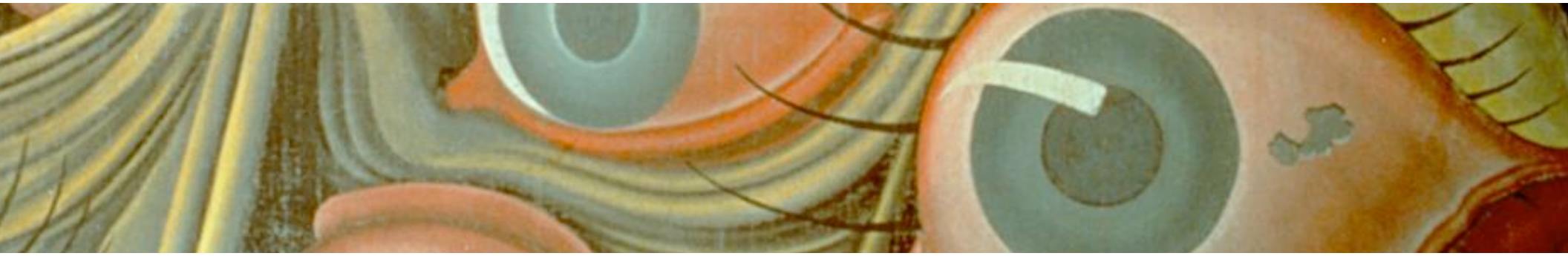
Performance Part I

why we measure things


THORN

how dynamic is dynamic



Richards, Lebresne, Burg, Vitek, **An Analysis of the Dynamic Behavior of JavaScript Programs**. PLDI'10

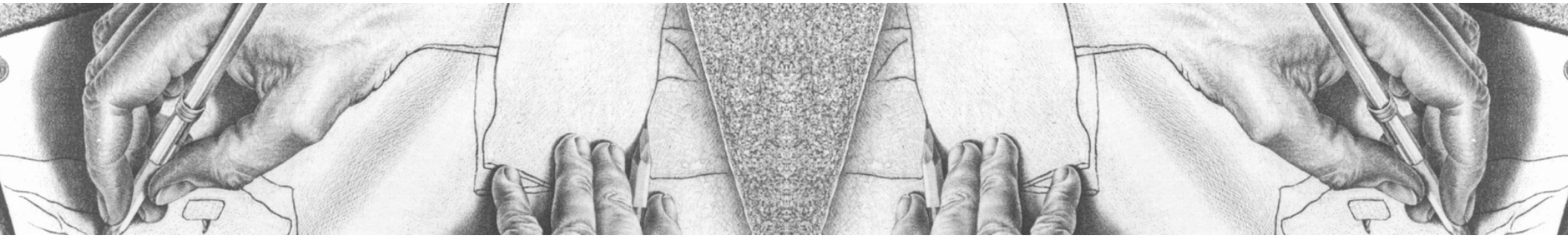
Corpus

Traced Alexa top 100

8GB of trace data

500MB distilled DB

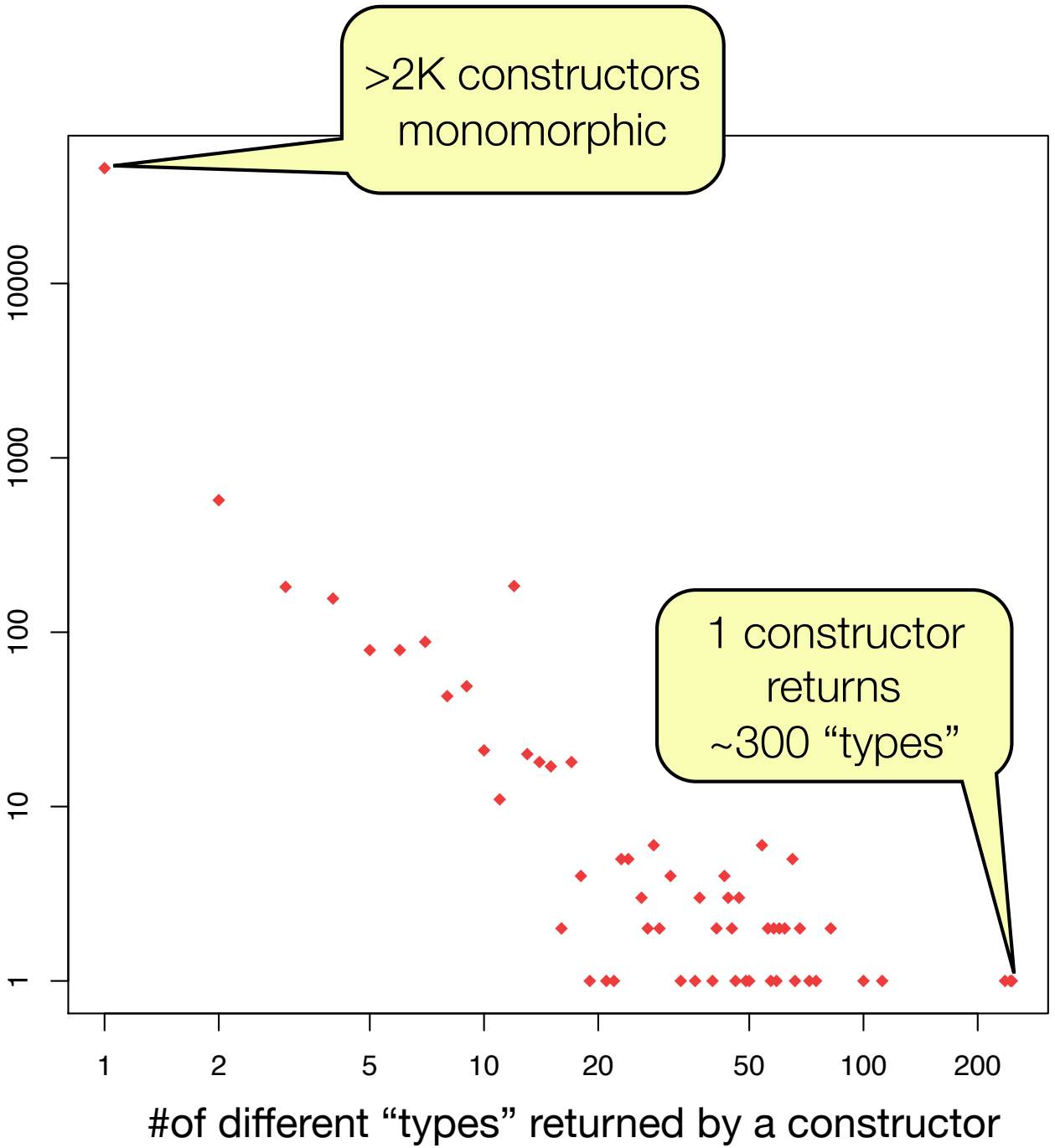
Alias	Library	URL
280S	Objective-J[1]	280slides.com
BING		bing.com
BLOG		blogger.com
DIGG	jQuery[2]	digg.com
EBAY		ebay.com
FBOK		facebook.com
FLKR		flickr.com
GMAP	Closure[3]	maps.google.com
GMIL	Closure	gmail.com
GOGL	Closure	google.com
ISHK	Prototype[4]	imageshack.us
LIVE		research.sun.com/p
MECM	SproutCore[5]	me.com
TWIT	jQuery	twitter.com
WIKI		wikipedia.com
WORD	jQuery	wordpress.com
YTUB		youtube.com
ALL		Average over 103 sites



Constructor Return “type”

