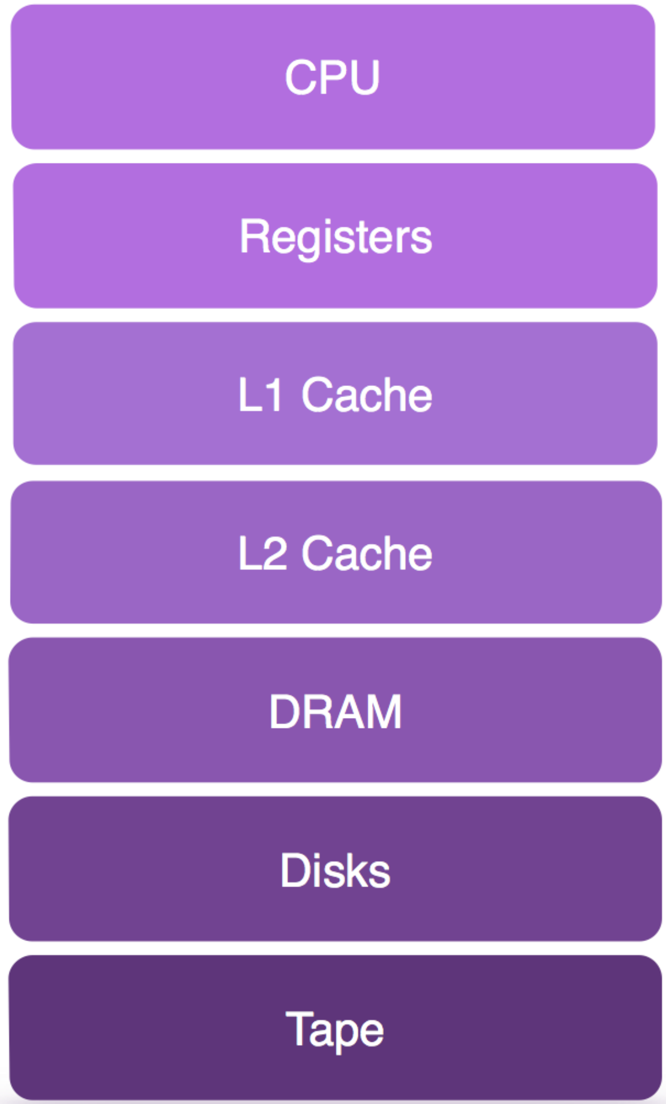
Characterizing Storage Workloads with Counter Stacks

Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, Andrew Warfield

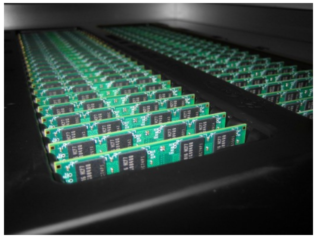
Coho Data, UBC

Memory Hierarchies

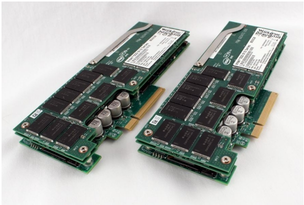


Memory Hierarchies

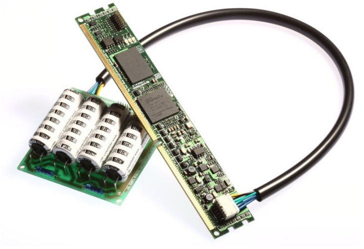
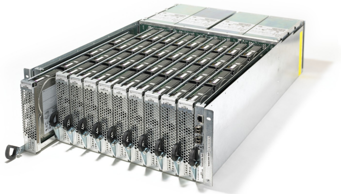
Challenge: Provisioning

512 GB DRAM + 8 TB SATA SSDs = $4,200 8.5 TB 10K – Millions IOPS

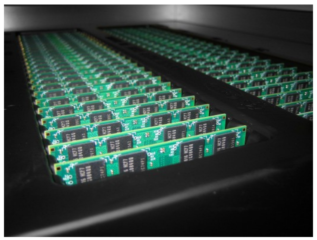
 

1.6 TB PCIe Flash + 12 TB HDDs = $12,000 13.6 TB 2.4K – 2M IOPS

8 GB NVDIMM + 60 TB JBOD = $8,000 60 TB 12K – Millions IOPS

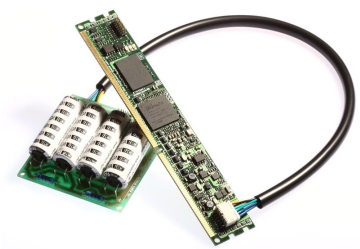
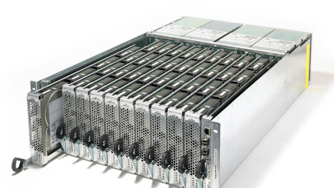
Challenge: Provisioning




512 GB DRAM + [illegible] SATA S[illegible]Ds = $4,20[illegible] [illegible]B 10K – Millions IOPS


1.6 TB PCIe Flash + 12 TB HDDs = $12,000 [illegible]6 TB 2.4K – 2M IOPS

???

