ON THE COST OF TYPE-TAG SOUNDNESS

Ben Greenman Zeina Migeed

Northeastern University

ON THE COST OF TYPE-TAG SOUNDNESS

1. Tag soundness
2. Performance cost of soundness
3. Evaluation method
4. Conclusions

TYPE-TAG SOUNDNESS

Type Soundness

If $\vdash \mathbf{e} : \tau$ then either:

- $\mathbf{e} \longrightarrow^{*} \mathbf{v}$ and $\vdash \mathbf{v} : \tau$
- $\mathbf{e}$ diverges
- $\mathbf{e} \longrightarrow^{*} \mathbf{Error}$ (division by zero, etc.)

No undefined behavior

Type-based reasoning

Type Soundness

If $\vdash e : \tau$ then either:

- $e \longrightarrow^{*} v$ and $\underline{\vdash v : \tau}$
- e diverges
- $e \longrightarrow^{*} \mathbf{Error}$ (division by zero, etc.)

Tag Soundness

If $\vdash e : \tau$ then either:

- $e \longrightarrow^{*} v$ and $\underline{\vdash v : \lfloor \tau \rfloor}$
- e diverges
- $e \longrightarrow^{*} \mathbf{Error}$ (division by zero, etc.)

Tag Soundness

If $\vdash e : \tau$ then either:

- $e \longrightarrow^{*} v$ and $\underline{\vdash v : \lfloor \tau \rfloor}$
- e diverges
- $e \longrightarrow^{*} \mathbf{Error}$ (division by zero, etc.)

$\boxed{\lfloor \tau \rfloor = K}$

$\lfloor \mathrm{Int} \rfloor = \mathrm{Int}$

$\lfloor \tau \times \tau' \rfloor = \mathrm{Pair}$

$\lfloor \tau \rightarrow \tau' \rfloor = \mathrm{Fun}$

…

Tag Soundness

If $\vdash e : \tau$ then either:

- $e \longrightarrow^{*} v$ and $\vdash v : \lfloor \tau \rfloor$
- e diverges
- $e \longrightarrow^{*} \mathrm{Error}$

$\boxed{\lfloor \tau \rfloor = K}$

$\lfloor \mathrm{Int} \rfloor = \mathrm{Int}$

$\lfloor \tau \times \tau' \rfloor = \mathrm{Pair}$

$\lfloor \tau \to \tau' \rfloor = \mathrm{Fun}$

...

No undefined behavior

Tag-based reasoning

Types vs. Tags

If $\vdash e : \mathrm{Int} \times \mathrm{Int}$ and $e \longrightarrow^{*} v$ then v might be:

Type Soundness	Tag Soundness
	("A", 0)
(0, 0)	(0, 0)
(3, 2)	(3, 2)
(-7, 9)	(-7, 9)
	(0, (1, 2))

Types vs. Tags

If $\vdash e : \mathrm{Int} \times \mathrm{Int}$ and $\underline{e \longrightarrow^{*} v}$ then v might be:

Type Soundness

(0, 0)

(3, 2)

(-7, 9)

Tag Soundness

("A", 0)

(0, 0)

(3, 2)

(-7, 9)

(0, (1, 2))

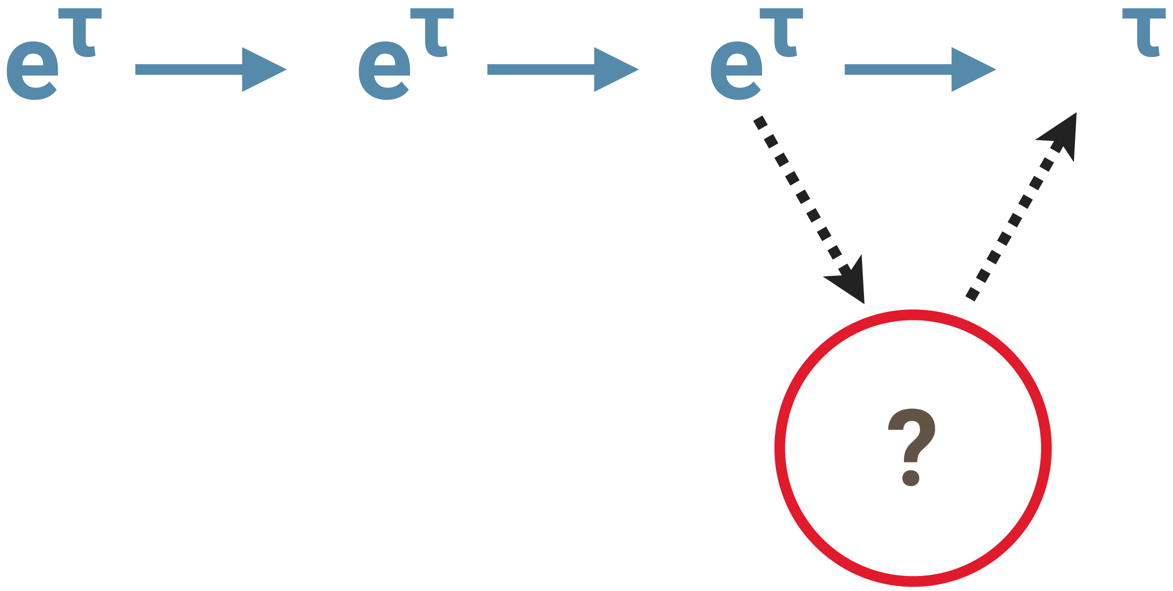
*fast

*slow

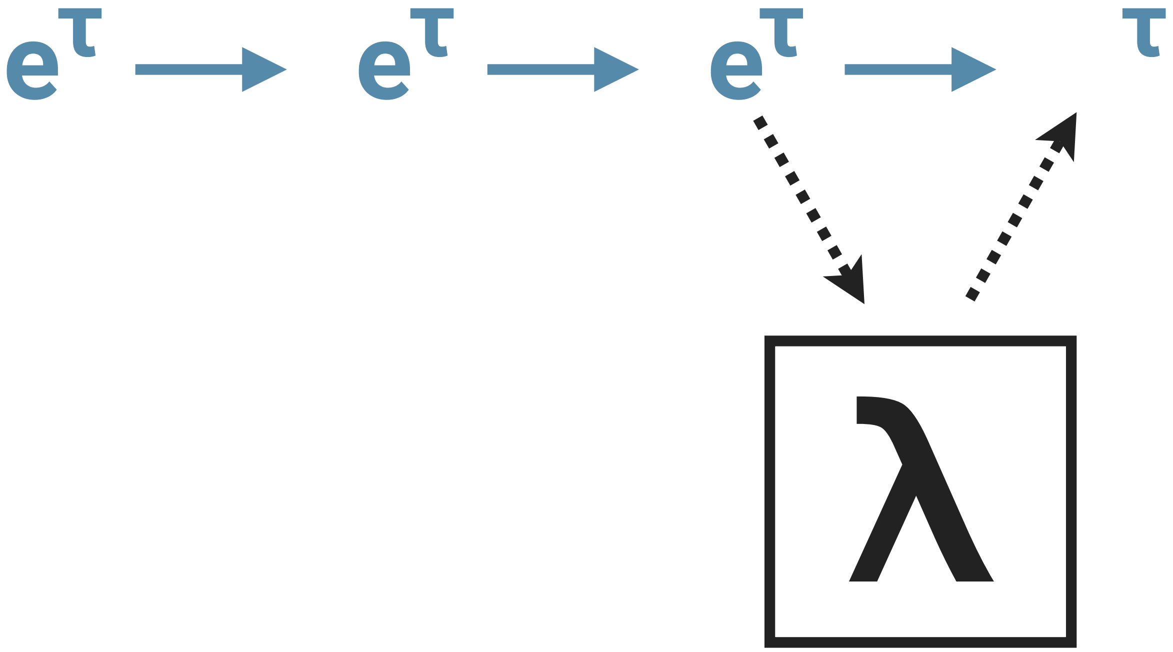
	$\rightarrow^{*}_{\text{fast}}$	$\rightarrow^{*}_{\text{slow}}$
Type Sound?	✗	✓
Tag Sound?	✓	✓

