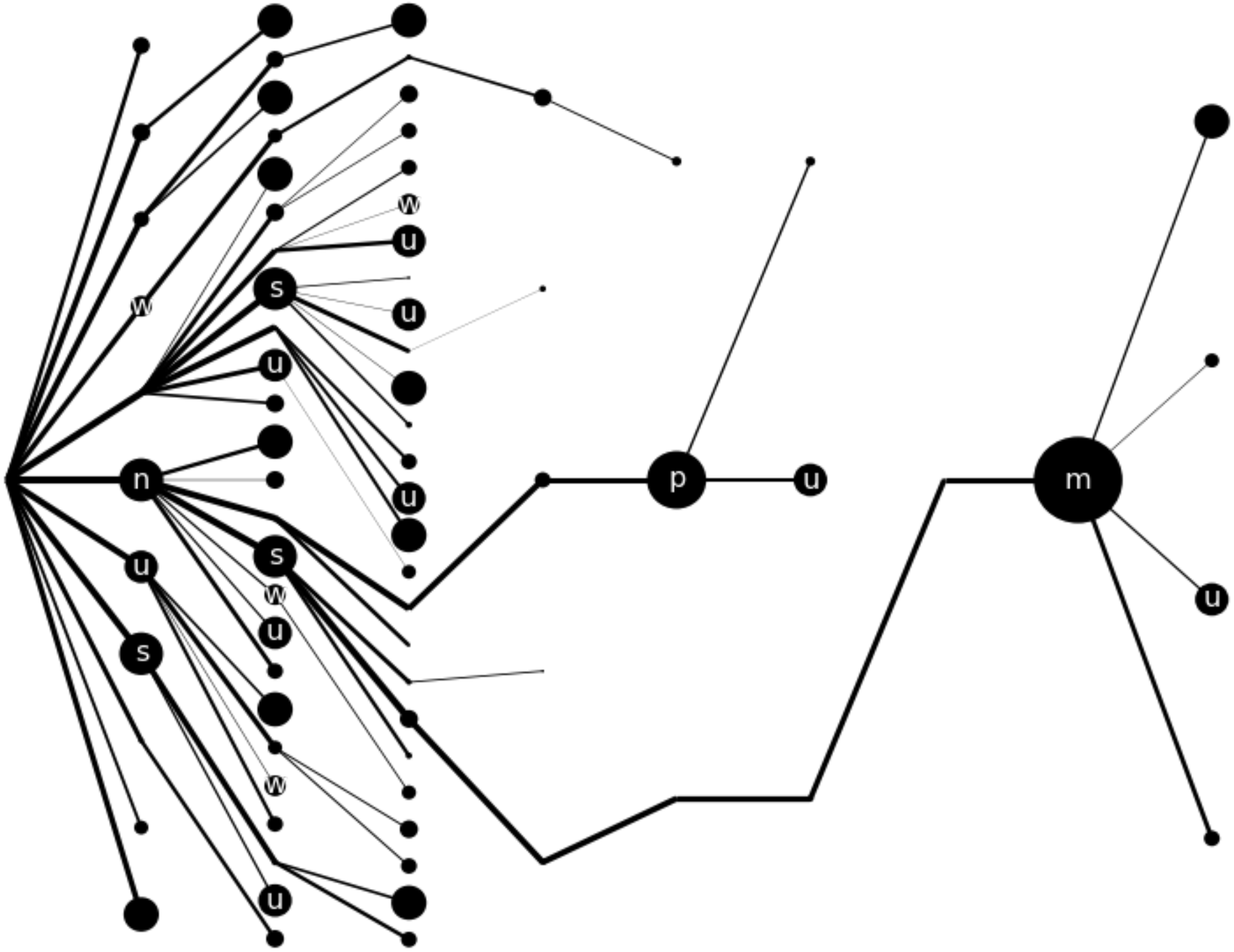


A, C, and D have no deps. B is large and depends on A, which is tiny.