8 GB NVDIMM + 60 TB JBOD = $8,000 60 TB 12K – Millions IOPS

Challenge: Placement

Workload Characterization

- Provisioning and placement are difficult problems
- **What are the key workload characteristics we can use to solve these problems?**

Optimal

MIN (Belady, '66): prioritize pages with shortest *forward distance*

Practical

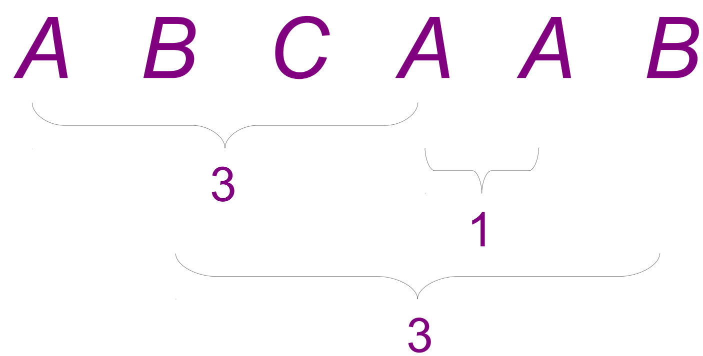
LRU: prioritize pages with shortest *reuse distance*

Practical

LRU: prioritize pages with shortest *reuse distance*

Reuse Distances

- # of distinct symbols since previous reference

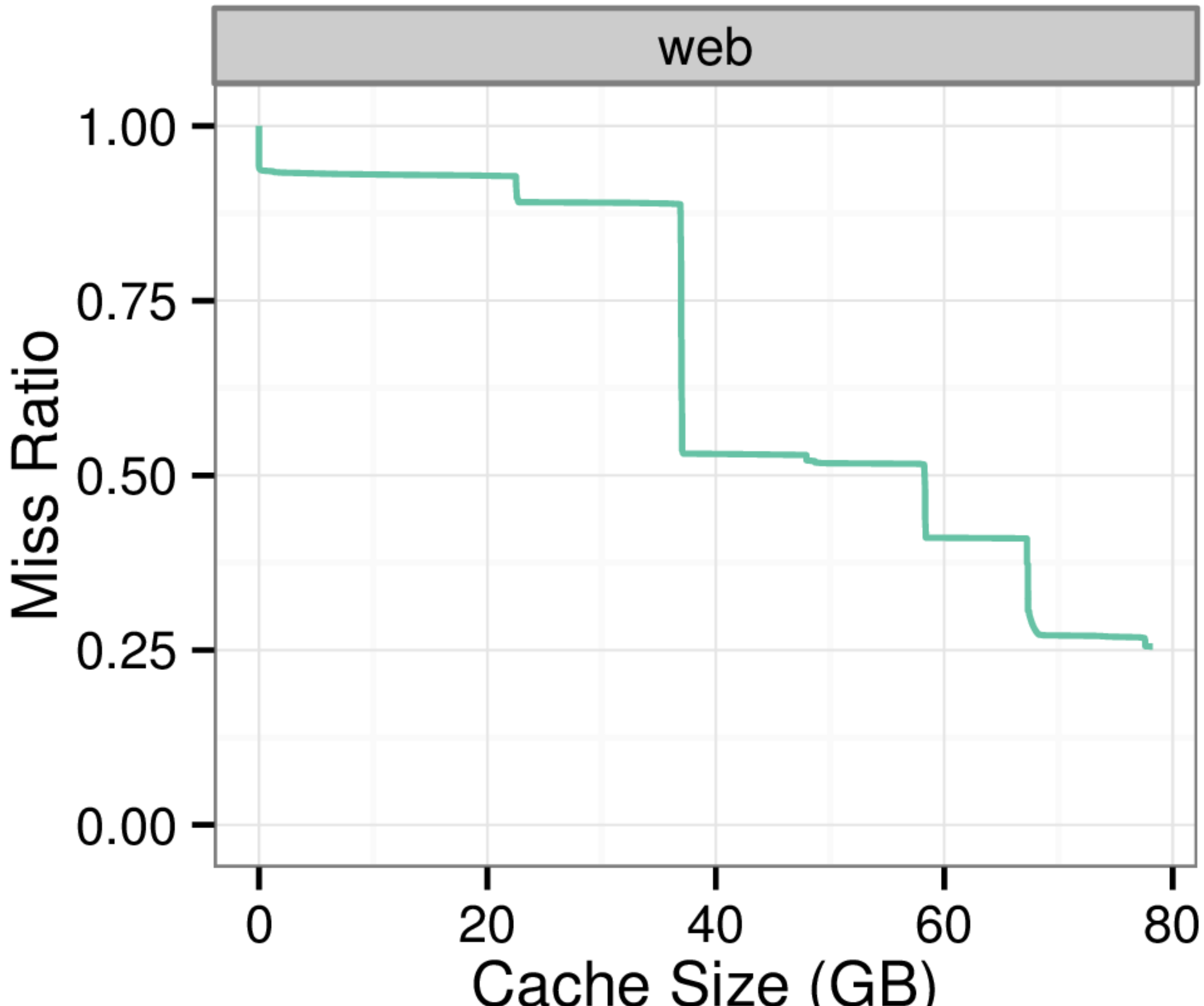


- Measure of workload locality
- Model of memory behavior

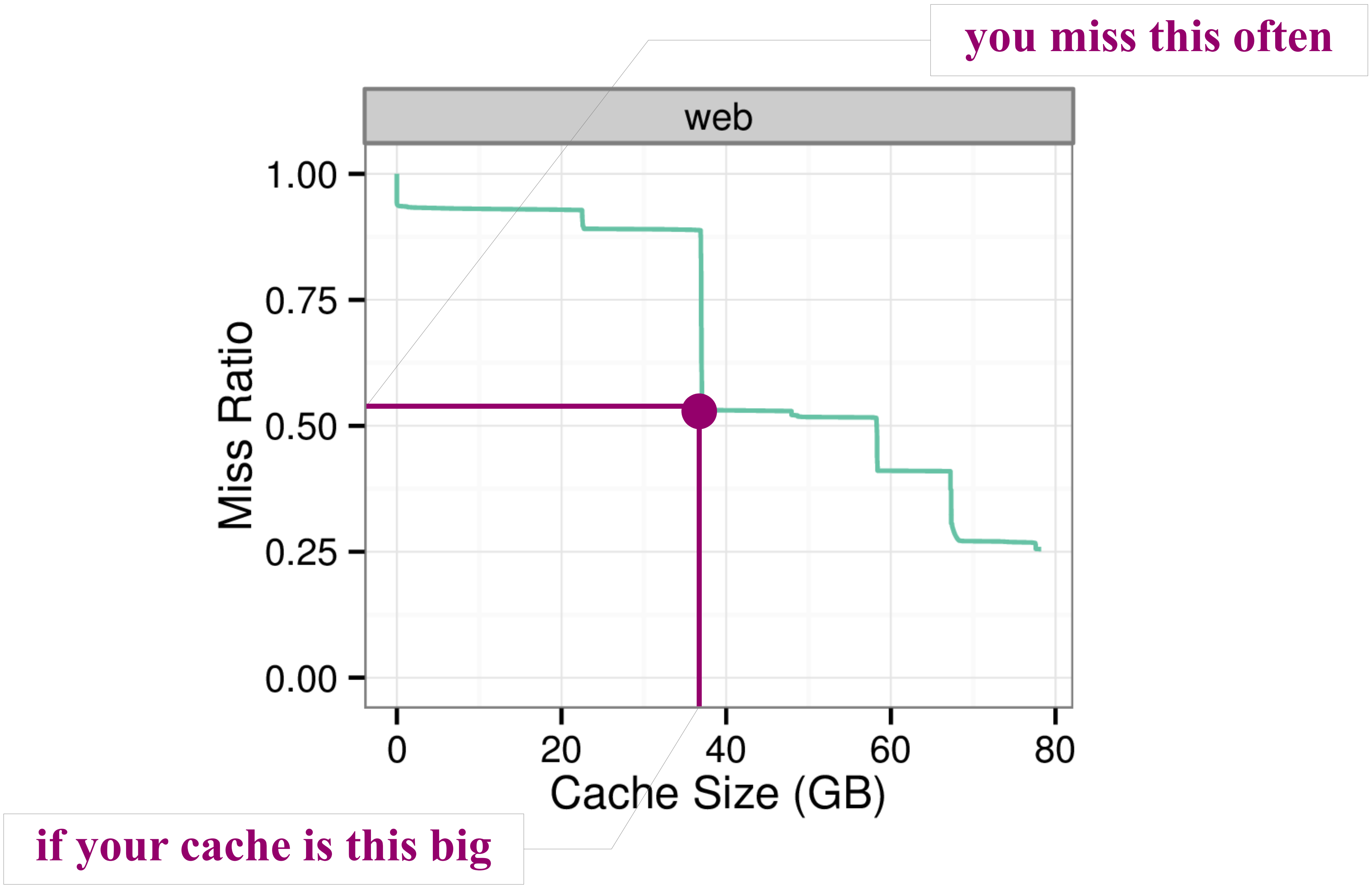
Miss Ratio Curves

- A plot of miss rate vs. cache size for a given workload under a given replacement policy
  - With LRU, this is the distribution of reuse distances