PERFORMANCE COST OF SOUNDNESS

Problem: Safe Interaction



Gradual Typing

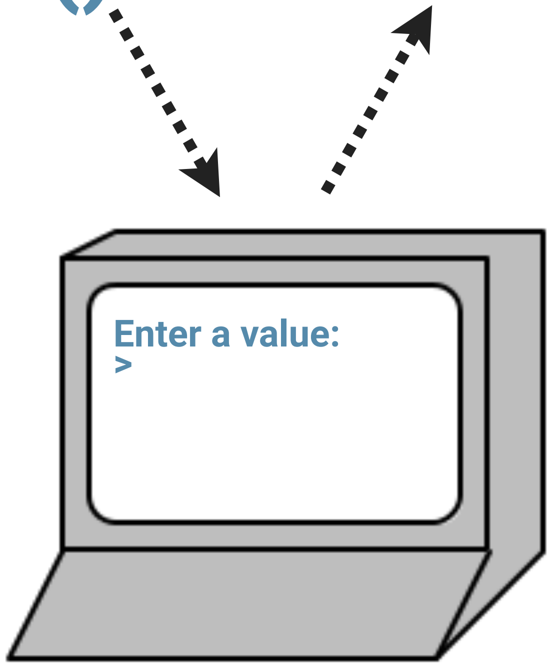


$e^{\tau} \rightarrow e^{\tau} \rightarrow e^{\tau} \rightarrow \tau$

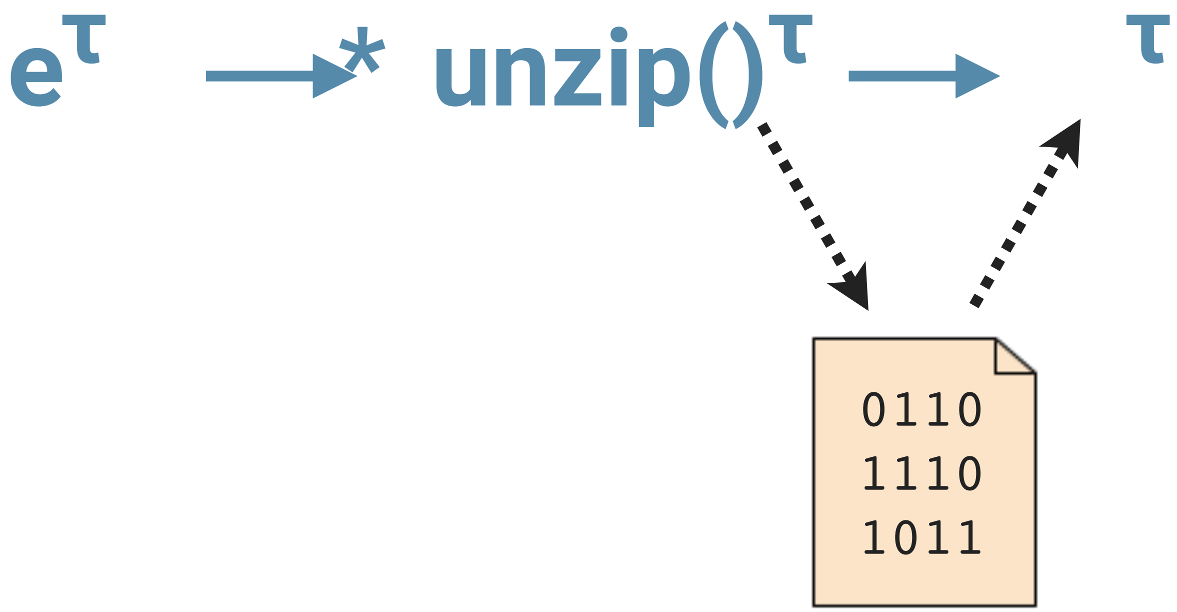
$\boxed{\lambda}$

User Input

$$e^{\tau} \longrightarrow^{*} \text{read}()^{\tau} \longrightarrow \tau$$



Deserialization

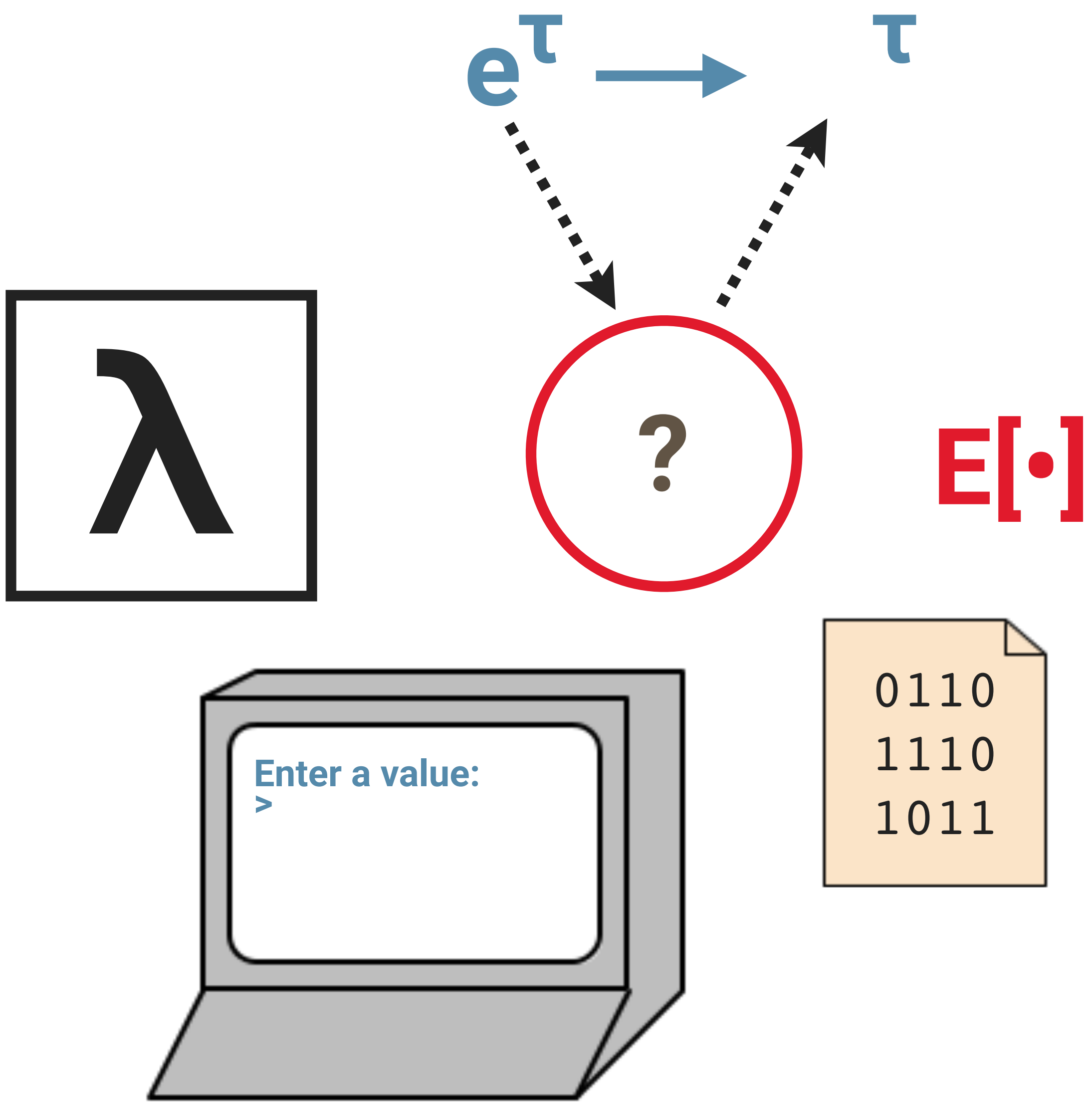


$$e^{\tau} \longrightarrow^{*} \text{unzip}()^{\tau} \longrightarrow \tau$$

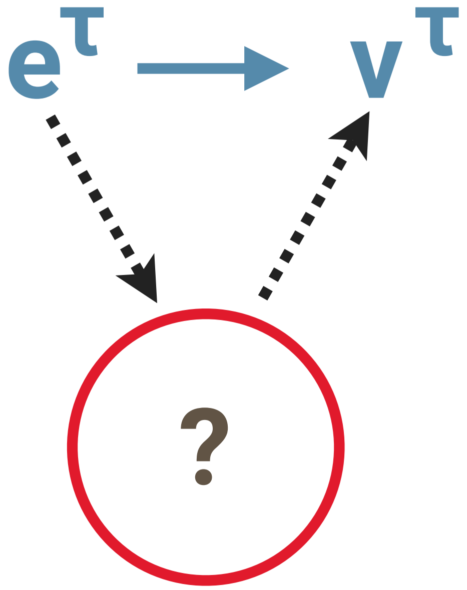
Primitive Operations (δ)

$$\begin{tikzcd}
e^{\mathrm{Int}} \arrow[r, "*"] & v + v^{\mathrm{Int}} \arrow[r] \arrow[d, dotted] & & \mathrm{Int} \\
& E[v,v] \arrow[r] & \ldots \arrow[r] & v' \arrow[u, dotted]
\end{tikzcd}$$

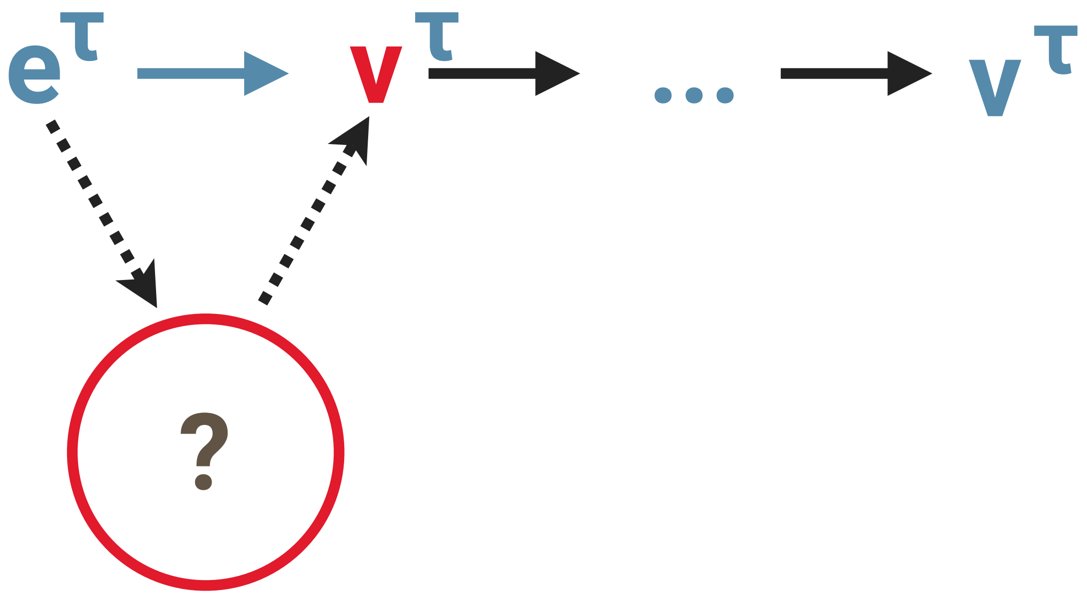
Unreliable Source



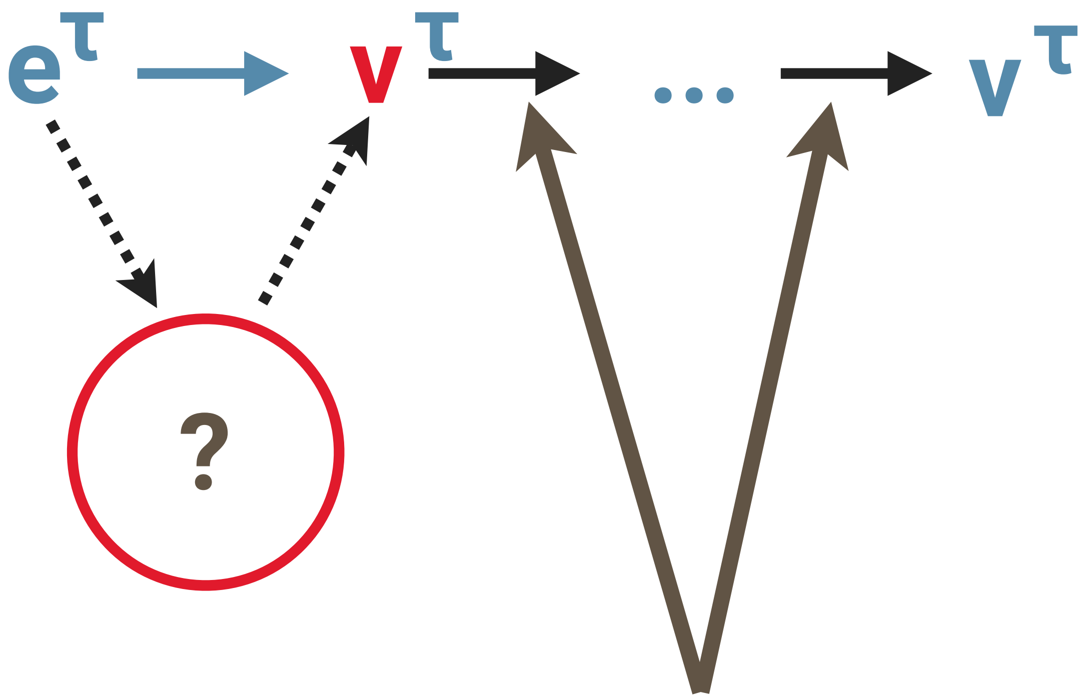
Option 1: Trust



Option 2: Check

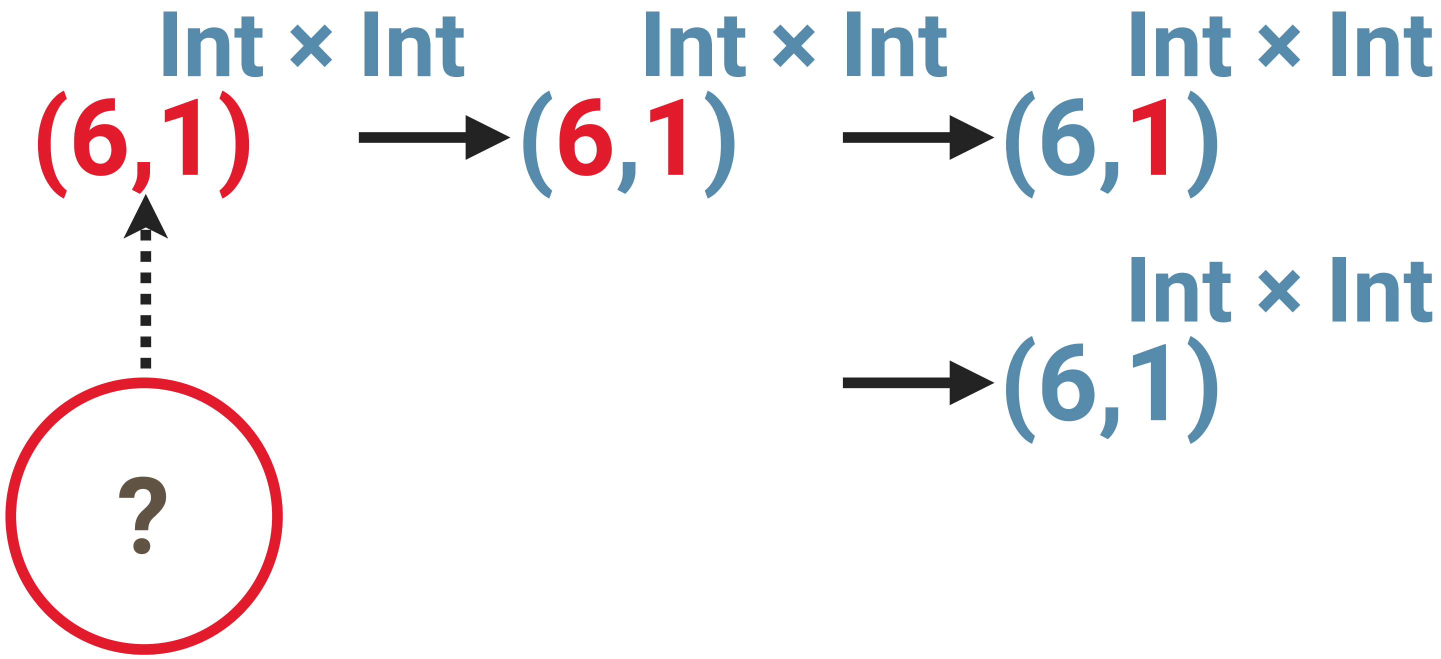


Option 2: Check



COST OF SOUNDNESS

Cost of Types ($\longrightarrow^*_{\text{slow}}$)



Int × Int

(6,1)

Int × Int

(6,1)