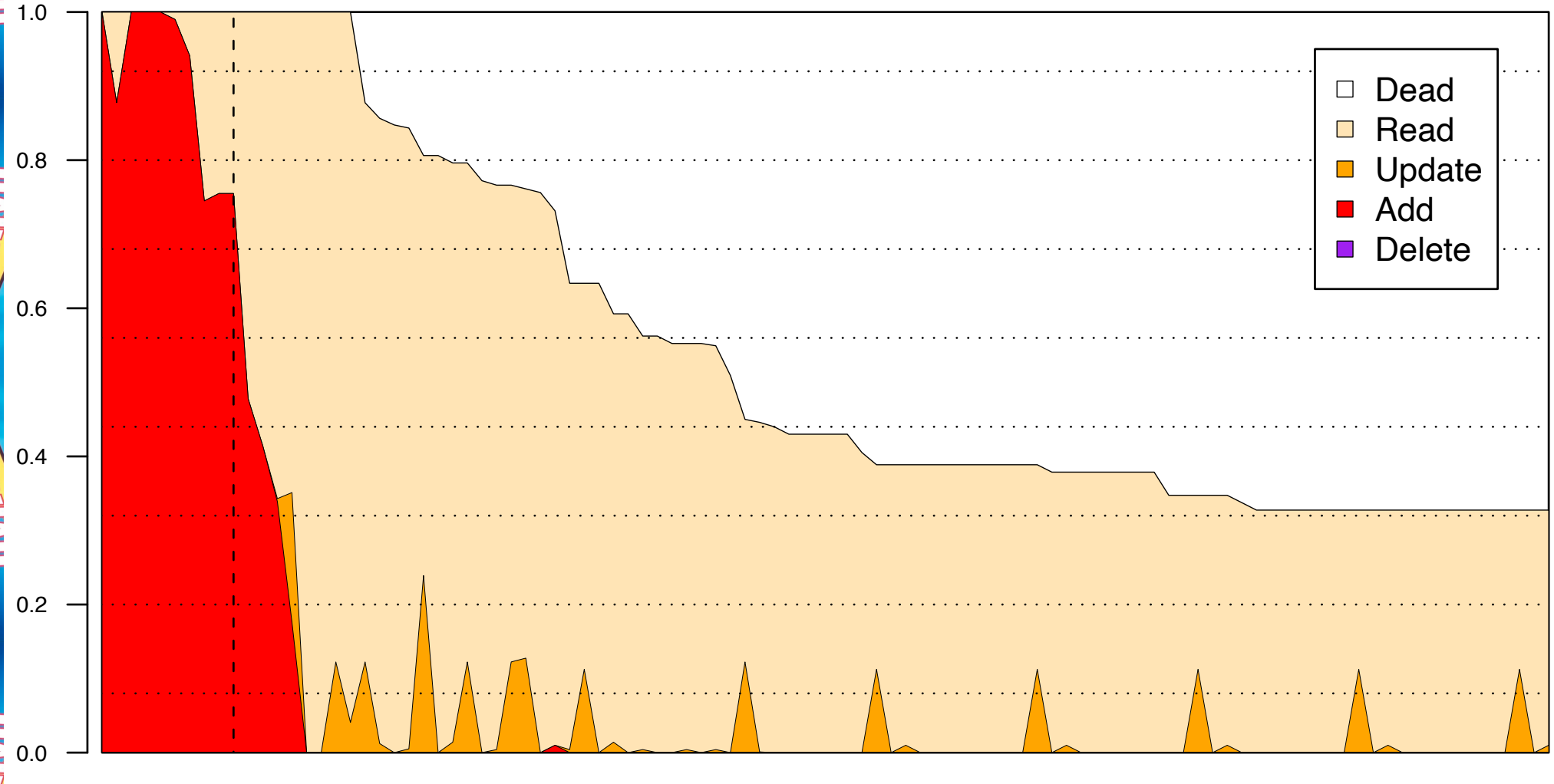
- Constructors are “just” functions that side-effect **`this`**.
- Accordingly a constructor can return different “types”, i.e. objects with different properties

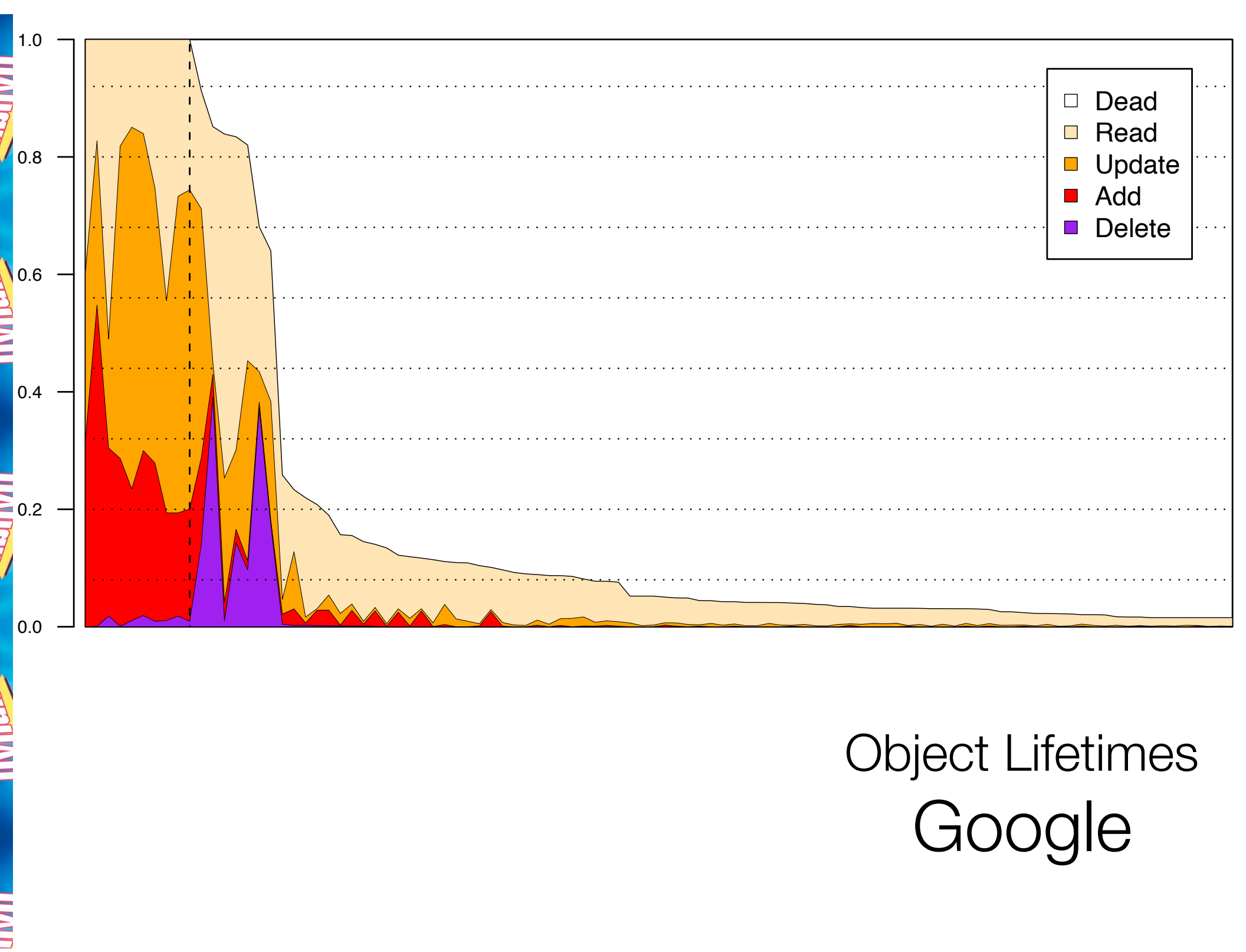
```
function Person(n,M){
  this.name=n;
  this.sex=M;
  if(M){
   this.likes= “guns”
  }
}
```