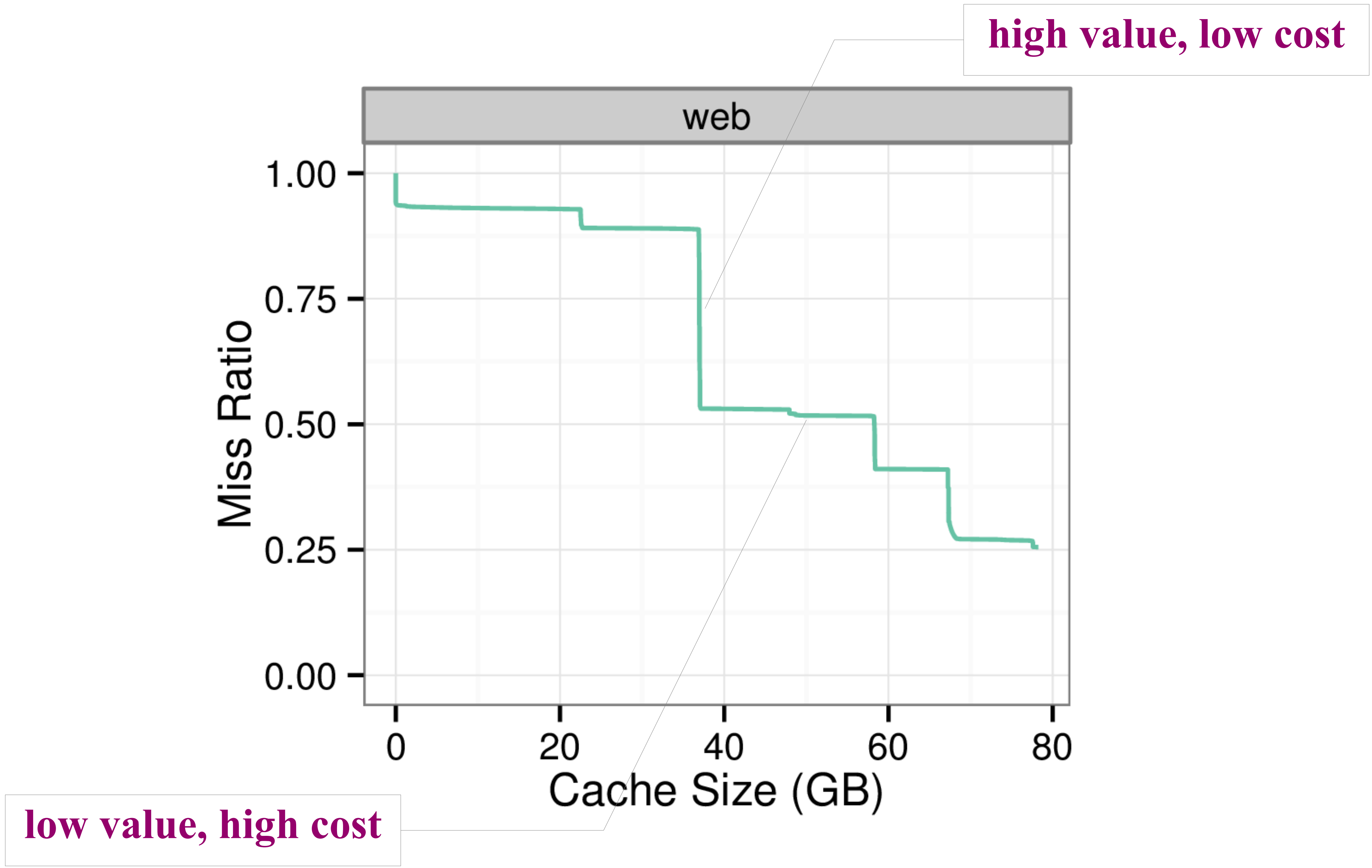
Miss Ratio Curves



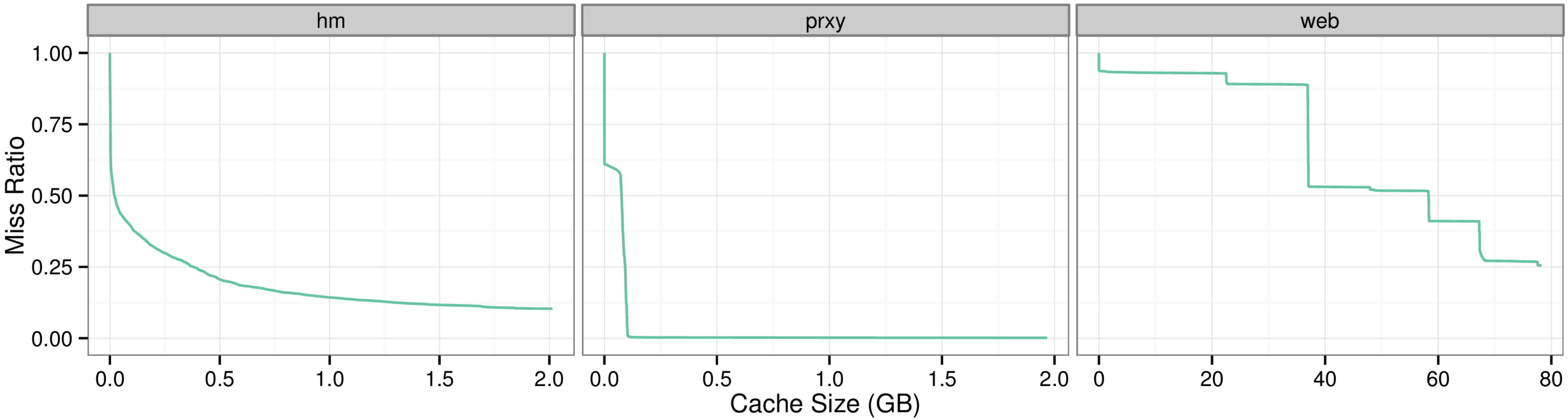
Miss Ratio Curves



Miss Ratio Curves



Miss Ratio Curves

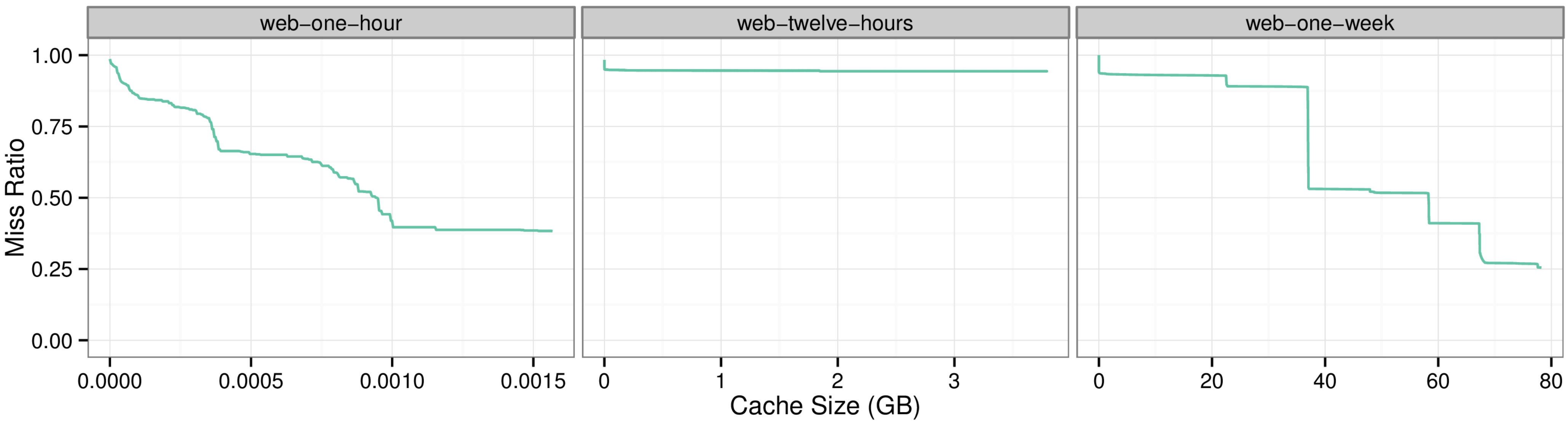


Hardware Monitor

Web Proxy

Web/SQL Server

Miss Ratio Curves



One Hour Twelve Hours One Week

Computing MRCs

- Naïve approach
  - Simulate workload once at each cache size

Computing MRCs

- Naïve approach
  - Simulate workload once at each cache size

Computing MRCs

- Mattson's Stack Algorithm ('70)
  - Some replacement policies are *inclusive*
    - Larger caches always include contents of smaller caches