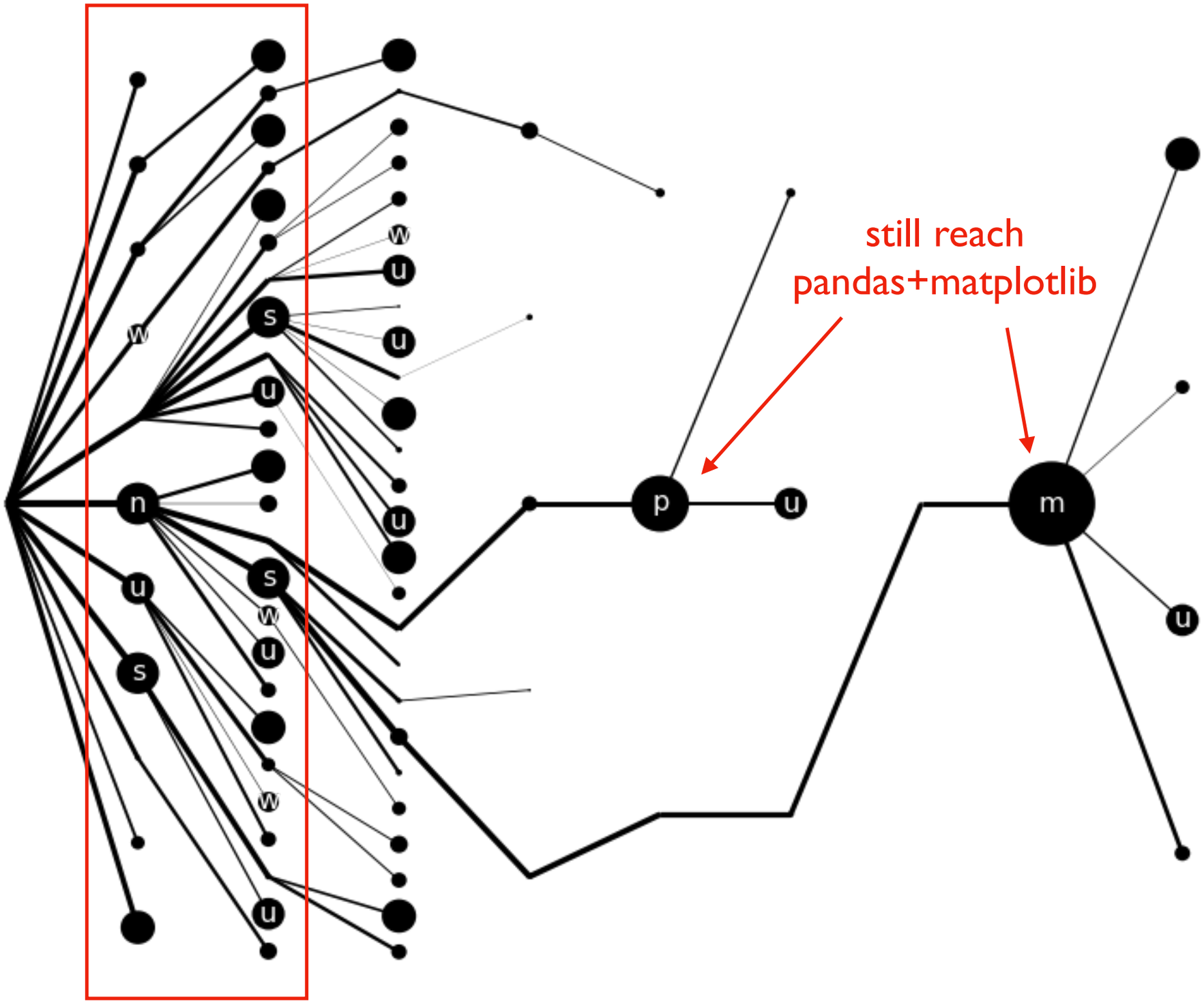
Without Rule 4



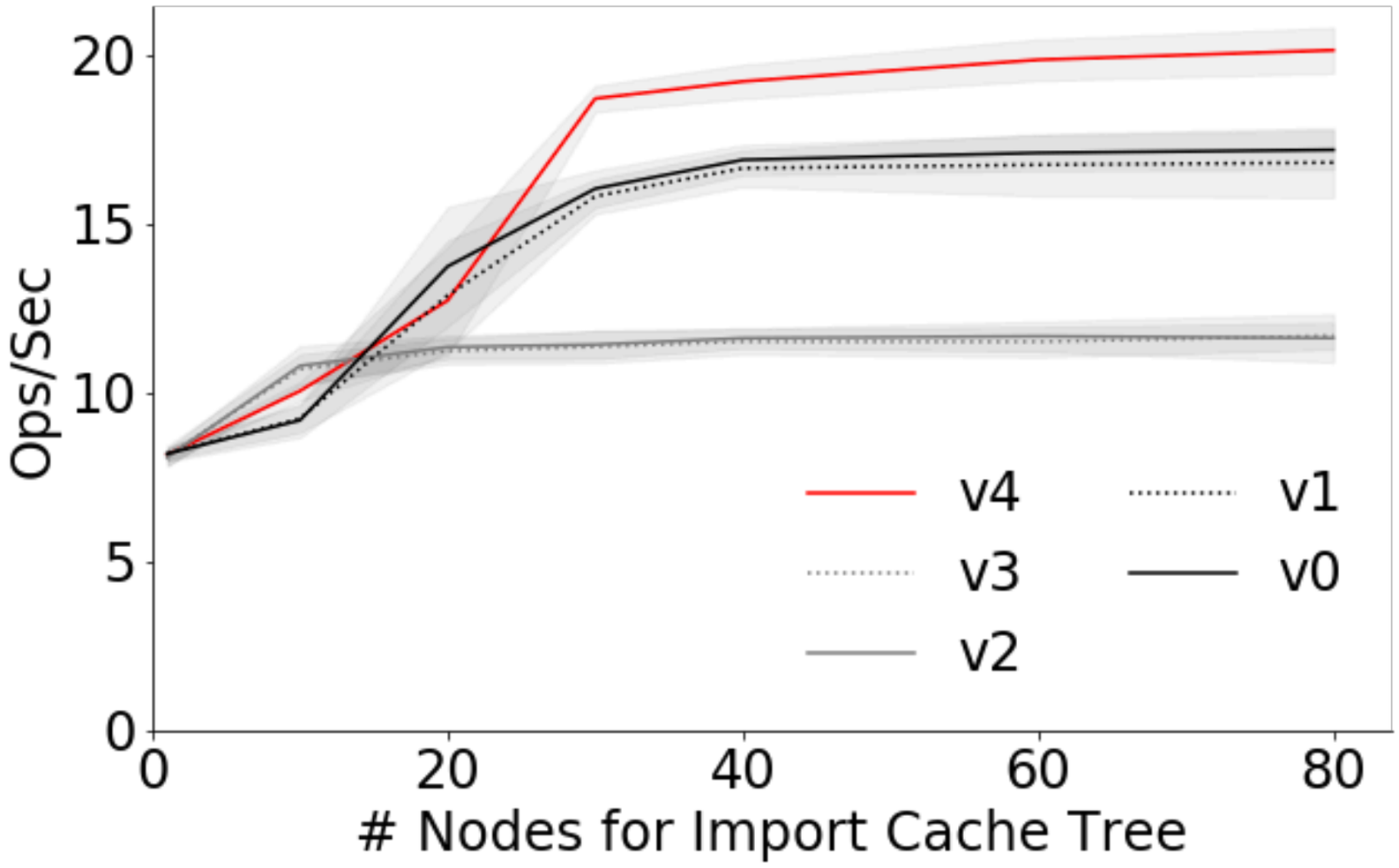
With Rule 4



With Rule 4



lots of heavy packages



Observation: adding priority inheritance gives best performance!

Concluding Thoughts...

1. Start with a smart static policy
   - Make dynamic as a tweak

2. What ML algorithms can we repurpose as policies?
   - A small tweak to the decision tree algorithm converts it from a classifier to a cache policy