Object Lifetimes

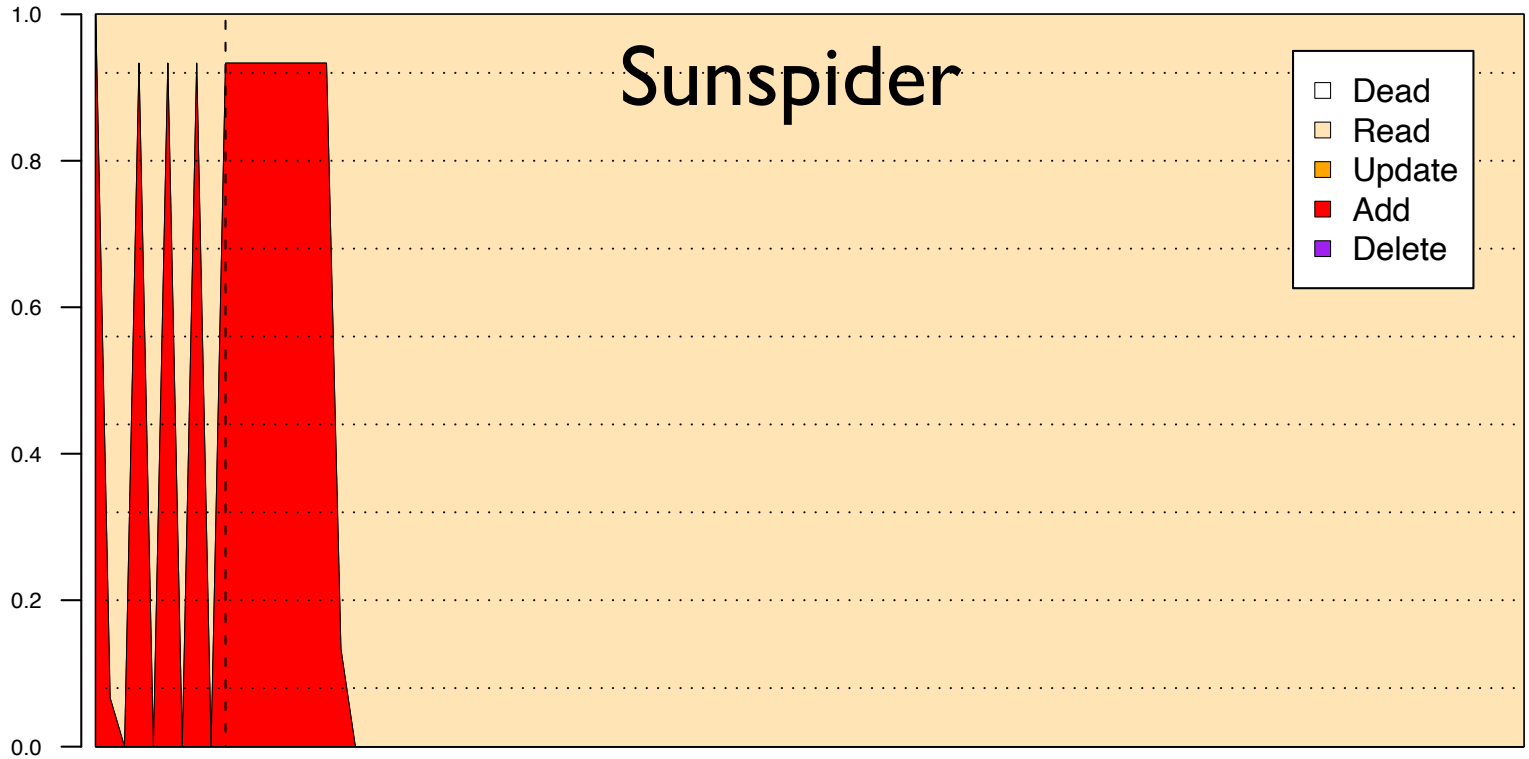
Twitter

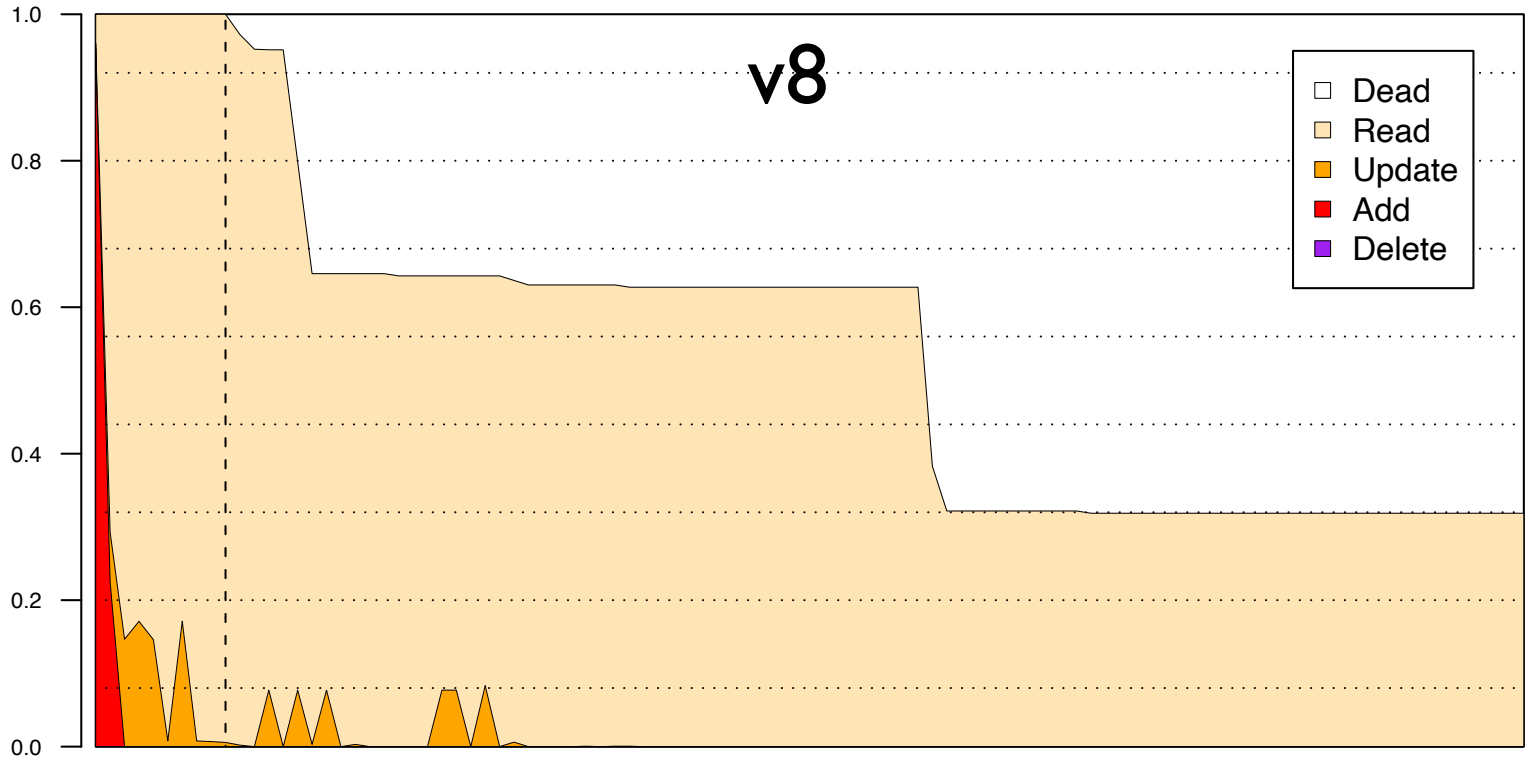




Object Lifetimes

Google


Sunspider
Dead
Read
Update
Add
Delete
1.0
0.8
0.6
0.4
0.2
0.0


v8
Dead
Read
Update
Add
Delete
1.0
0.8
0.6
0.4
0.2
0.0

benchmarks for free

Richards, Gal, Eich, Vitek. **JSBench: Automating the Construction of JavaScript Benchmarks.** OOPSLA'11

JS Benchmarks

Sunspider, V8, etc.

- Consistent behavior
- Easy to use
- Long-running (in principle)
- Wildly unrealistic

Real web pages

- Browser-dependent
- Difficult to automate
- Short-running (actual compute time)
- Representative by definition