Computing MRCs

- Mattson's Stack Algorithm ('70)
  - Some replacement policies are *inclusive*
    - Larger caches always include contents of smaller caches
  - LRU, LFU, MIN, ...
  - For such policies, simulate all cache sizes in one pass
    - Hits at size N are hits at all M > N

Stack Algorithm for LRU

- To compute miss ratio curves for LRU:
  - Compute reuse distance of each request
  - Aggregate distances in a histogram
  - Compute the cumulative sum (CDF)

Stack Algorithm for LRU

- Complexity (N records, M unique symbols):
  - Time: O(N * M)
    - Reduced to O(N * log(N)) (Bennett et al., '75)
    - Reduced to O(N * log(M)) (Almási et al., '02)
  - Space: O(M)

Stack Algorithm for LRU

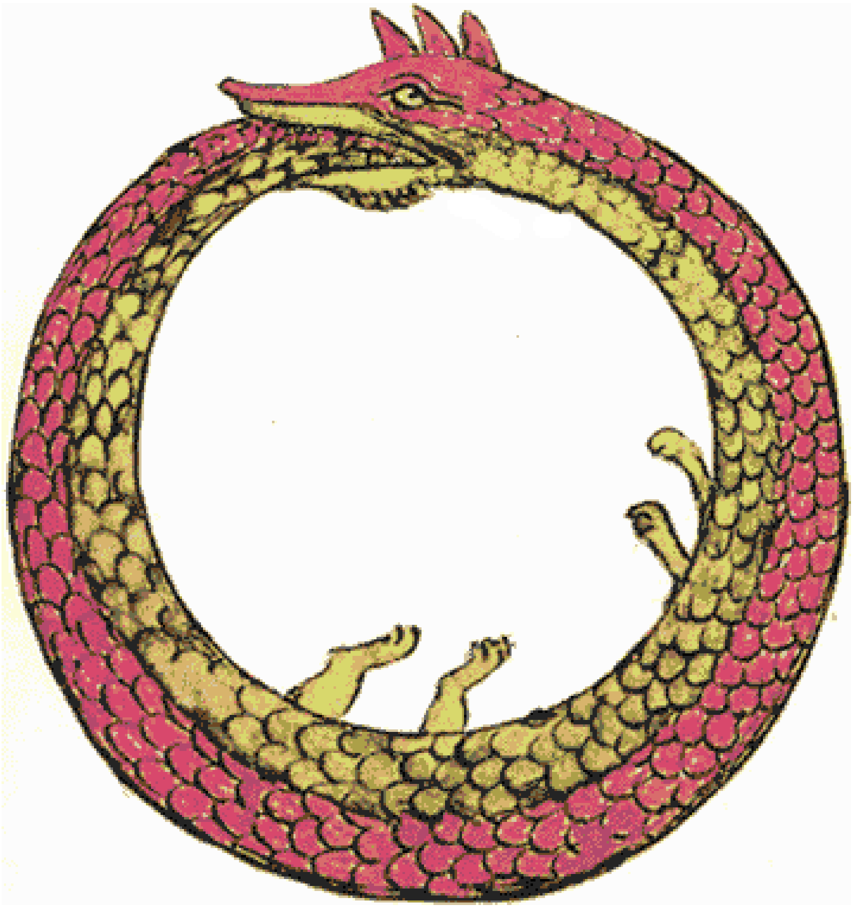
- Complexity (N records, M unique symbols):
  - Time: O(N * M)
    - Reduced to O(N * log(N)) (Bennett et al., '75)
    - Reduced to O(N * log(M)) (Almási et al., '02)
  - Space: O(M)
    - ...

Still Not Practical

- 92 GB RAM to compute MRC of 3 TB workload

Still Not Practical

- 92 GB RAM to compute MRC of 3 TB workload



Stack Algorithm for LRU

- To compute miss ratio curves for LRU:
  - **Compute reuse distance of each request**
  - Aggregate distances in a histogram
  - Compute the cumulative sum (CDF)

- **Can we do this more efficiently?**

Stack Algorithm for LRU

- To compute miss ratio curves for LRU:
  - **Compute reuse distance of each request**
  - Aggregate distances in a histogram
  - Compute the cumulative sum (CDF)

- **Can we do this more efficiently? Yes.**
  - **80 MB for approximate MRC of 3 TB workload**

Counter Stacks

- *Measure uniqueness over time*

- Observation: computing reuse distances is related to counting distinct elements

- Consider a 'stack' of cardinality counters, one for each request

Calculating with Counts

Reference String: A

Calculating with Counts

Reference String:	A
cardinality counter started at t_0	1

Calculating with Counts

Reference String: *A* B

cardinality counter started at t_0	1

Calculating with Counts

Reference String:	*A*	B
cardinality counter started at t_0	1	2

Calculating with Counts

Reference String:	*A*	B
cardinality counter started at t_0	1	2
cardinality counter started at t_1		1

Calculating with Counts

Reference String:	*A*	*B*	C
cardinality counter started at t_0	1	2	
cardinality counter started at t_1		1	

Calculating with Counts

Reference String:	*A*	*B*	C
cardinality counter started at t_0	1	2	3
cardinality counter started at t_1		1	

Calculating with Counts

Reference String:	*A*	*B*	C
cardinality counter started at t_0	1	2	3
cardinality counter started at t_1		1	2

Calculating with Counts

Reference String:	*A*	*B*	C
cardinality counter started at t_0	1	2	3
cardinality counter started at t_1		1	2
cardinality counter started at t_2			1