Int × Int

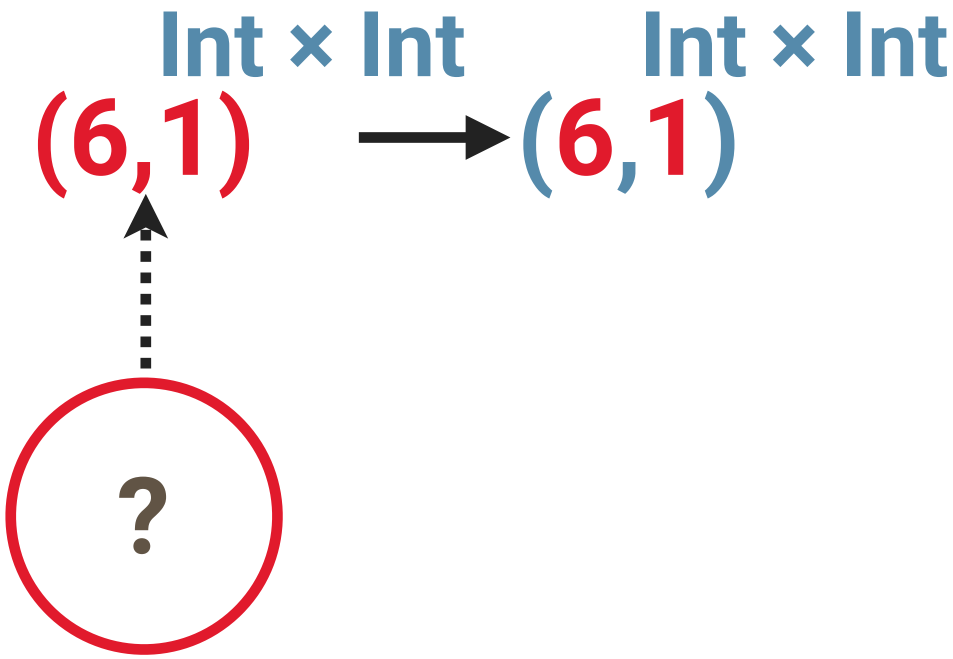
(6,1)

Int × Int

(6,1)

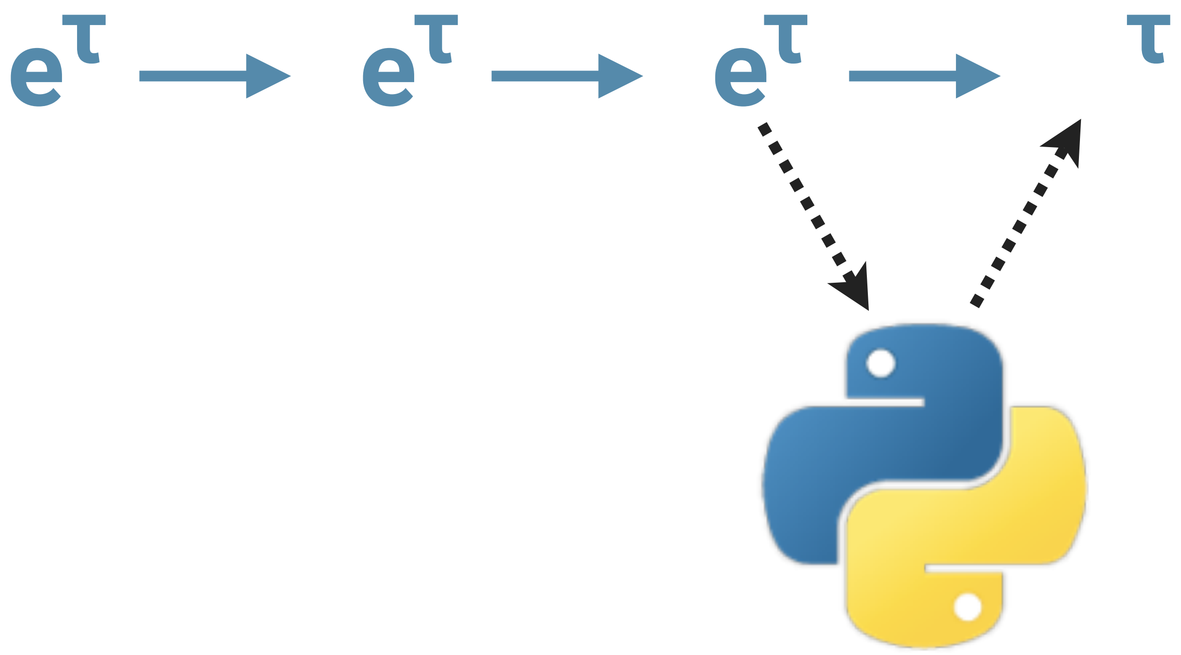
?

Cost of Tags ($\longrightarrow^*_{\text{fast}}$)



COST OF SOUNDNESS IN RETICULATED

Retic vs. Python



Reticulated

```
def dist(pt : Tuple(Int,Int)) -> Int:
    x = pt[0]
    y = pt[1]
    return abs(x + y)
```

Reticulated

```
def dist(pt : Tuple(Int,Int)) -> Int:
    x = pt[0]
    y = pt[1]
    return abs(x + y)
```

```
dist((0, 0))  →*  0
```

Reticulated

```
def dist(pt : Tuple(Int,Int)) -> Int:
    x = pt[0]
    y = pt[1]
    return abs(x + y)
```

```
    dist("NaN")   →*   Expected Tuple
```

Reticulated

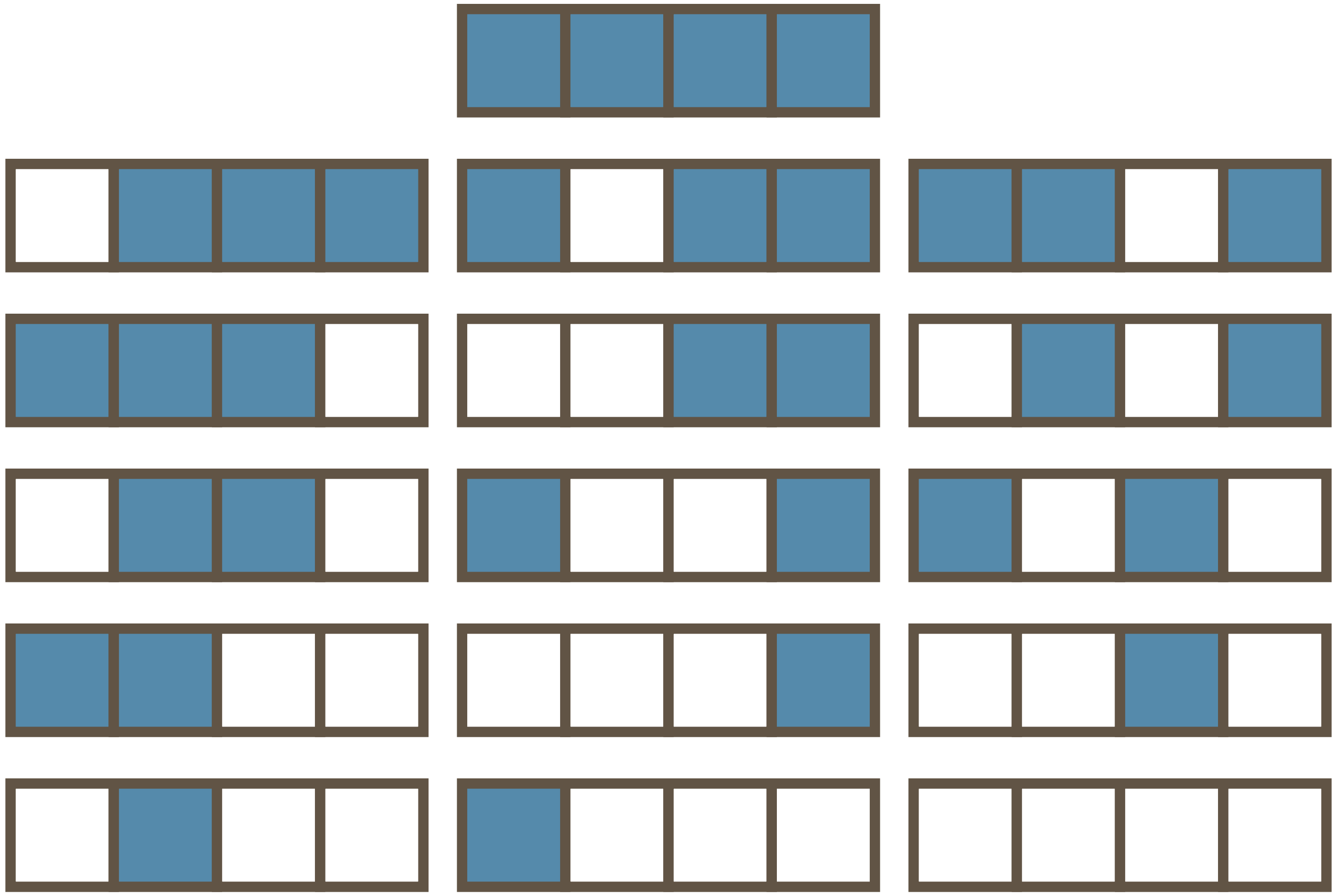
```
def dist(pt : Tuple(Int,Int)) -> Int:
    x = pt[0]
    y = pt[1]
    return abs(x + y)
```

`dist((0, "NaN"))` $\longrightarrow^*$ `Expected Int`

Evaluation Method

1. Fully-Typed

2. Configurations



3. Measure

	11s	
7s	9s	2s
5s	24s	9s
14s	5s	21s
9s	6s	9s
8s	4s	5s

What % have at most Dx overhead?

D = 4, vs.

11s

7s
5s
14s
9s
8s

9s
24s
5s
6s
4s

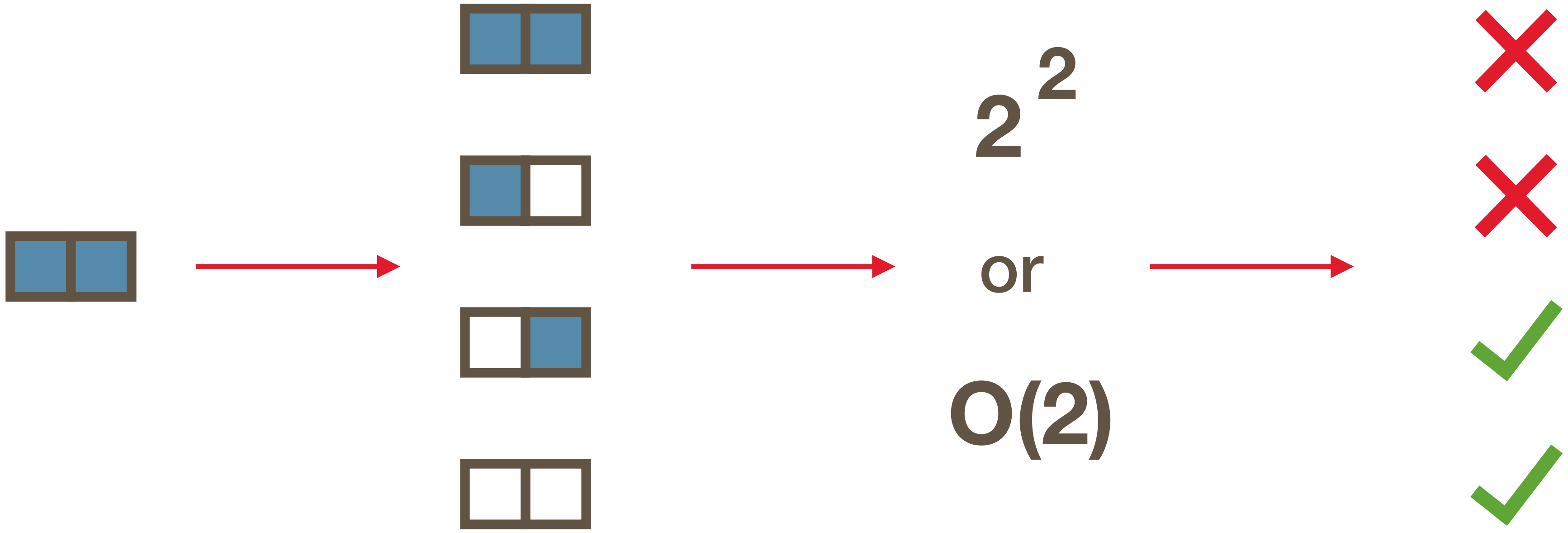
2s
9s
21s
9s
5s

What % have at most Dx overhead?

D = 4, vs.