JSBench

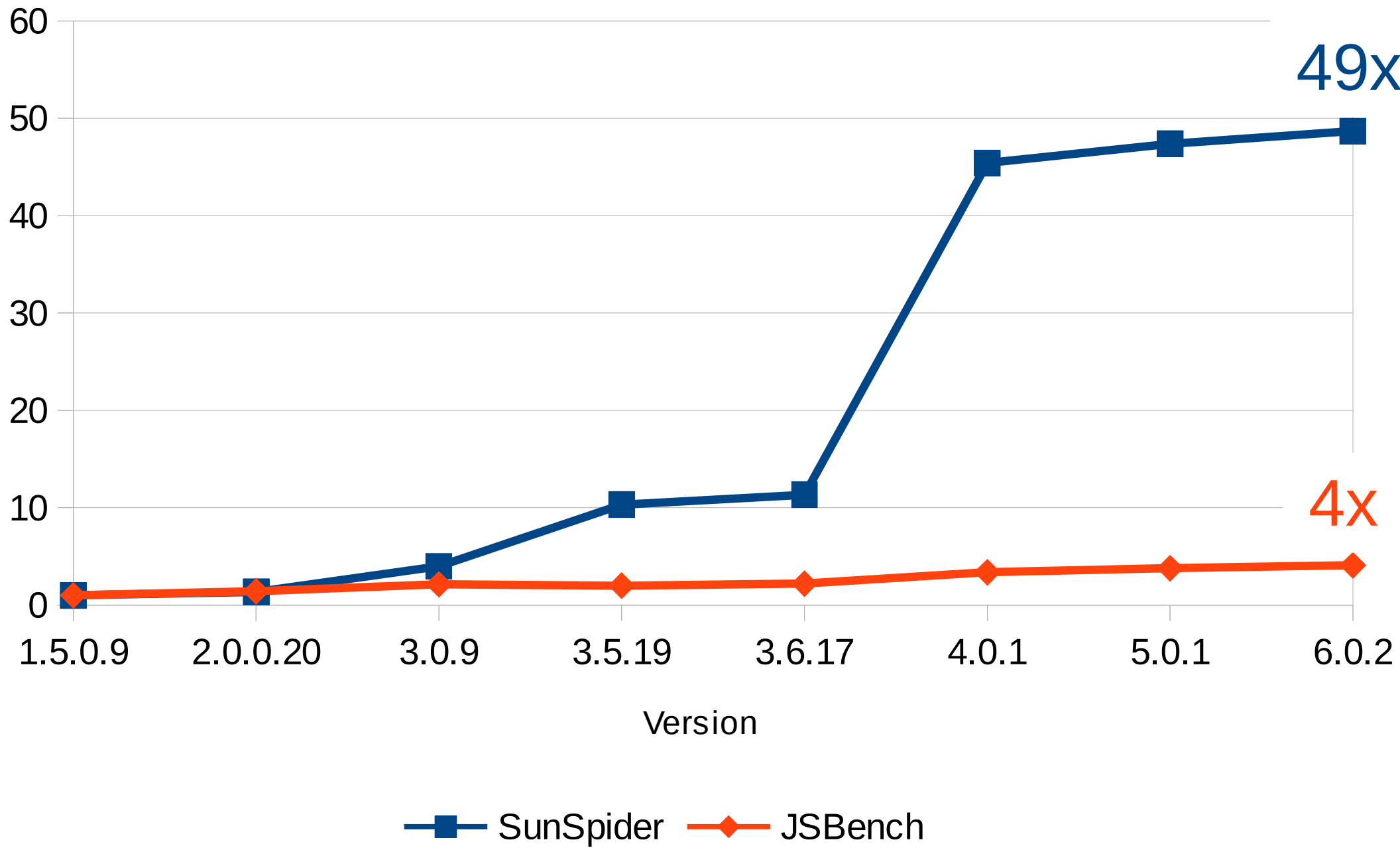
Record and Replay

Record:

- HTTP proxy
- JS instrumentation
- JS library log use of APIs

Replay:

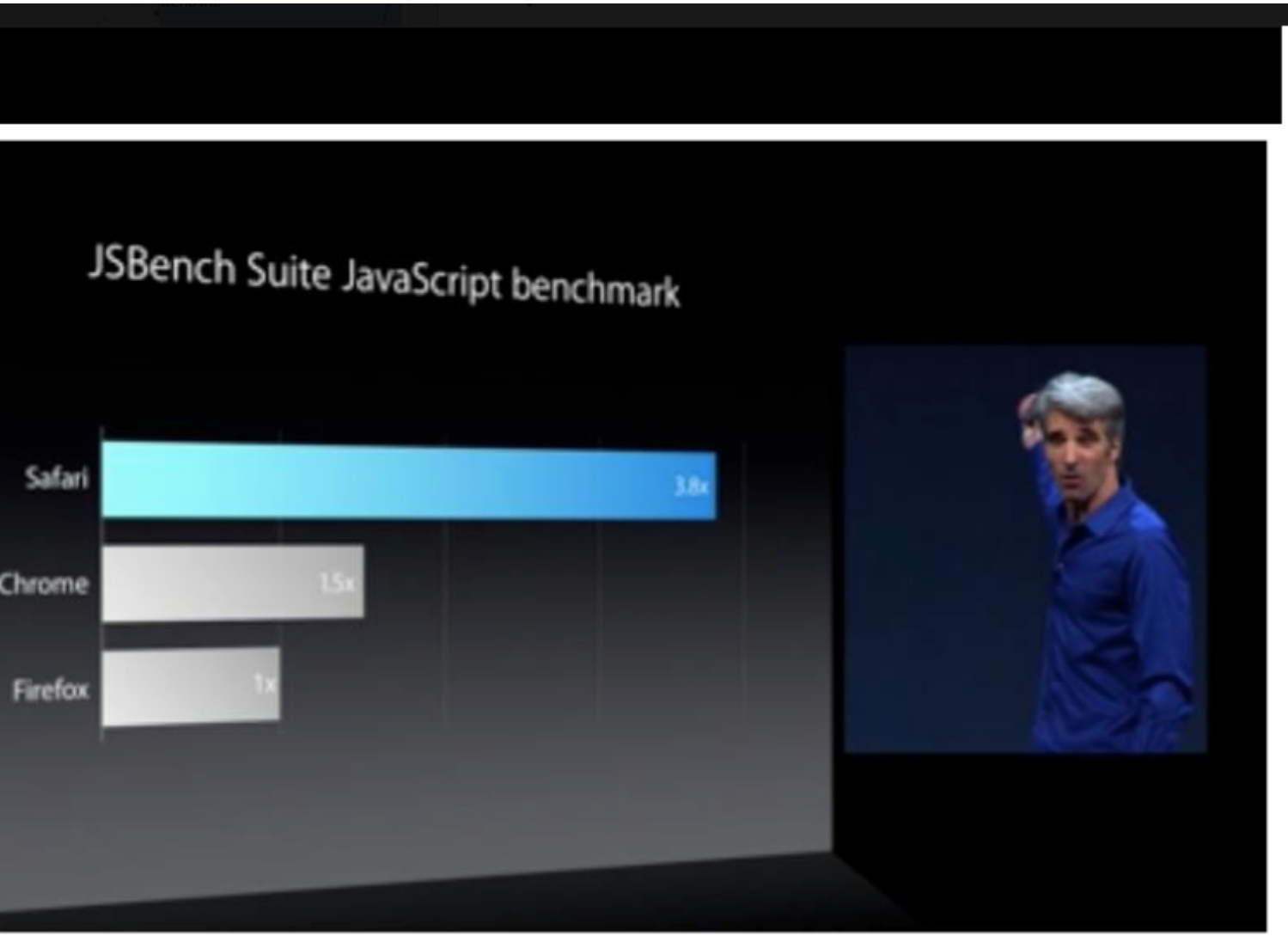
- APIs replayed from log
- “Push-button” benchmarks
- Works on any JS *engine*


60
50
40
30
20
10
0
49x
4x
1.5.0.9
2.0.0.20
3.0.9
3.5.19
3.6.17
4.0.1
5.0.1
6.0.2
Version
SunSpider
JSBench

Research v. The World

Research v. The World

Research v. The World



Jan Vitek
June 10, 2013

Holy mother of god... they are talking about JSBench at the Apple Keynote... that's got to b worth like a bunch of POPL papers.

Tag Photo · Add Location · Edit

Like · Comment · Stop Notifications · Share · Edit

Domagoj Babic, Koushik Sen, Suresh Jagannathan
35 others like this.