Calculating with Counts

Reference String:	*A*	*B*	*C*	A
cardinality counter started at t_0	1	2	3	
cardinality counter started at t_1		1	2	
cardinality counter started at t_2			1	

Calculating with Counts

Reference String:	*A*	*B*	*C*	A
cardinality counter started at t_0	1	2	3	3
cardinality counter started at t_1		1	2	
cardinality counter started at t_2			1	

Calculating with Counts

Reference String:	*A*	*B*	*C*	A
cardinality counter started at t_0	1	2	3	3
cardinality counter started at t_1		1	2	3
cardinality counter started at t_2			1	

Calculating with Counts

Reference String:	*A*	*B*	*C*	A
cardinality counter started at t_0	1	2	3	3
cardinality counter started at t_1		1	2	3
cardinality counter started at t_2			1	2

Calculating with Counts

Reference String:	*A*	*B*	*C*	A
cardinality counter started at t_0	1	2	3	3
cardinality counter started at t_1		1	2	3
cardinality counter started at t_2			1	2
cardinality counter started at t_3				1

Calculating with Counts

Reference String:	*A*	*B*	*C*	*A*	
cardinality counter started at t_0	1	2	3	3	+0
cardinality counter started at t_1		1	2	3	+1
cardinality counter started at t_2			1	2	
cardinality counter started at t_3				1	

Observation 1: A difference in the change between adjacent counters implies a repeated reference.

Calculating with Counts

Reference String:	*A*	*B*	*C*	*A*	
cardinality counter started at t_0	1	2	3	3	+0
cardinality counter started at t_1		1	2	3	+1
cardinality counter started at t_2			1	2	
cardinality counter started at t_3				1	

Observation 1: A difference in the change between adjacent counters implies a repeated reference.

Observation 2: The location of the difference stores the reuse distance.

Calculating with Counts

A	B	C	A
1	2	3	3
	1	2	3
		1	2
			1

Matrix C

A	B	C	A
1	1	1	0
	1	1	1
		1	1
			1

Δx

A	B	C	A
0	0	0	1
	0	0	0
		0	0
			0

Δy

Perfect Counting

- One cardinality counter per request
- Quadratic overhead!

Perfect Counting

- ~5 ZB RAM to compute MRC of 3 TB workload

Practical Counting

- $\mathbb{C}$ is highly redundant
  - Space/accuracy tradeoff

A	*B*	*C*	*A*	*A*	*C*	*A*	*B*	*C*	*A*
1	2	3	3	3	3	3	3	3	3
	1	2	3	3	3	3	3	3	3
		1	2	2	2	2	3	3	3
			1	1	2	2	3	3	3
				1	2	2	3	3	3
					1	2	3	3	3
						1	2	3	3
							1	2	3
								1	2
									1

Practical Counting

- *Downsample*

A	B	C	A	A	C	A	B	C	A
1	2	3	3	3	3	3	3	3	3
	1	2	3	3	3	3	3	3	3
		1	2	2	2	2	3	3	3
			1	1	2	2	3	3	3
				1	2	2	3	3	3
					1	2	3	3	3
						1	2	3	3
							1	2	3
								1	2
									1

Practical Counting

- *Downsample*
  - Only output every k^{th} counter

A	*B*	*C*	*A*	*A*	*C*	*A*	*B*	*C*	*A*
1	2	3	3	3	3	3	3	3	3
	1	2	3	3	3	3	3	3	3
		1	2	2	2	2	3	3	3
			1	1	2	2	3	3	3
				1	2	2	3	3	3
					1	2	3	3	3
						1	2	3	3
							1	2	3
								1	2
									1

Practical Counting

- *Downsample*
  - Only output every k^{th} counter
  - Only output every k^{th} count

A	*B*	*C*	*A*	*A*	*C*	*A*	*B*	*C*	*A*
1	2	3	3	3	3	3	3	3	3
	1	2	3	3	3	3	3	3	3
		1	2	2	2	2	3	3	3
			1	1	2	2	3	3	3
				1	2	2	3	3	3
					1	2	3	3	3
						1	2	3	3
							1	2	3
								1	2
									1

Practical Counting

- *Prune*: discard counters with similar values (i.e., differing less than pruning distance *p)*

A	B	C	A	A	C	A	B	C	A
1	2	3	3	3	3	3	3	3	3
	1	2	3	3	3	3	3	3	3
		1	2	2	2	2	3	3	3
			1	1	2	2	3	3	3
				1	2	2	3	3	3
					1	2	3	3	3
						1	2	3	3
							1	2	3
								1	2
									1

Practical Counting

- *Prune*: discard counters with similar values (i.e., differing less than pruning distance *p*)

A	B	C	A	A	C	A	B	C	A
1	2	3	3	3	3	3	3	3	3
	1	2	3	3	3	3	3	3	3
		1	2	2	2	2	3	3	3
			1	1	2	2	3	3	3
				1	2	2	3	3	3
					1	2	3	3	3
						1	2	3	3
							1	2	3
								1	2
									1

Approximate Counting

- *Estimate*: use probabilistic counters

Approximate Counting

- *Estimate*: use probabilistic counters
  - HyperLogLog (Flajolet et al., '07)
  - Accurate estimates of large multisets with sublinear space