	✗	
✓	✗	✓
✓	✗	✗
✗	✓	✗
✗	✓	✗
✓	✓	✓

Evaluation Method



2^2

or

$O(2)$

EXPERIMENT & RESULTS

Benchmarks

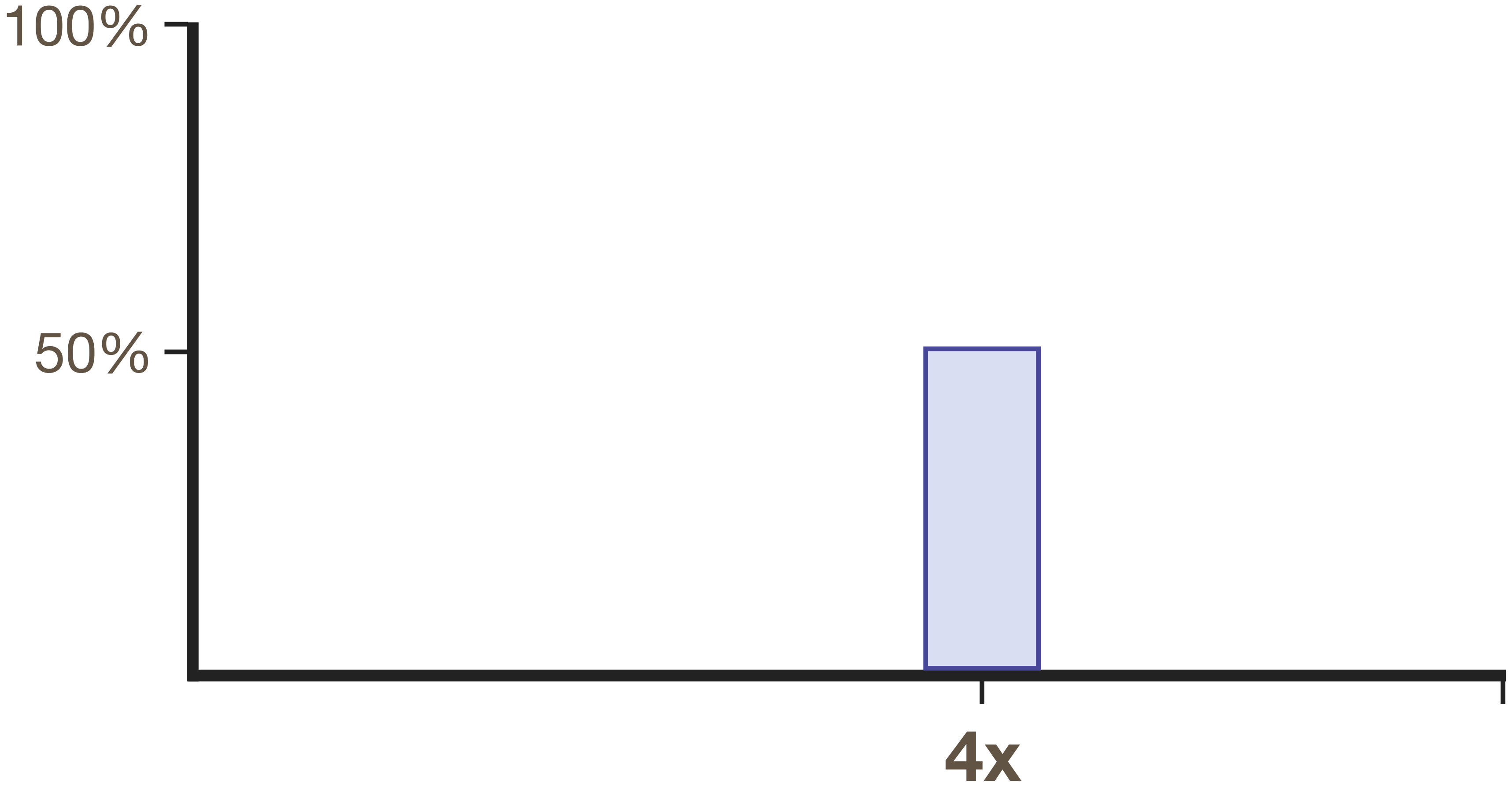
DLS 2014	POPL 2017	PEPM 2018
`futen`	`call_method`	`espionage`
`http2`	`call_simple`	`pythonflow`
`slowSHA`	`chaos`	`take5`
`aespython`	`fannkuch`	`sample_fsm`
`stats`	`go`	
	`meteor`	
	`nbody`	
	`nqueens`	
	`pidigits`	
	`pystone`	
	`spectralnorm`	

# Typed Components

DLS 2014	POPL 2017	PEPM 2018
15	7	12
4	6	12
17	15	16
34 *	1	19 *
79 *	7	
	8	
	5	
	2	
	5	
	14	
	5	

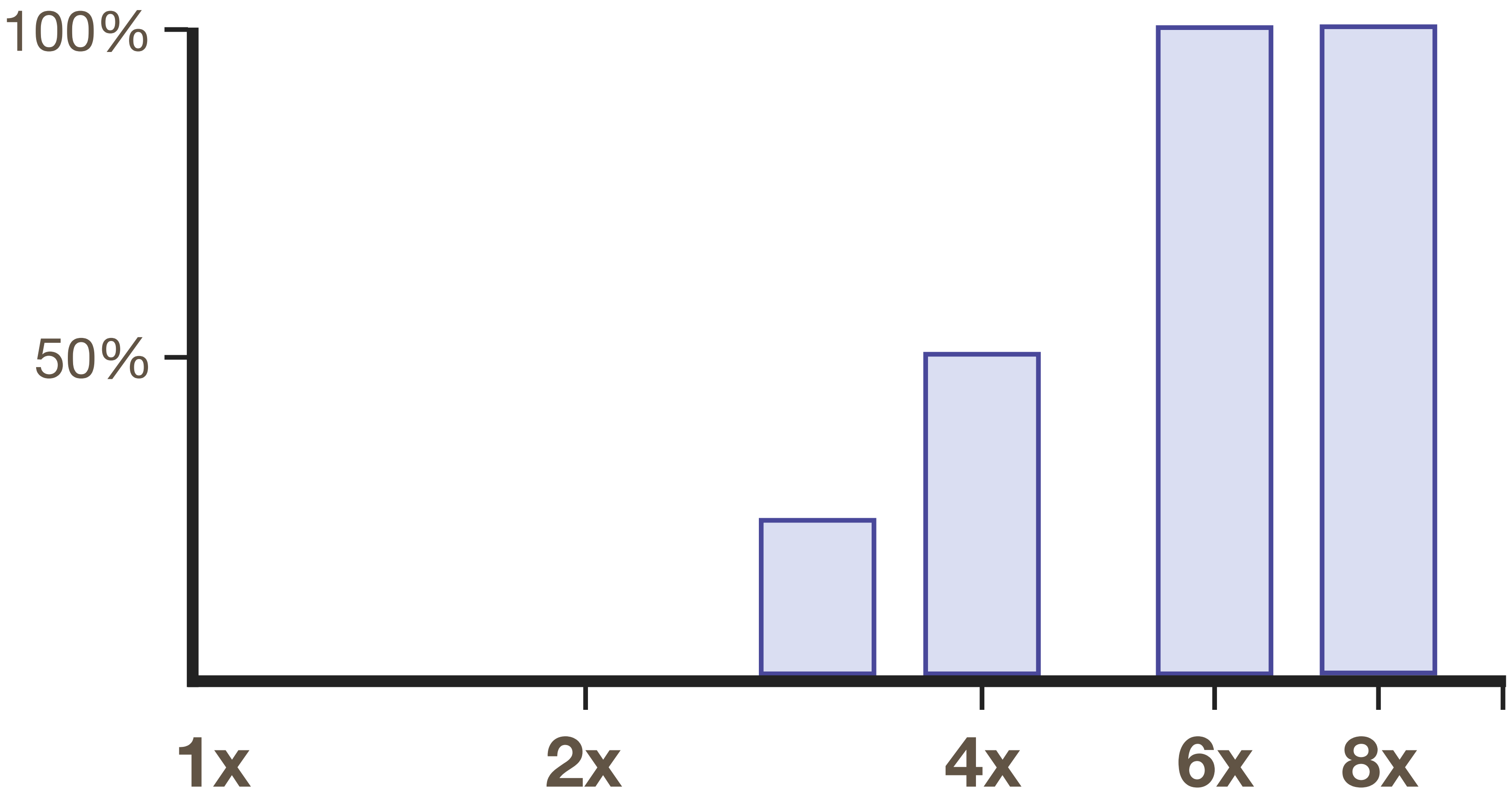
Exhaustive Results

What % of configurations have at most **4x** overhead?



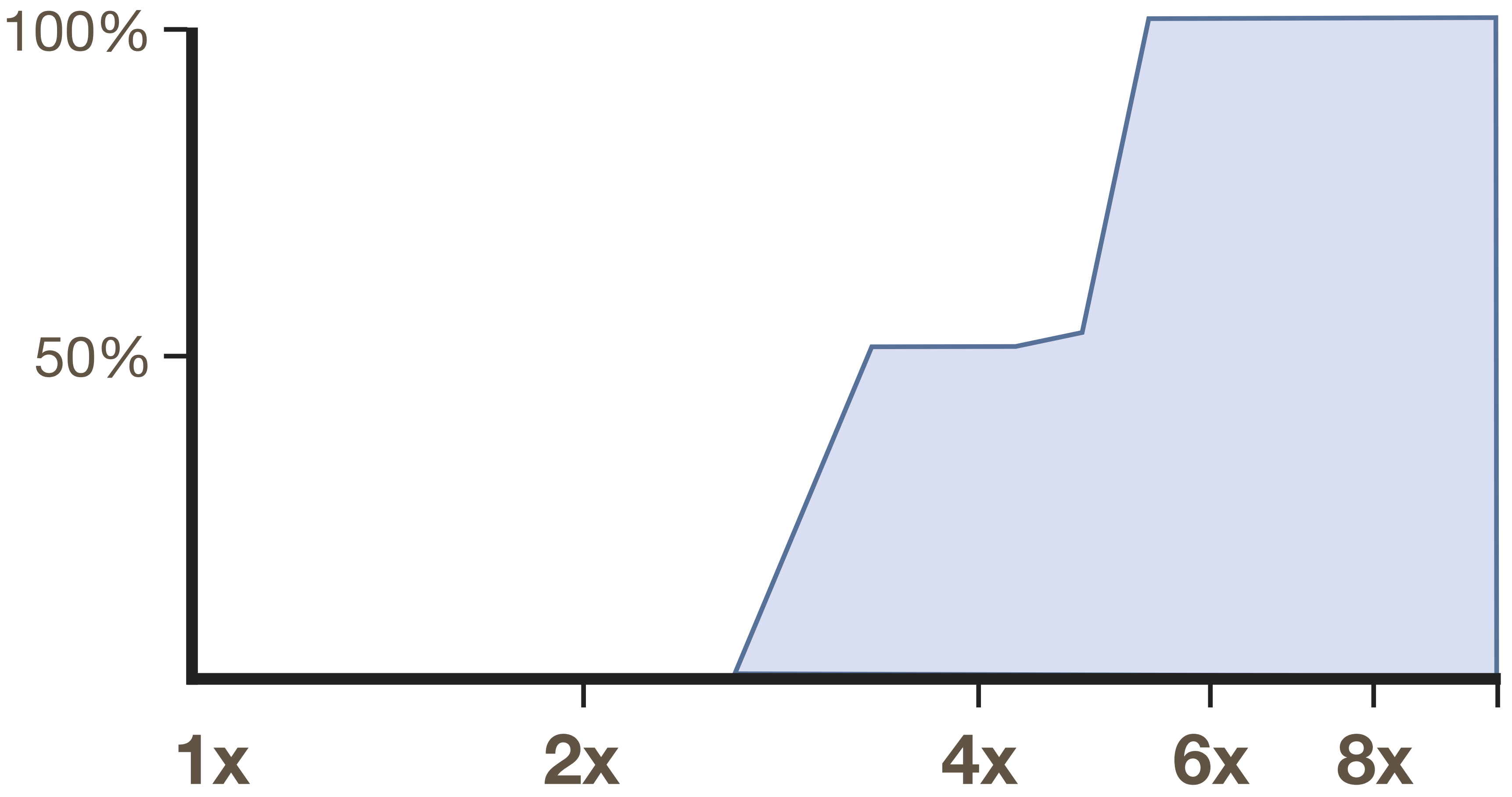
Exhaustive Results

What % of configurations have at most **Dx** overhead?



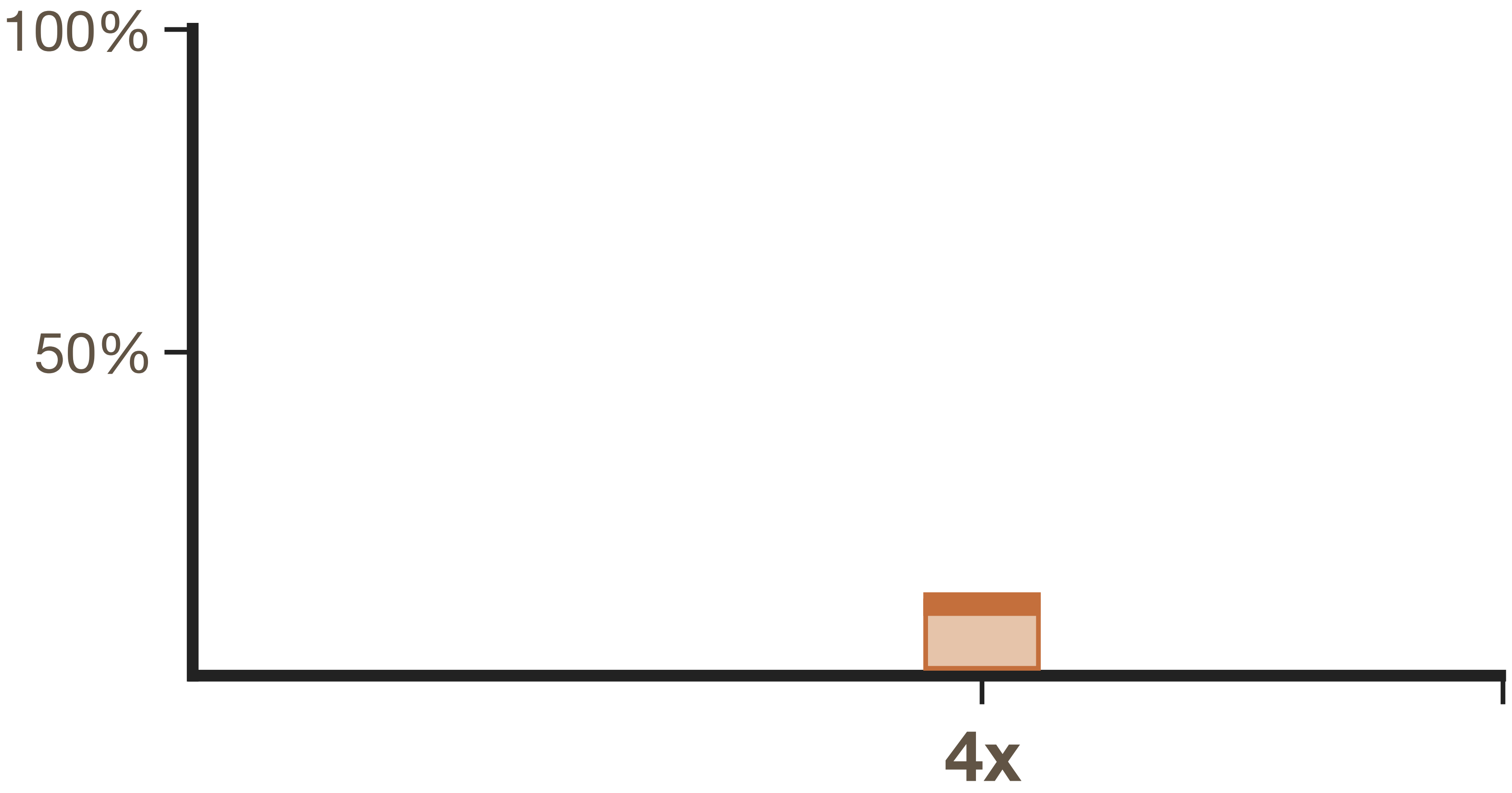
Exhaustive Results

What % of configurations have at most **Dx** overhead?