View 8 more comments

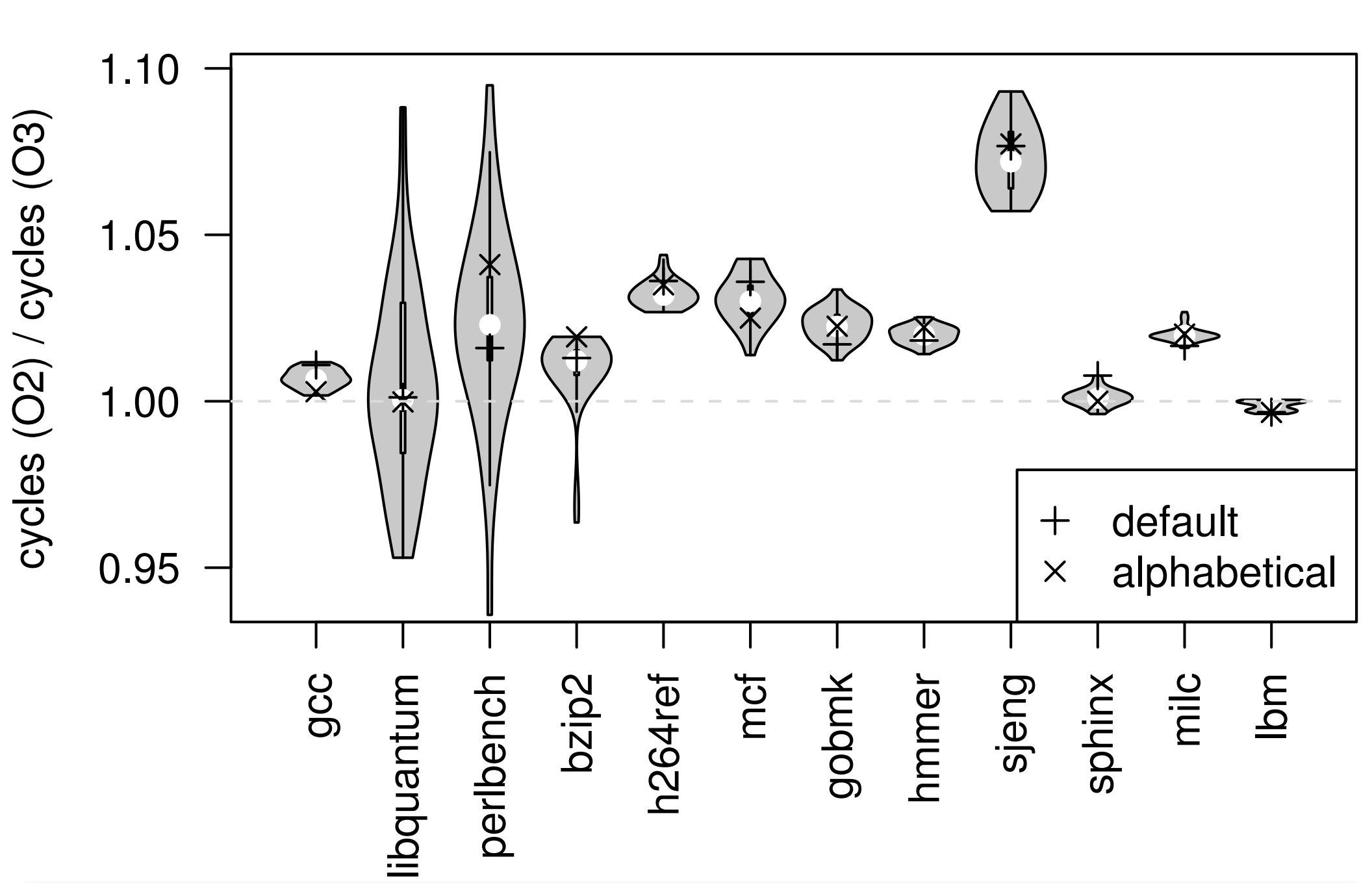
Derek Dreyer Huh, it took forever to load on Chrome, and on my Safari it says it can't run it at all because my Webkit is old (v 534.59.8). M Macbook Pro is only 2.5 years old.
June 11, 2013 at 2:35am · Like

Jan Vitek You, old. Now that you have tenure splurge and buy one of those nifty new MacBookAirs with Intel's new Hasbeen process They come with Safari installed 🙂
June 11, 2013 at 2:36am · Like

Derek Dreyer But why? This Macbook Pro is t first Apple computer I ever had that was not broken, and it still works great. I am too shellshocked from the narcissistic personality disorder exhibited by my first Macbook Air to contemplate buying one of those damn... See M

Performance Part II

how we measure things



Mytkowicz, Diwan, Hauswirth, Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! *ASPLOS'09*

Know the goals

- Ends-based
  - Performance characteristics of a single system
  - Comparison of two or more systems
- Explanatory
  - Find hints, evidence to explain observed behavior
- Scope
  - How general answer we’re looking for?

Usual goals in PL/Systems

- Ends-based
  - Improvement over the best performing system

    (in execution time, parallel speedup, power consumption, code size, pause time, reaction time)
  - Measured on application benchmarks, kernels
- Explanatory
  - Explain the improvements/overheads (cache misses, cache size, TLB, time spent in GC, memory utilization)
  - Sometimes using directed micro-benchmarks
- Scope
  - Pick one or two common platforms & OS
  - Common benchmark suites

Kinds of performance quantities

- Responsiveness
  - Time (response time, latency)
  - *Time between arrival of packet to the gateway and its successful delivery to destination*
- Productivity
  - Rate (throughput, speed, network bandwidth)
  - *Number of transactions processed per second by application server*
- Utilization
  - Percentage of time a particular resource is at least at given load level
  - *Percentage of time the CPU is not running the idle task*
- Stalls
  - Cache-misses, page-faults, pipeline

Prevailing metrics in PL/Systems are based on execution time

- Ratio of times – measure of optimizations
  - Improvement in execution time
  - Speed-up, parallel speed-up, performance overhead
- Absolute time
  - Time of a system to boot and start accepting input
  - Time of a garbage collection cycle
  - Pause time in concurrent garbage collector
  - Time to call a function

Factors impacting execution time

- Fixed effects
  - Algorithm/code/optimization – what we work on
  - Input (benchmark programs)
  - CPU, OS, libraries, compiler optimizations, location in virtual memory
  - Report (reduce scope) or **randomize**
- Random effects
  - Location in physical memory, system load, scheduling, context switches, hw interrupts, randomized algorithms
  - **Model, summarize using statistics**

Run DaCapo fop benchmark (Java) repeatedly, record execution times

```
for I in `seq 1 100` ; do
    java -jar dacapo-9.12-bach.jar fop
done > fop.out 2>&1
```

Read the times into R, into vector “x” (in seconds)

```
out <- readLines("fop.out")
rlines <-  grep("==== DaCapo .* in [0-9]+ msec.*", out, val=T)
timesms <-  as.numeric(gsub(".* in ([0-9]+) msec.*","\\1",rlines))
x <- timesms / 1000
```

Show first 10 times

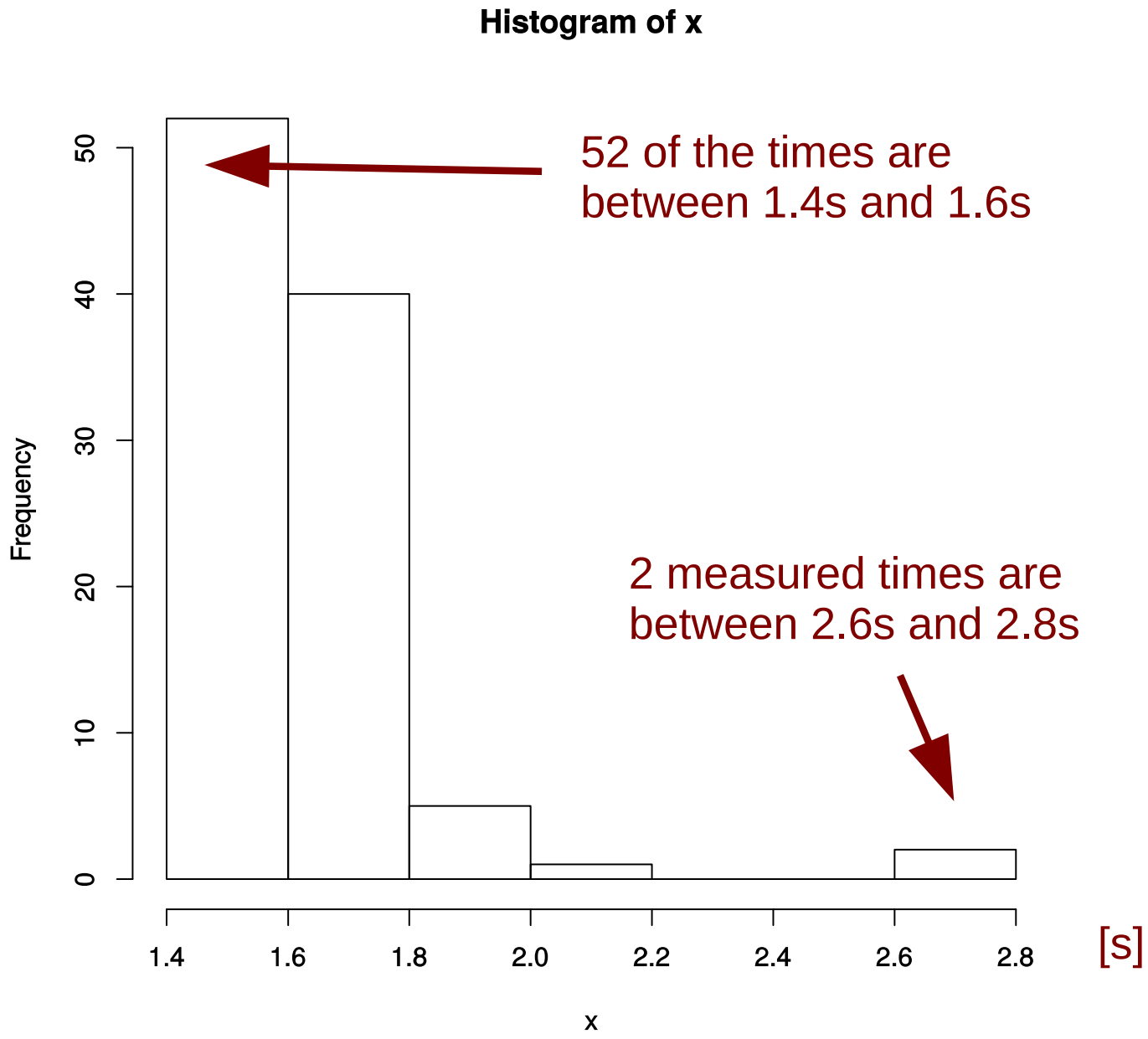
```
> x[1:10]
[1] 2.750 1.785 1.627 1.672 1.667 1.584 1.557 1.730 1.505 1.464
```