Counter Stacks

- Sublinear memory overhead
  - Practical for online computation

Counter Stacks

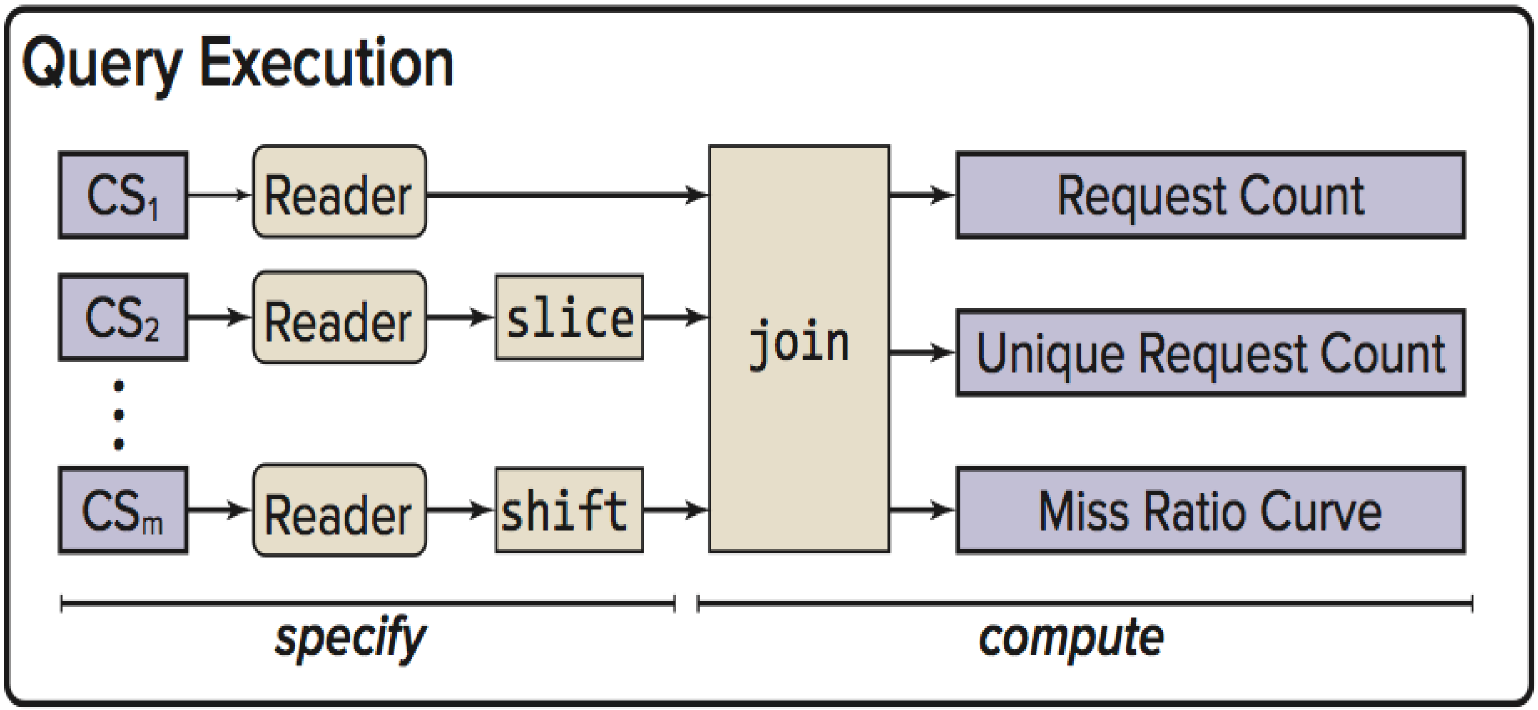
- Sublinear memory overhead
  - Practical for online computation

- But wait, there's more...