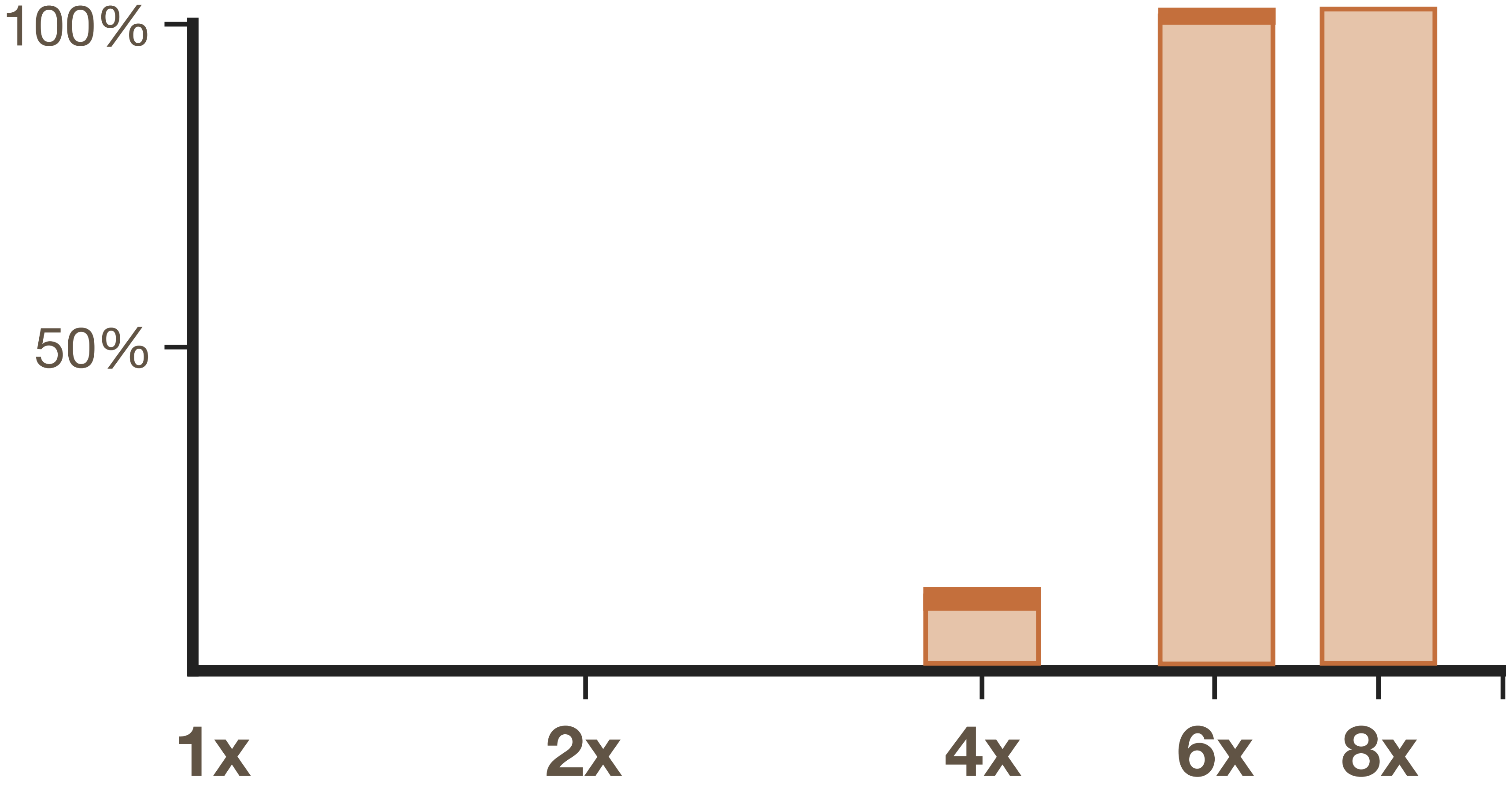
Approximate Results

What % of configurations have at most **4x** overhead, based on **R** samples of **S** configurations each?



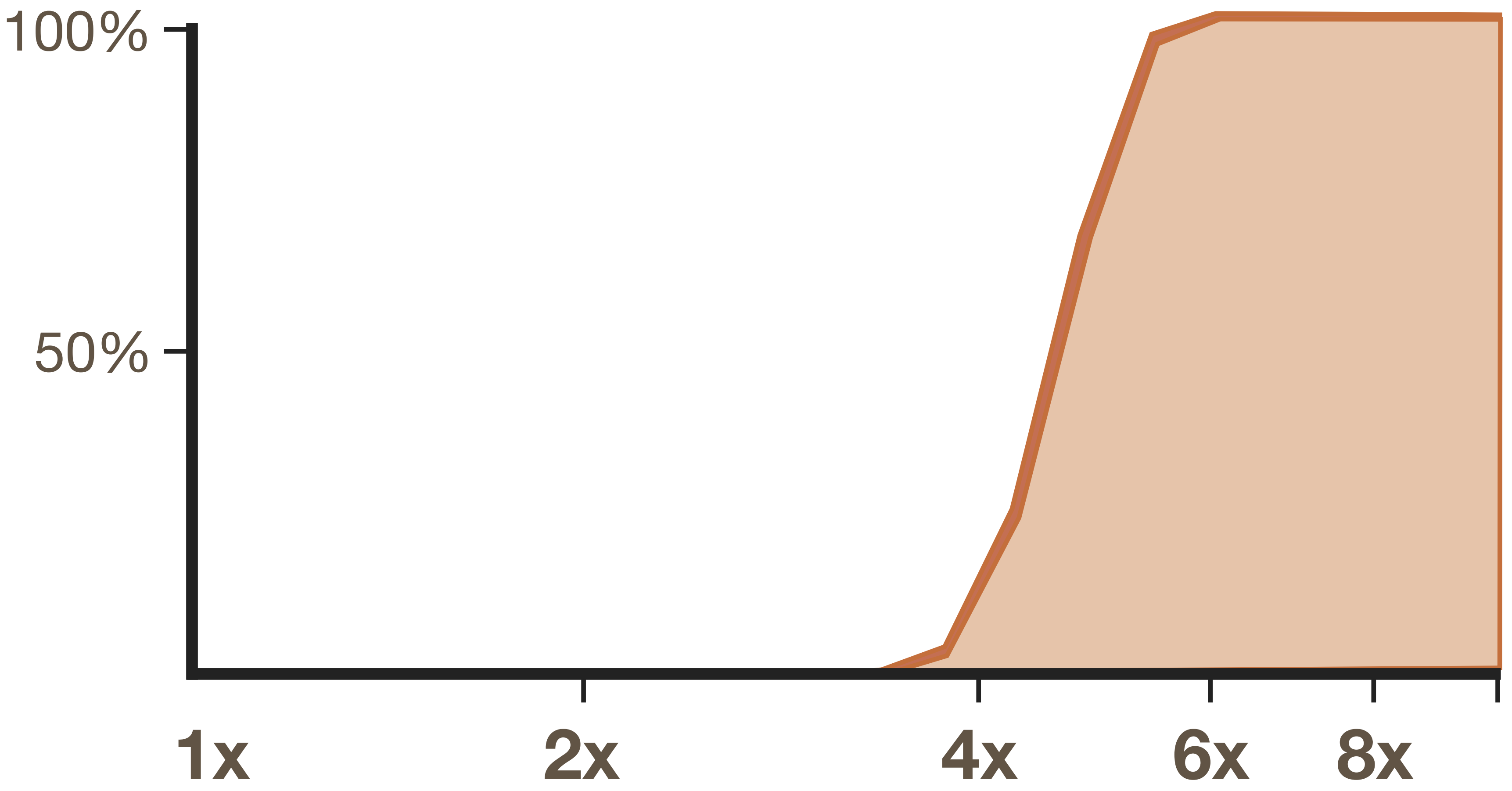
Approximate Results

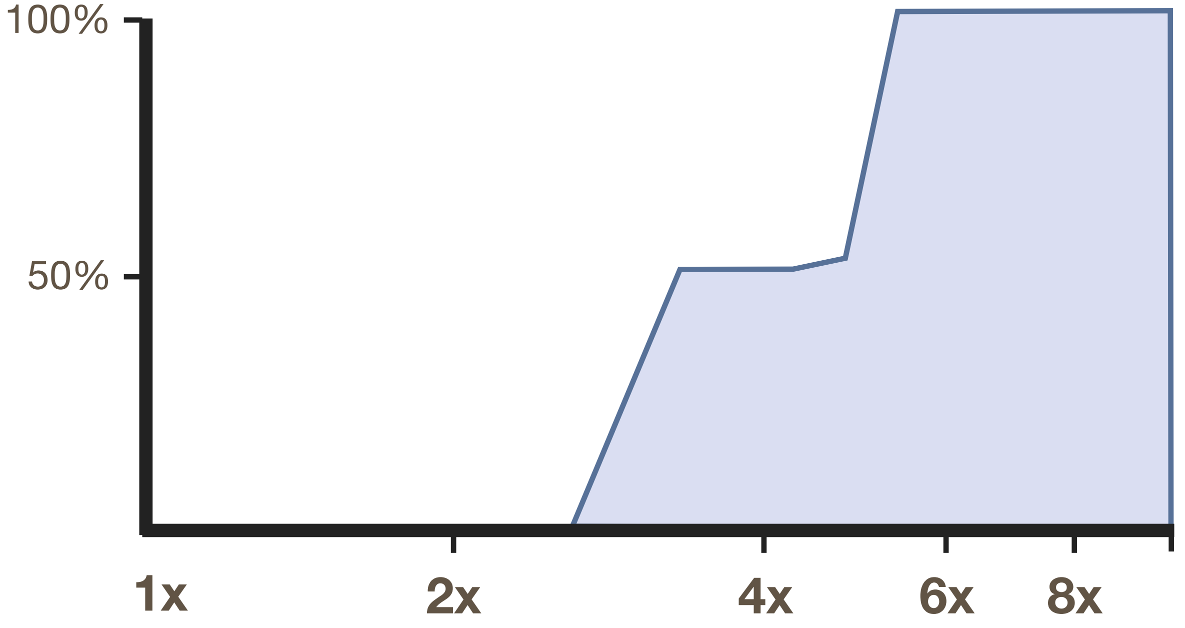
What % of configurations have at most **Dx** overhead, based on **R** samples of **S** configurations each?