We **assume** times are repetitions of the same process, we have the same expectations about x[1] as about x[2]

We **assume** times are (statistically) independent – the fact that x[1] is 1.785 does not give us a clue what x[2] will be.

Show a histogram

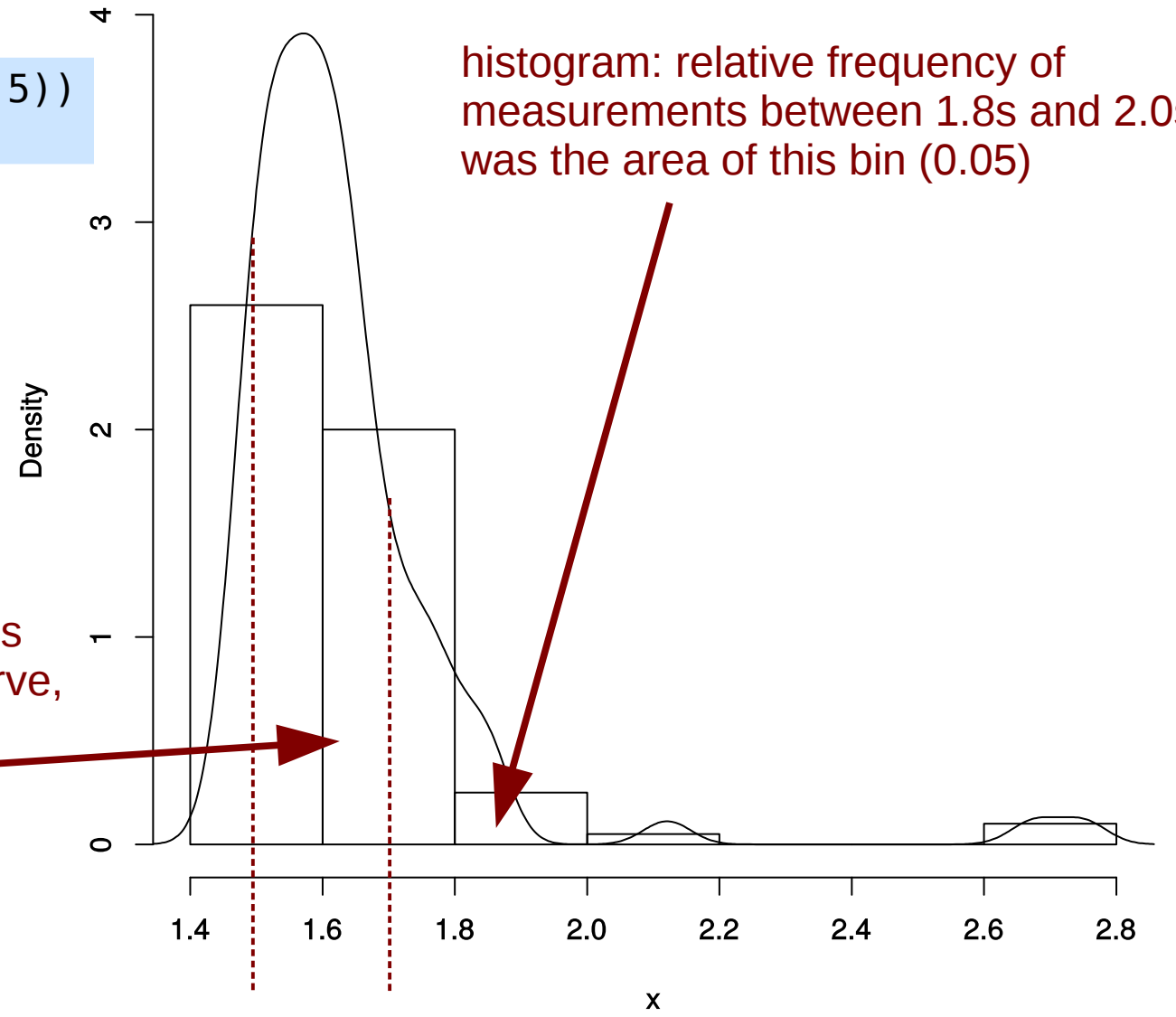
```
hist(x, freq=T)
```



We assume all times come from the same underlying process. With increasing number of iterations, the shape of the histogram should stabilize.

Histogram with **estimate** of probability density function

```
hist(x, prob=T, ylim=c(0,5))
lines(density(x))
```



histogram: relative frequency of measurements between 1.8s and 2.0s was the area of this bin (0.05)

density: probability of measured time to be between 1.5s and 1.7s is the area below the density curve, between 1.5 and 1.7.

Under our assumptions, the **true** density function would fully describe the execution times of the benchmark.

FFT

Now we have 100 runs of the benchmark
from each run we have 2048 measurements.

List of 100 vectors (100 runs of the benchmark)

File names are like fft_ia64/run1.out

```
runs <- lapply(1:100, function(n) {
  d <- read.table(
    paste("fft_ia64/run", n, ".out", sep=""),
    header=T
  )
  d[[1]]
})
ia64 <- do.call(c, runs)
ia64 <- ia64/(800.179008*1e6)
```

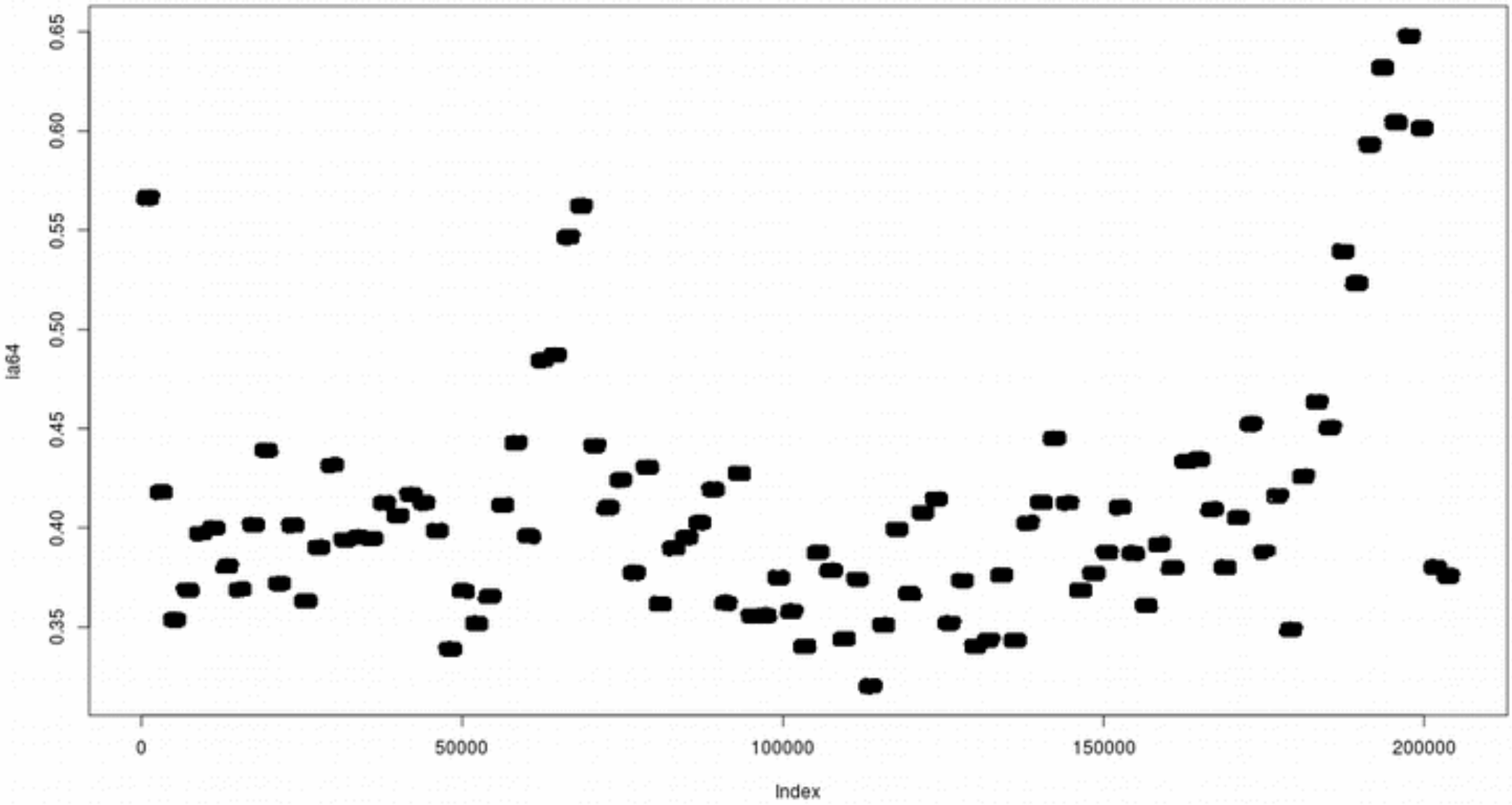
Vector of 2048 measurements
(first column of result data frame)

Vector of 100 * 2048 measurements

FFT

Lets explore the sequence of measurements from different runs.

```
plot(ia64)
```

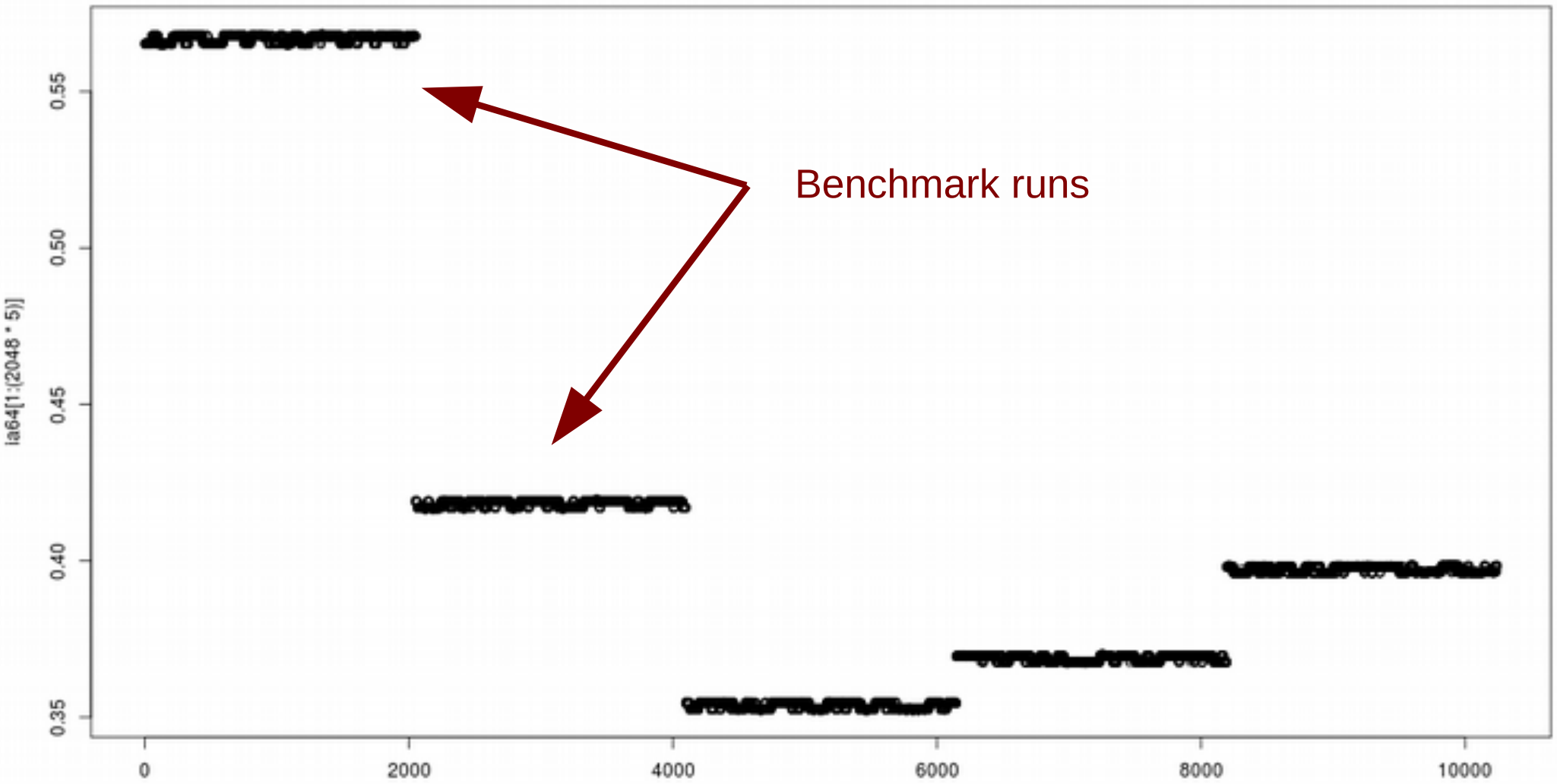


FFT

Lets explore the sequence of measurements from different runs.

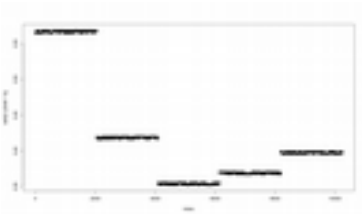
```
plot(ia64[1:(2048*5)])
```

(a simple variant of DEX scatter plot)



Non-determinism in execution (that does not appear in iterations)