Approximate Results

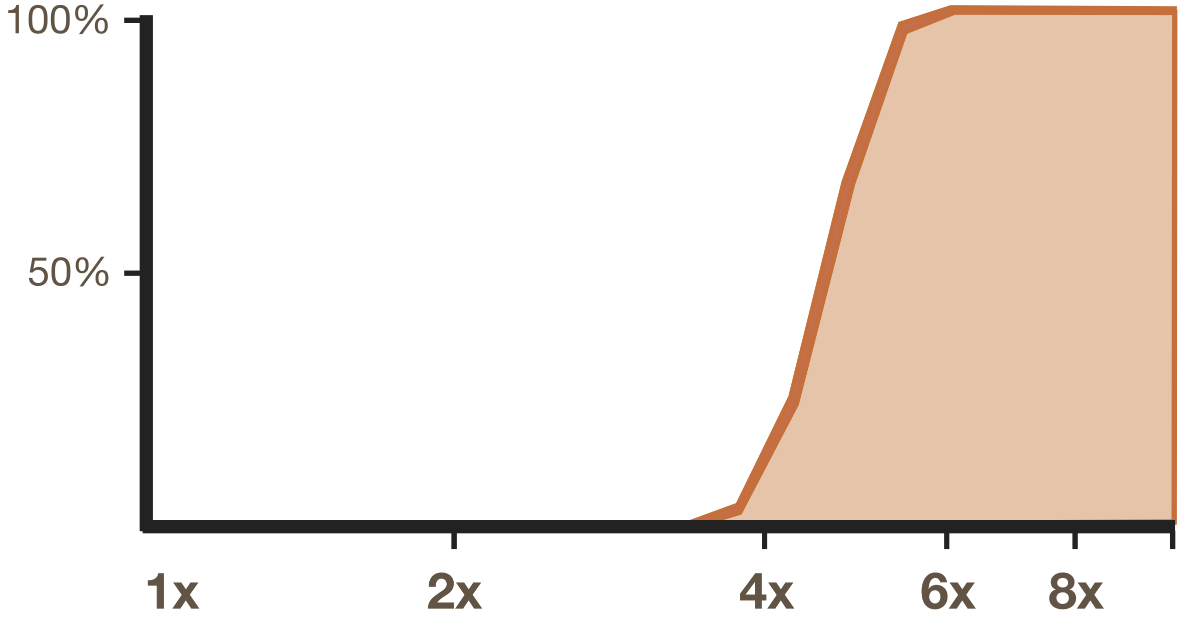
What % of configurations have at most **Dx** overhead, based on **R** samples of **S** configurations each?





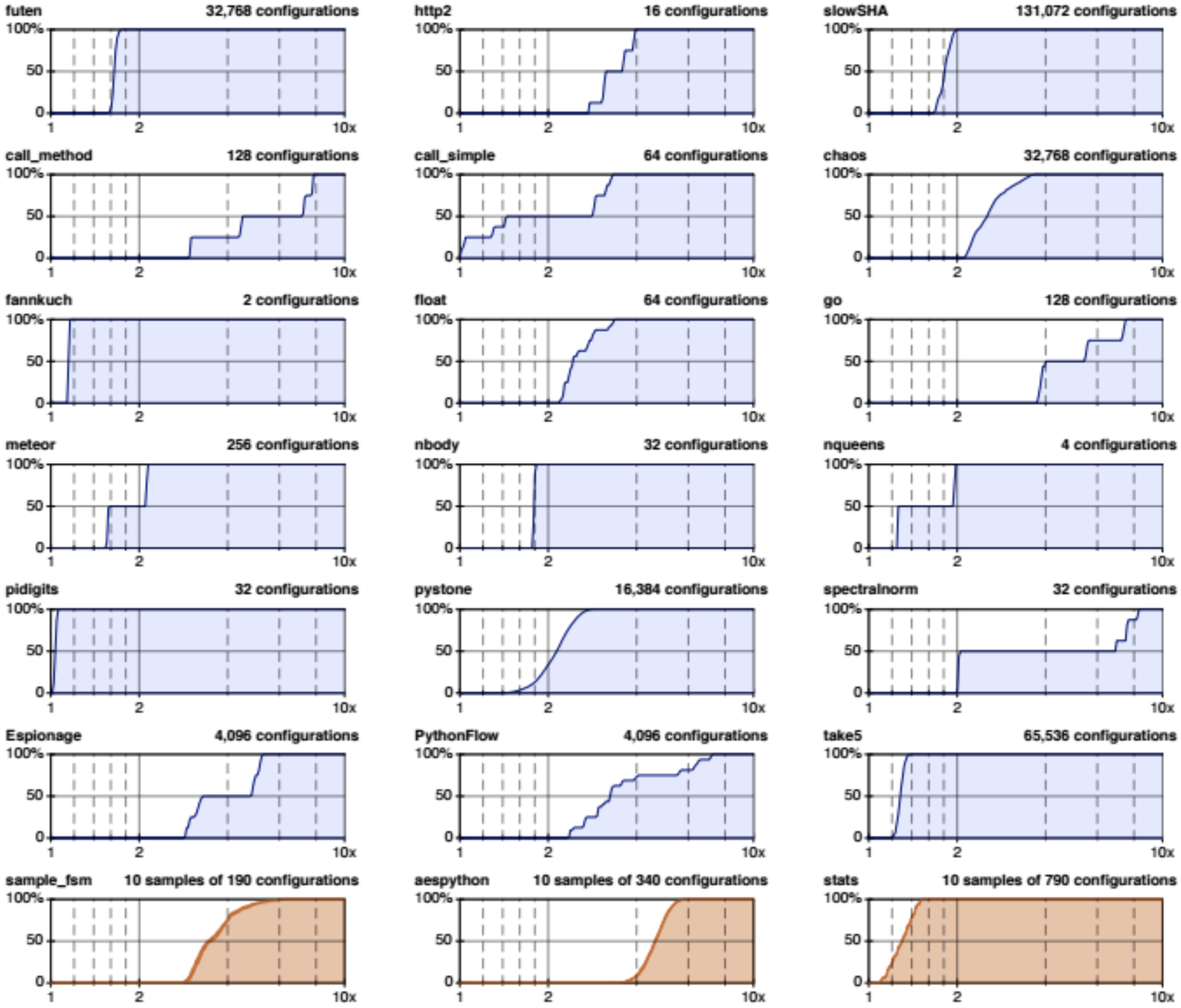
espionage

4,096 configurations



aespython

10 samples
of 340 configurations


futen
32,768 configurations
http2
16 configurations
slowSHA
131,072 configurations
call_method
128 configurations
call_simple
64 configurations
chaos
32,768 configurations
fannkuch
2 configurations
float
64 configurations
go
128 configurations
meteor
256 configurations
nbody
32 configurations
nqueens
4 configurations
pidigits
32 configurations
pystone
16,384 configurations
spectralnorm
32 configurations
Espionage
4,096 configurations
PythonFlow
4,096 configurations
take5
65,536 configurations
sample_fsm
10 samples of 190 configurations
aespython
10 samples of 340 configurations
stats
10 samples of 790 configurations
100%
50
0
1
2
10x

Cost of Tag Soundness

- Worst-case overhead: under 10x

This is an APPLES to ORANGES comparison!