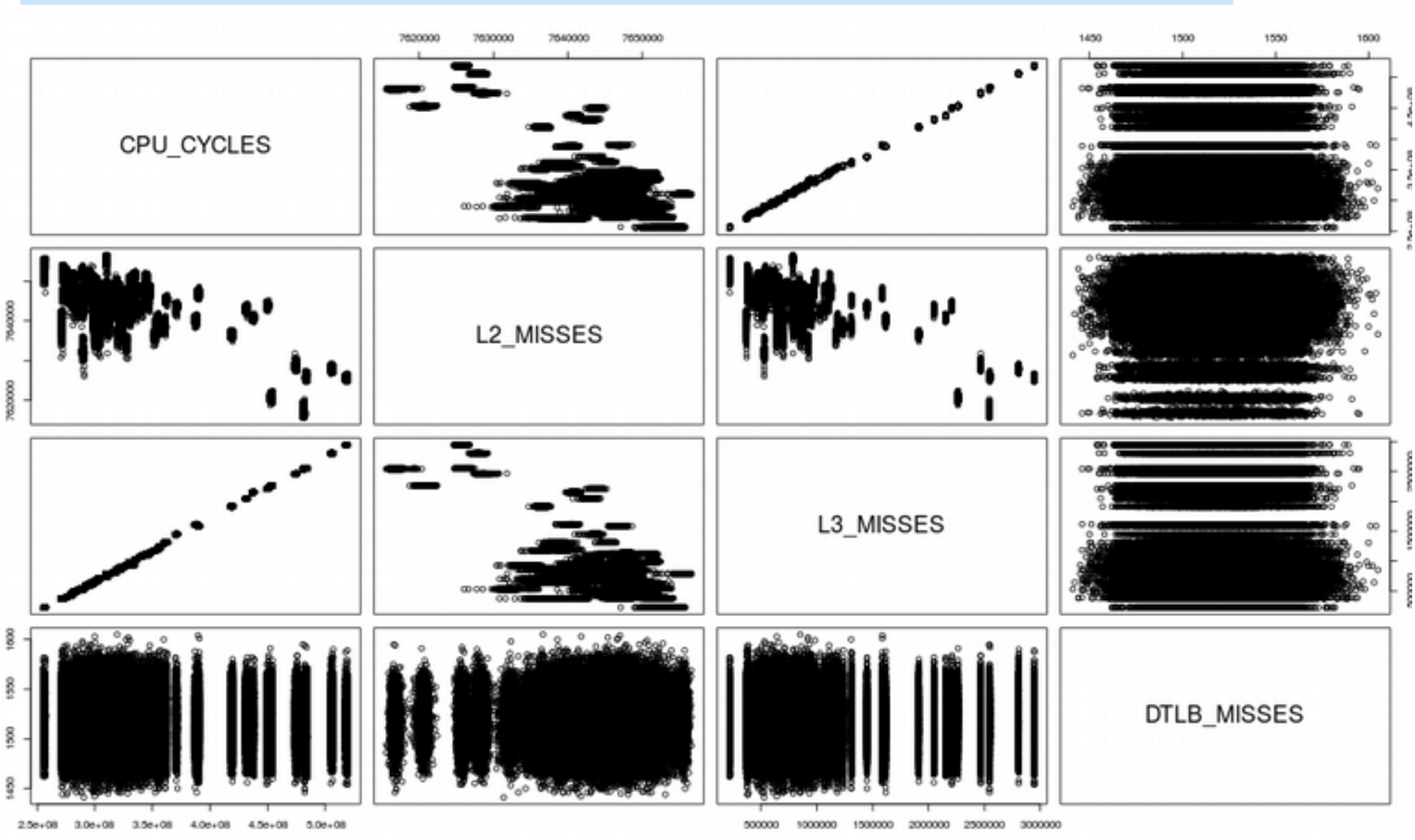
- Different executions of a benchmark have different performance
  - Plus with FFT, the difference is much bigger than between iterations in the same run
- Uncontrolled fixed effect
- Must re-run executions to avoid **bias**
  - And given the big impact of “execution”, no need to repeat iterations within execution



What is the cause of this non-determinism?

```
runs <- lapply(1:100, function(n)
  read.table(paste("fft_ia64/run", n, ".out", sep=""), header=T)
)
ia64 <- do.call(rbind, runs)
plot(ia64)
```

Joining data frames from multiple runs



Non-determinism in compilation (that does not appear in executions)

- On some systems, linking order impacts performance (e.g. SPECCPU, training size)
  - Controlled fixed effects
  - Should be randomized – and then need to repeat compilation
- On some systems, build is non-deterministic
  - e.g. C++ compiler implementing anonymous namespaces
- Need to repeat compilation...

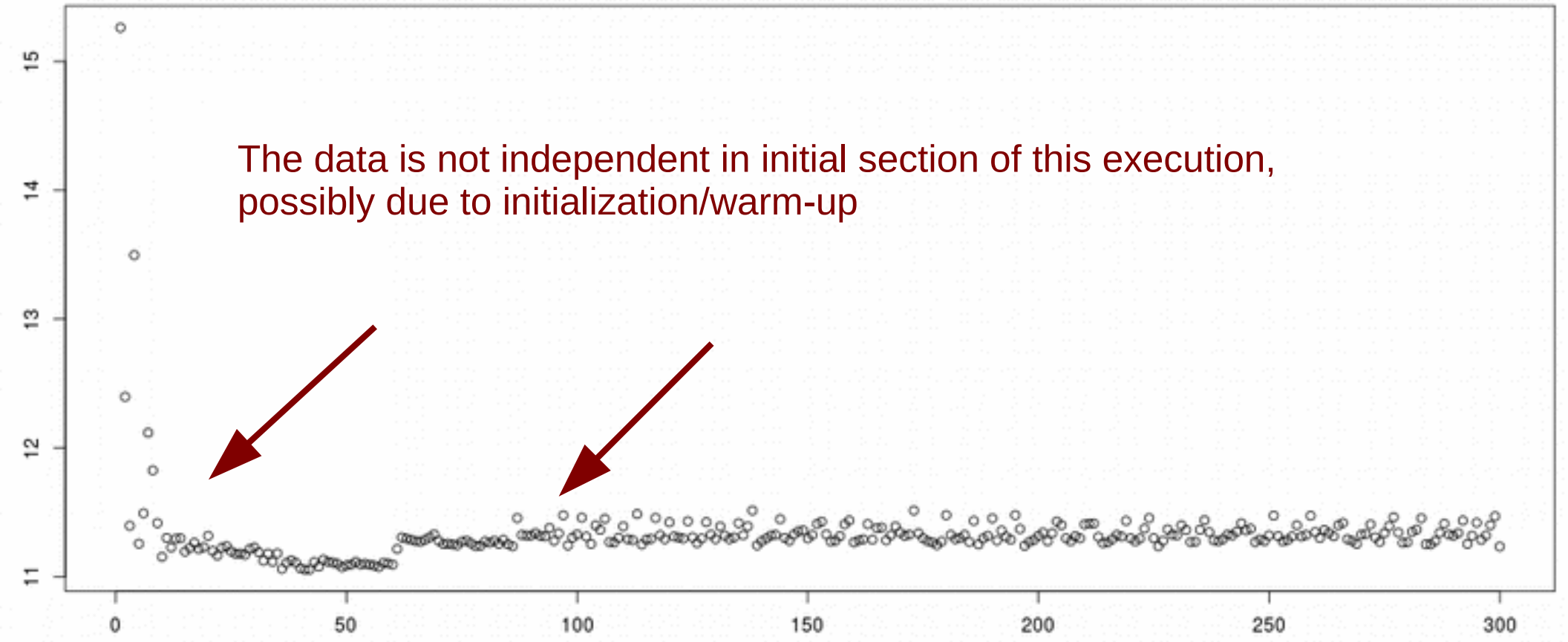
NOTE: naming of identifiers has also been reported to impact performance; code layout by function/data order does too..

Highest level for repetition

- Find the highest “level” of non-determinism in the given system/benchmark
  - Identify important uncontrolled fixed effects and randomize them (like linking order, code layout)
  - Check if building is deterministic, binaries have different performance
- If it is cheap enough, repeat at a higher level anyway
  - If execution is cheap, repeat whole executions always
- If in doubt, repeating at a higher level is never wrong

Read chart6 times into R, into vector “x” (in seconds)

```
readDacapo <- function(fn) {
  out <- readLines(fn)
  rlines <-  grep("==== DaCapo .* in [0-9]+ msec.*", out, val=T)
  timesms <-  as.numeric(gsub(".* in ([0-9]+) msec.*","\\1",rlines))
  timesms / 1000
}
x <- readDacapo("dacapo/chart6/chart6_1_1.out")
plot(x)
```



Dealing with warm-up

- Identify #iterations affected by initialization
  - Using run-sequence plots of different scales
  - Validating on several runs
  - Only obvious initialization, after which results are stable/similar
- Identify #iterations to independent state
  - Using acf, lag.plot, run-sequence plots
  - **Only independent data can be used for summarization using confidence interval**
- Neither stability nor independence is always reached
- If not, can only use 1 number from run for summarization

Comparing two systems

T_{new}	Time on the new system (usually ours). Lower is better.	**58s**
T_{old}	Time on the old/baseline system.	**16s**

$\frac{T_{new}}{T_{old}}$	**0.28 (28%)**	**Ratio of execution times**
$1-\frac{T_{new}}{T_{old}}$	**0.72 (72%)**	**Percentage improvement in execution time**
$\frac{T_{old}}{T_{new}}$	**3.63 (363%, 3.63x)**	**Speedup**
$\frac{T_{old}}{T_{new}}-1$	**2.63 (263%)**	**“Percentage improvement in speed”**