Type Soundness

1x 1x

2x 4x

1x

1x

5x 5x

27x

29x

32x

10x

34x

43x 47x

233x

292x

139x

1527x

Tag Soundness

1x 1x

1x 1x

1x

3x 1x

2x 2x

2x

3x 3x

2x

5x 3x

7x

6x

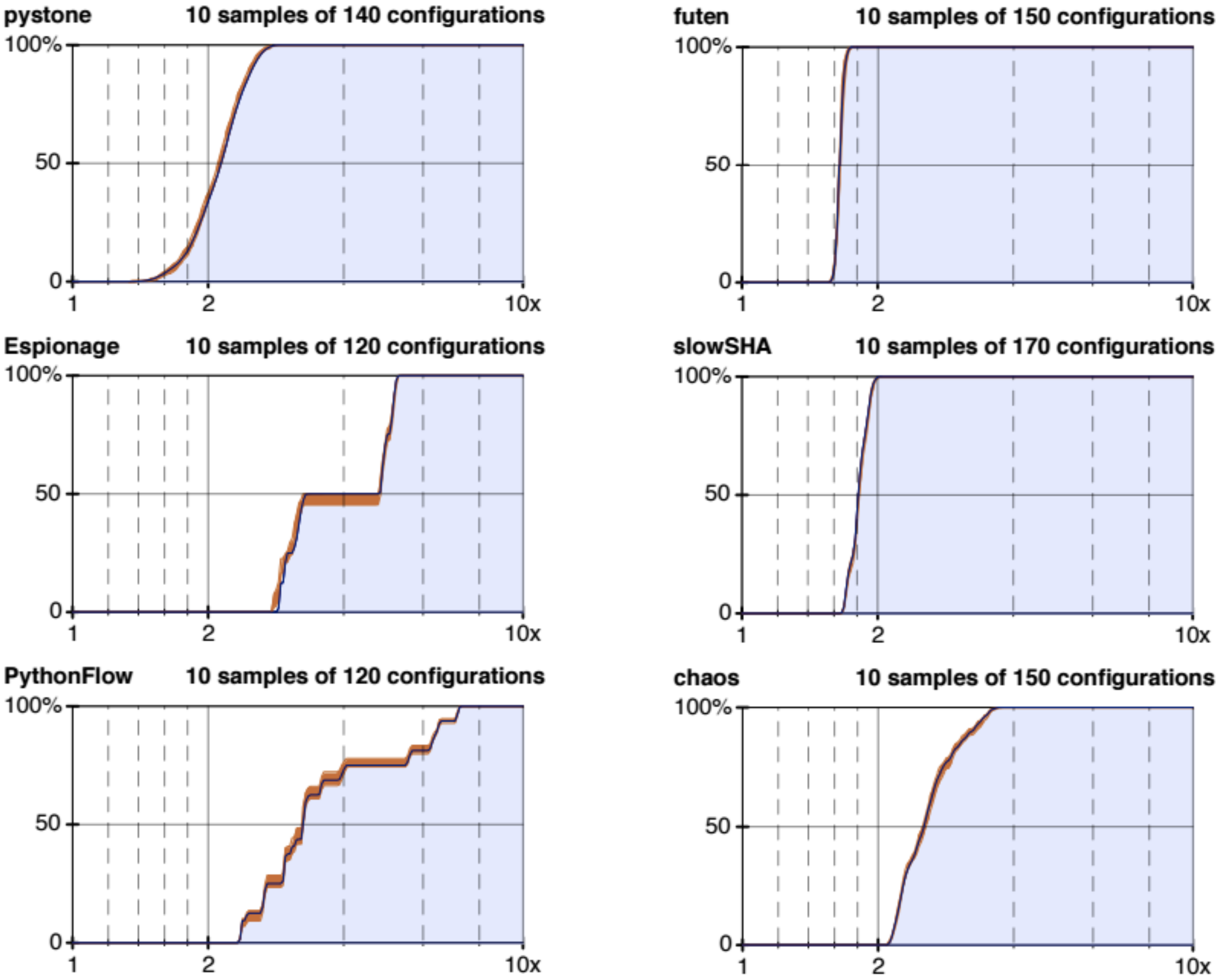
6x 7x

7x

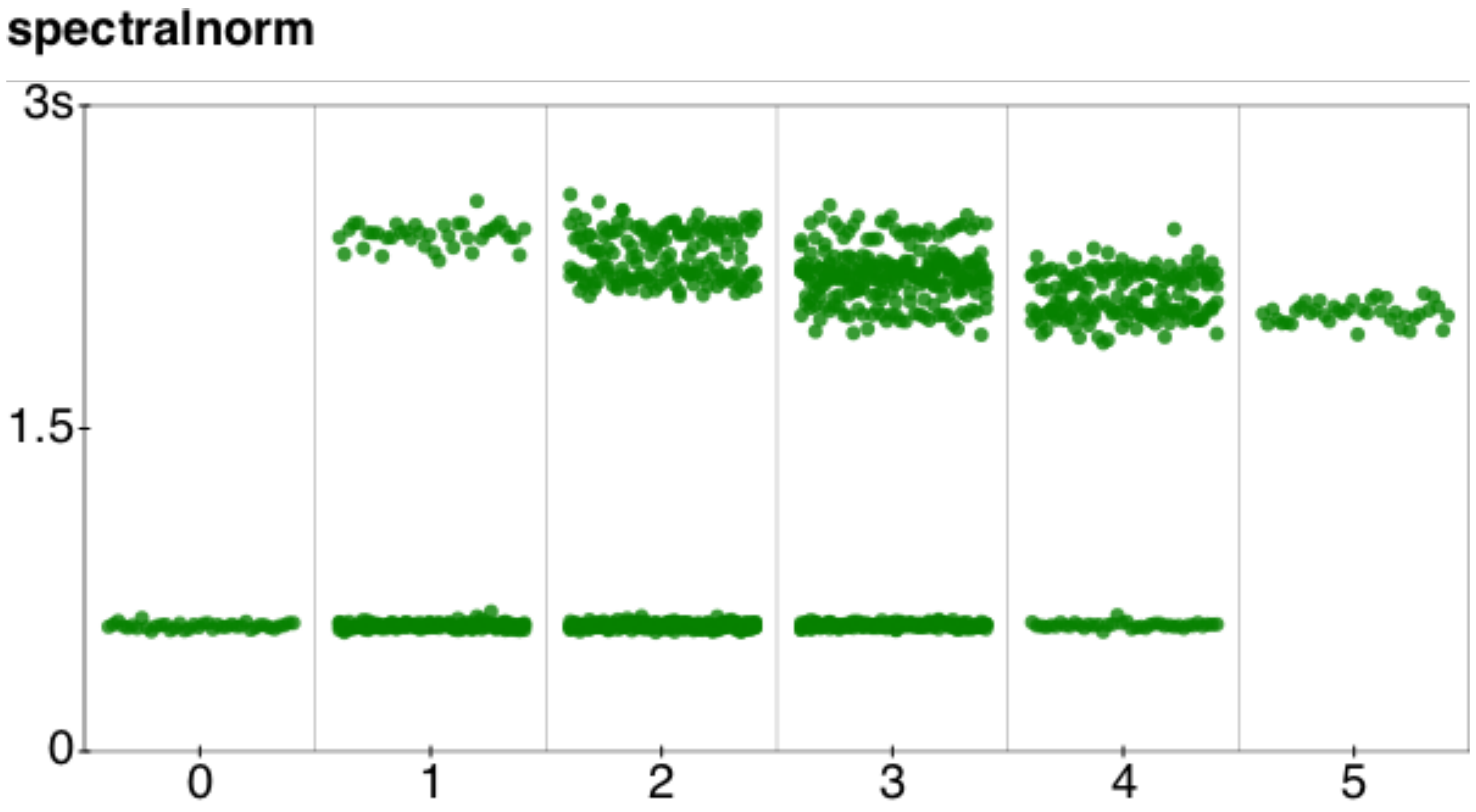
8x

Cost of Tag Soundness

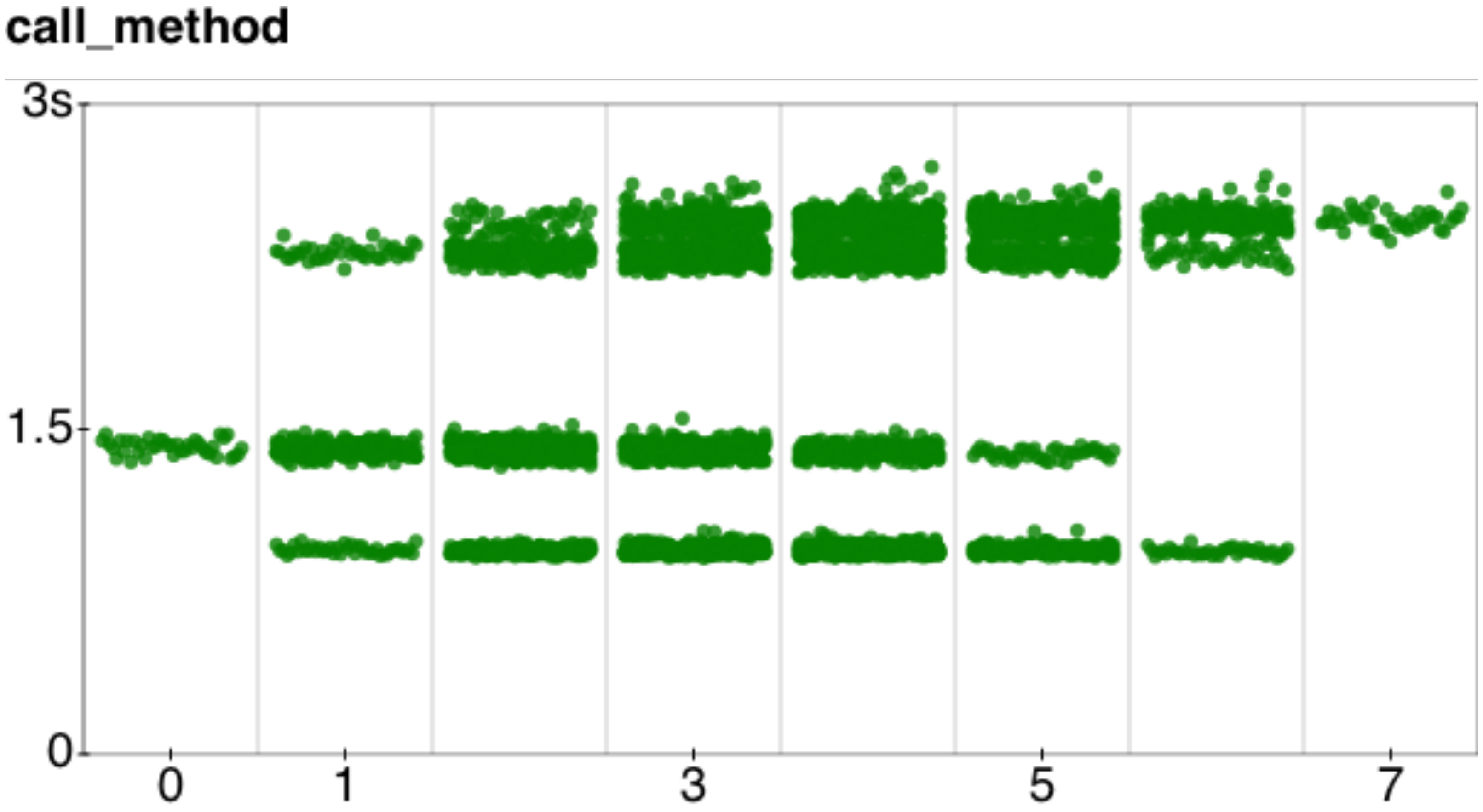
- Worst-case overhead: under 10x
- Best-case overhead: 1x -- 4x
  - adding types never* improves performance
- Slowest configuration: fully-typed
  - Overhead $\propto$ number of type annotations


pystone
10 samples of 140 configurations
futen
10 samples of 150 configurations
Espionage
10 samples of 120 configurations
slowSHA
10 samples of 170 configurations
PythonFlow
10 samples of 120 configurations
chaos
10 samples of 150 configurations
100%
50
0
1
2
10x

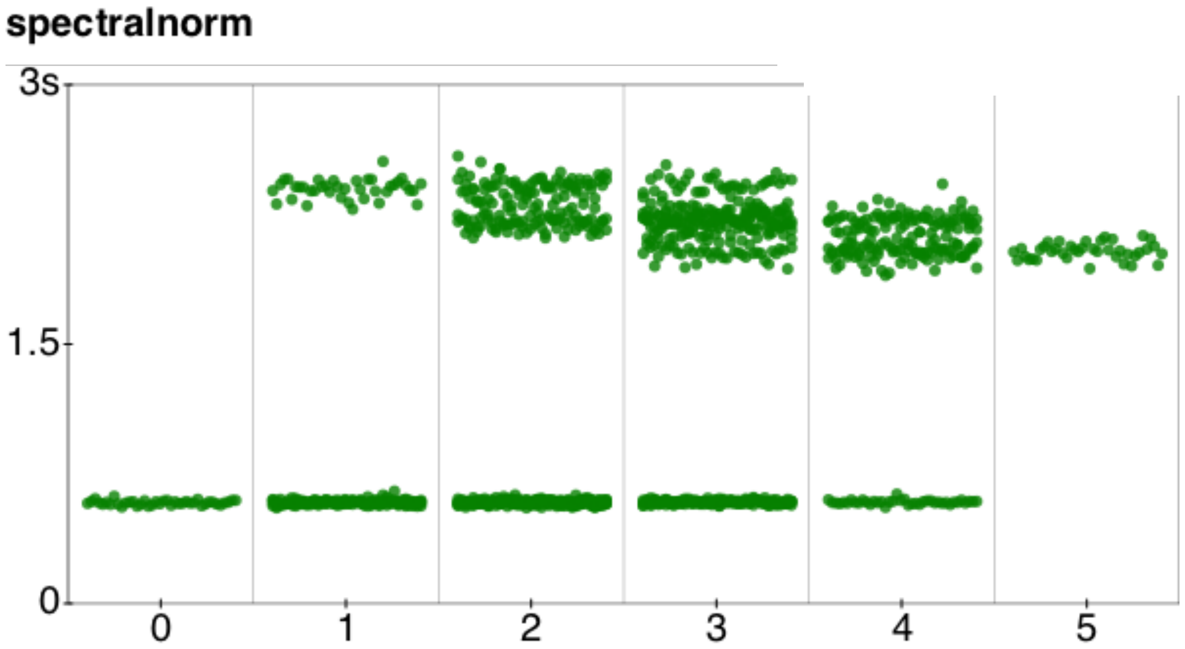
Runtime vs. # Types

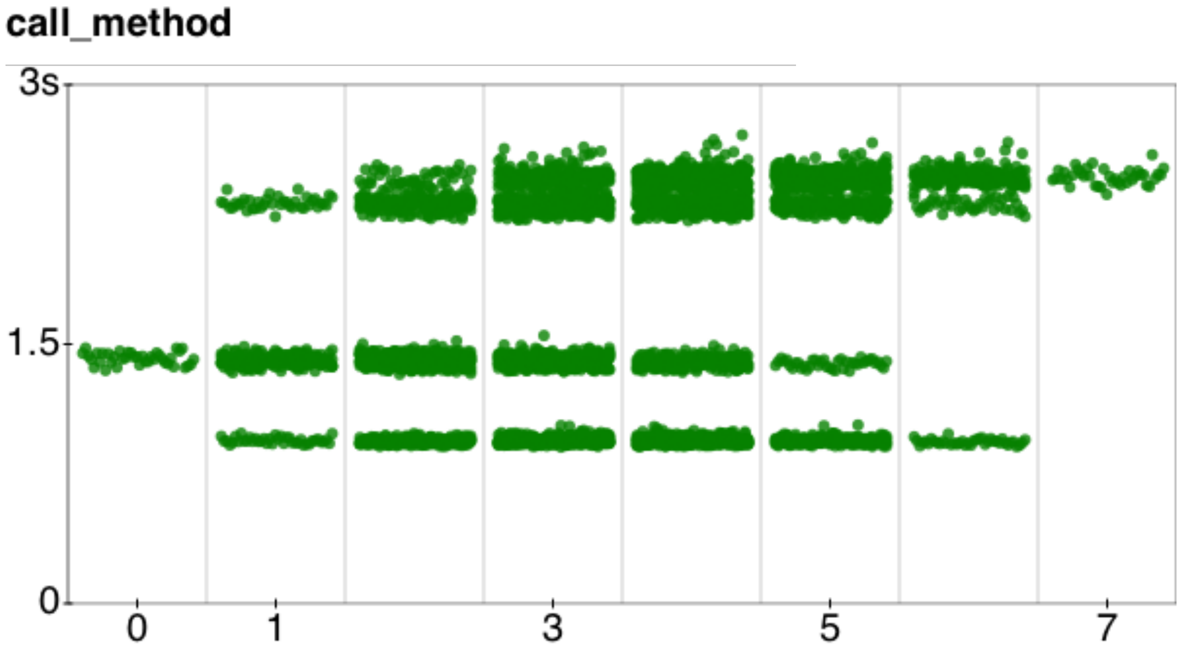


Runtime vs. # Types



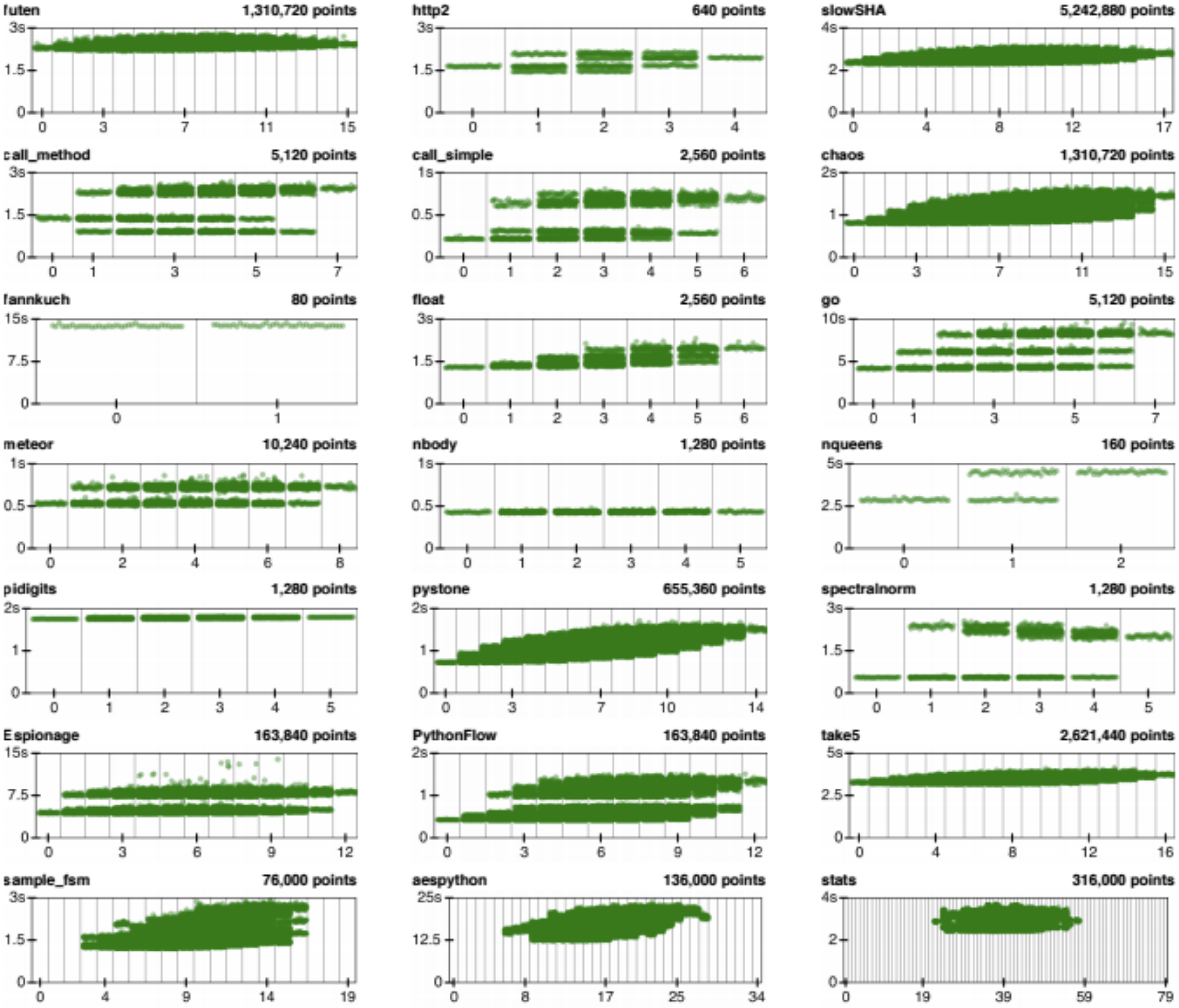
Speedup?





- Unsound optimization for read-only values (tuples)
- Double-checks method calls

Runtime vs. # Types



Experiment

- granularity: functions & class-fields
- 10 samples of [10 * (F + C)] configurations
- Karst at Indiana University cluster (32GB RAM, 250GB other)
- Reticulated, master branch, commit e478343
- Python 3.4.3
- 40 iterations per configuration, report average
- 200 values of D on x-axis

POPL 2017

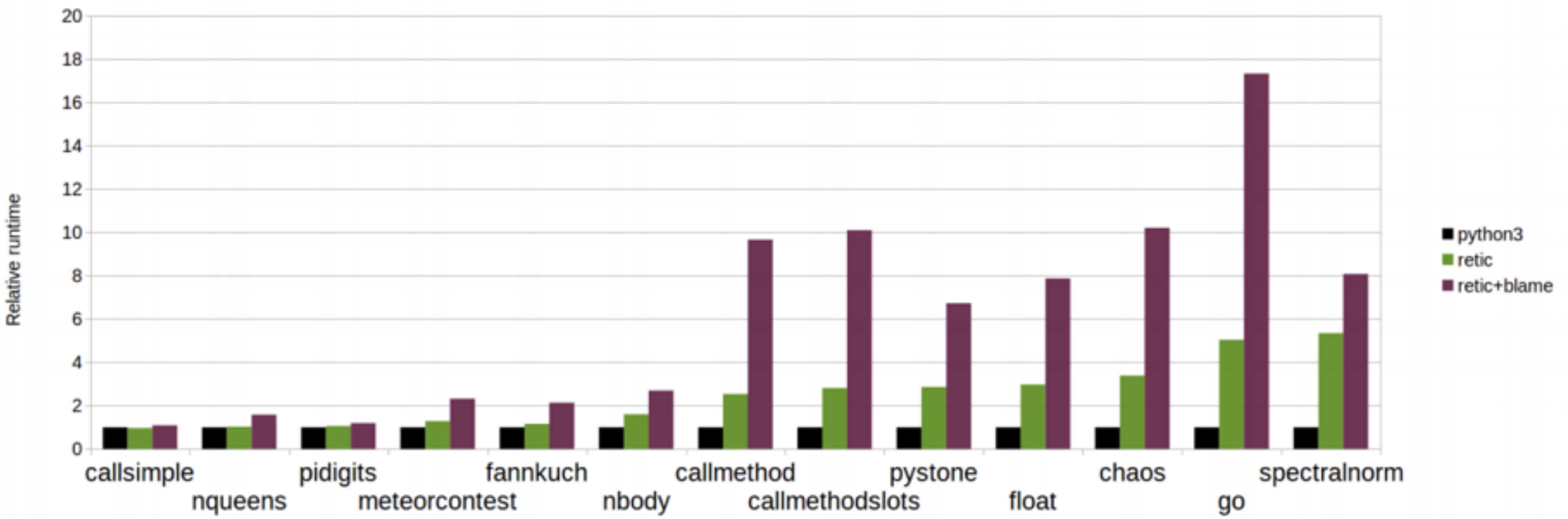
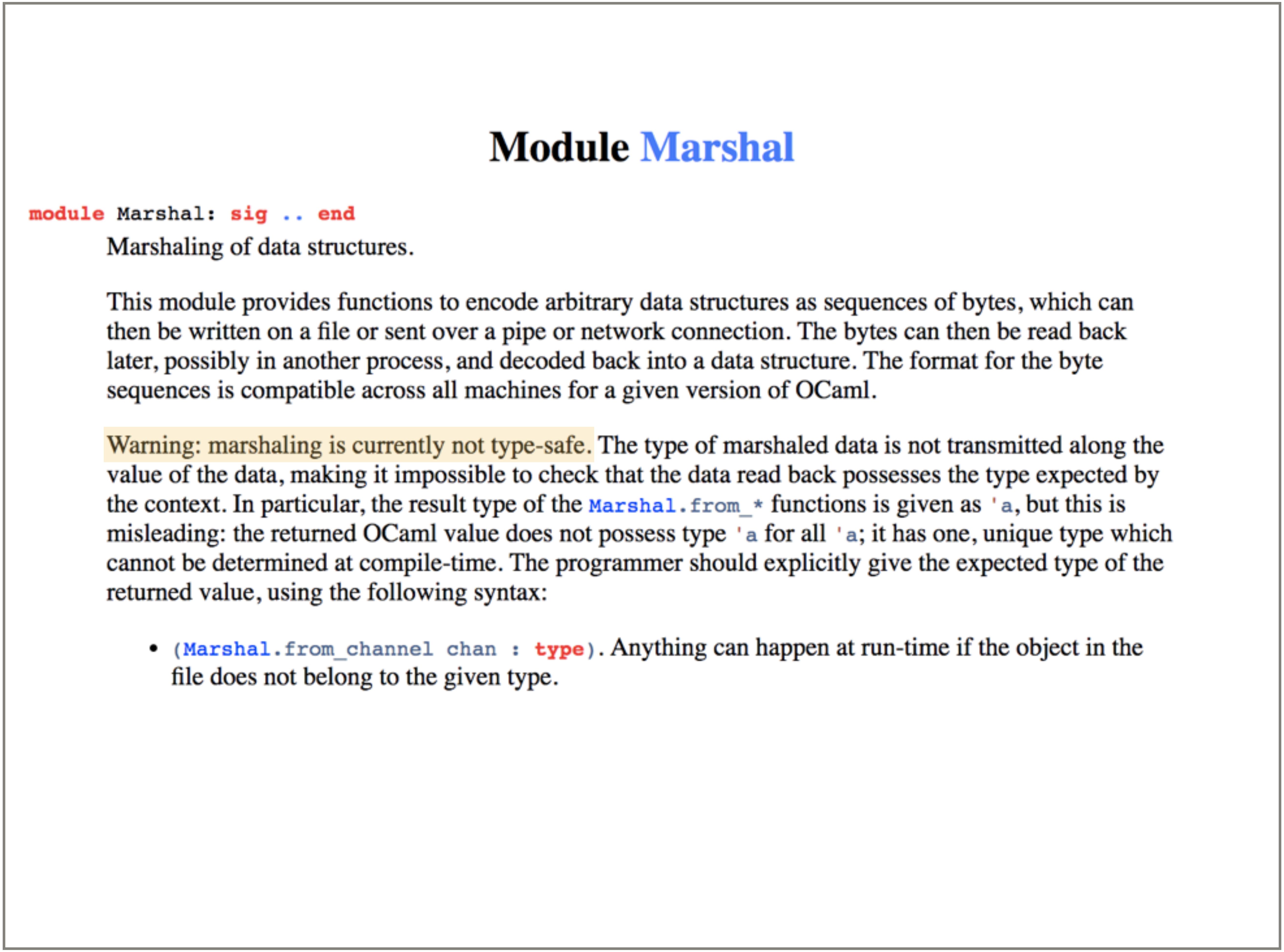


Figure 11. Runtime comparison of Reticulated Python to standard Python 3.4. Experiments were performed on an Ubuntu 14.04 laptop with a 2.8GHz Intel i7-3840QM CPU and 16GB memory.



Module Marshal

```
module Marshal: sig .. end
```

Marshaling of data structures.

This module provides functions to encode arbitrary data structures as sequences of bytes, which can then be written on a file or sent over a pipe or network connection. The bytes can then be read back later, possibly in another process, and decoded back into a data structure. The format for the byte sequences is compatible across all machines for a given version of OCaml.

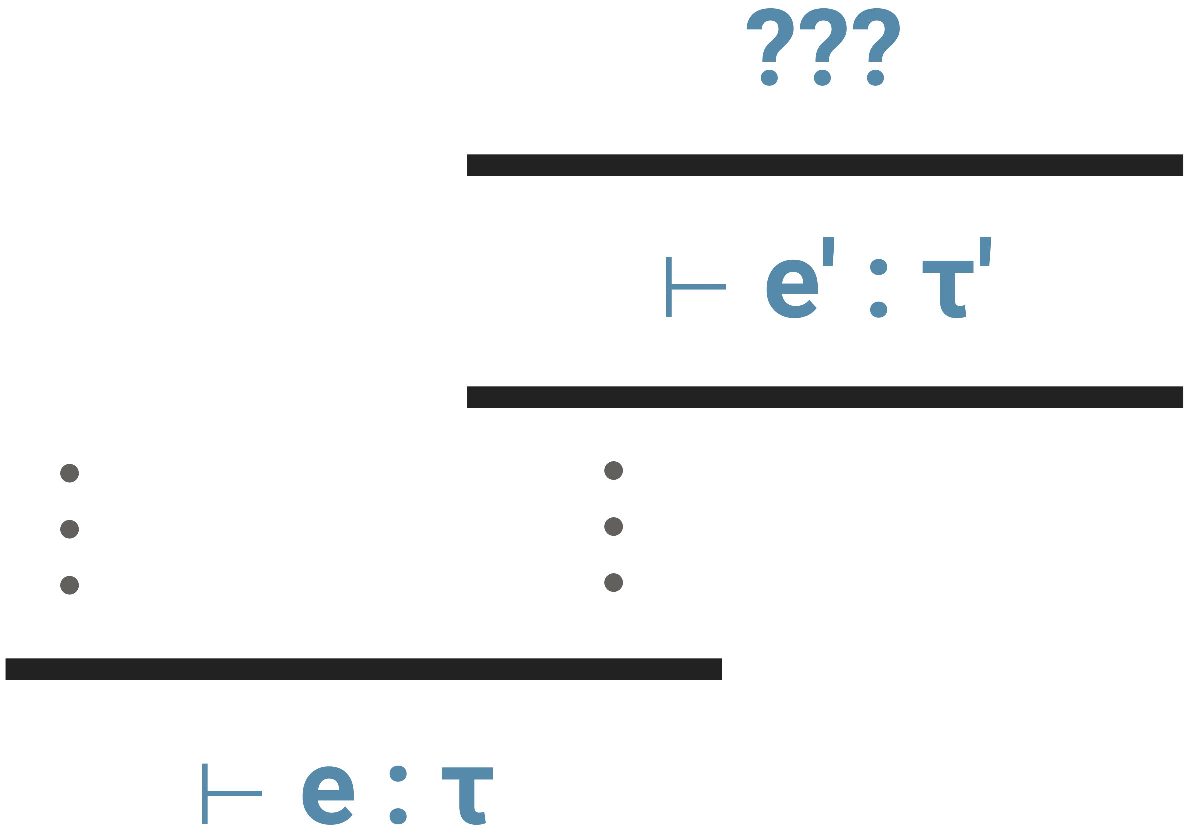
Warning: marshaling is currently not type-safe. The type of marshaled data is not transmitted along the value of the data, making it impossible to check that the data read back possesses the type expected by the context. In particular, the result type of the `Marshal.from_*` functions is given as `'a`, but this is misleading: the returned OCaml value does not possess type `'a` for all `'a`; it has one, unique type which cannot be determined at compile-time. The programmer should explicitly give the expected type of the returned value, using the following syntax:

- `(Marshal.from_channel chan : type)`. Anything can happen at run-time if the object in the file does not belong to the given type.

References

- Vitousek, Swords, Siek. *Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems.* POPL 2017
- Takikawa, Feltey, Greenman, New, Vitek, Felleisen. *Is Sound Gradual Typing Dead?* POPL 2016.

UNUSED SLIDES



???

$\vdash e' : \tau'$

$\vdash e : \tau$

Takikawa Method

- granularity
- experimental modules, fixed modules
- configurations
- baseline
- performance ratio