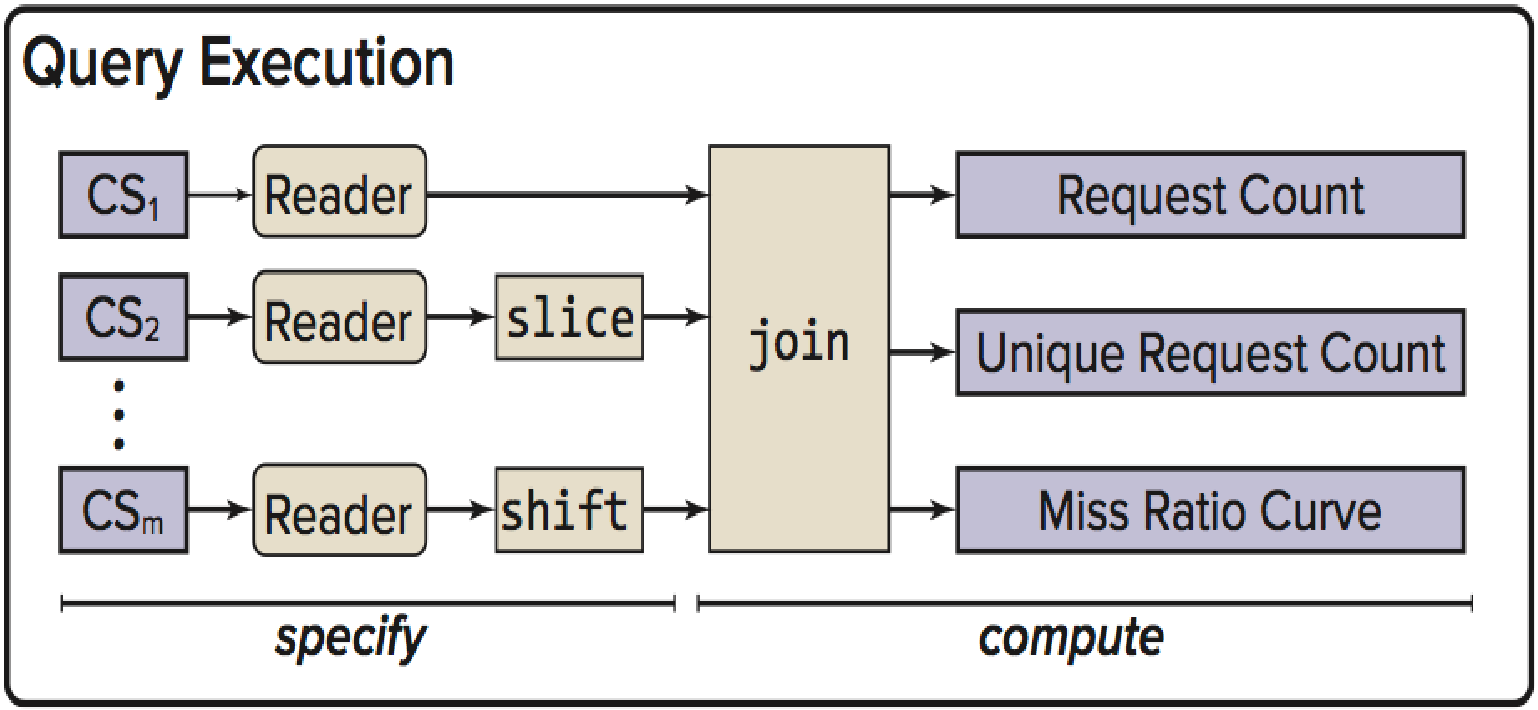
Counter Stack Streams

- We can compute Δx, Δy, and reuse distances with only the last two columns of C
- We store all columns on disk as a **Counter Stack Stream**
  - Preserves a *history of locality*

Counter Stack Stream Queries



Counter Stack Stream Queries



- Search for outliers
- Identify phase changes
- Explore coarse-grain scheduling

How Much Do They Cost?

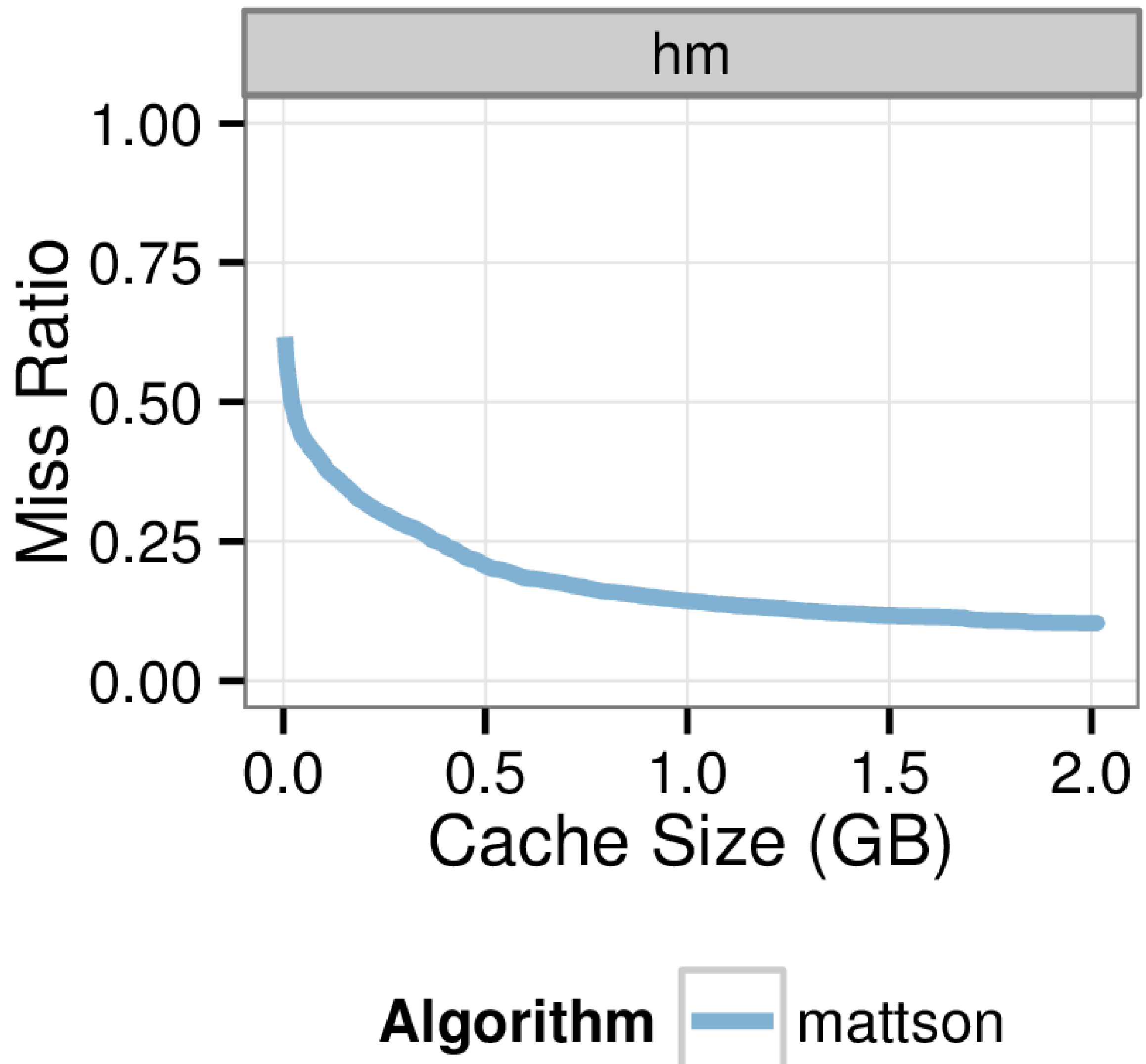
- MSR Cambridge storage traces
  - 2.7 TB unique data
  - 13 servers, 36 volumes, one week
  - 417 million records in 5 GB of gzipped CSV

Technique	RAM	Throughput	Storage
Mattson	92 GB	680 K reqs/sec	2.9 GB
high-fidelity CS	80.6 MB (1168x)	2.29 M reqs/sec (3.37x)	11 MB (270x)
low-fidelity CS	78.5 MB (1200x)	2.31 M reqs/sec (3.40x)	747 KB (4070x)

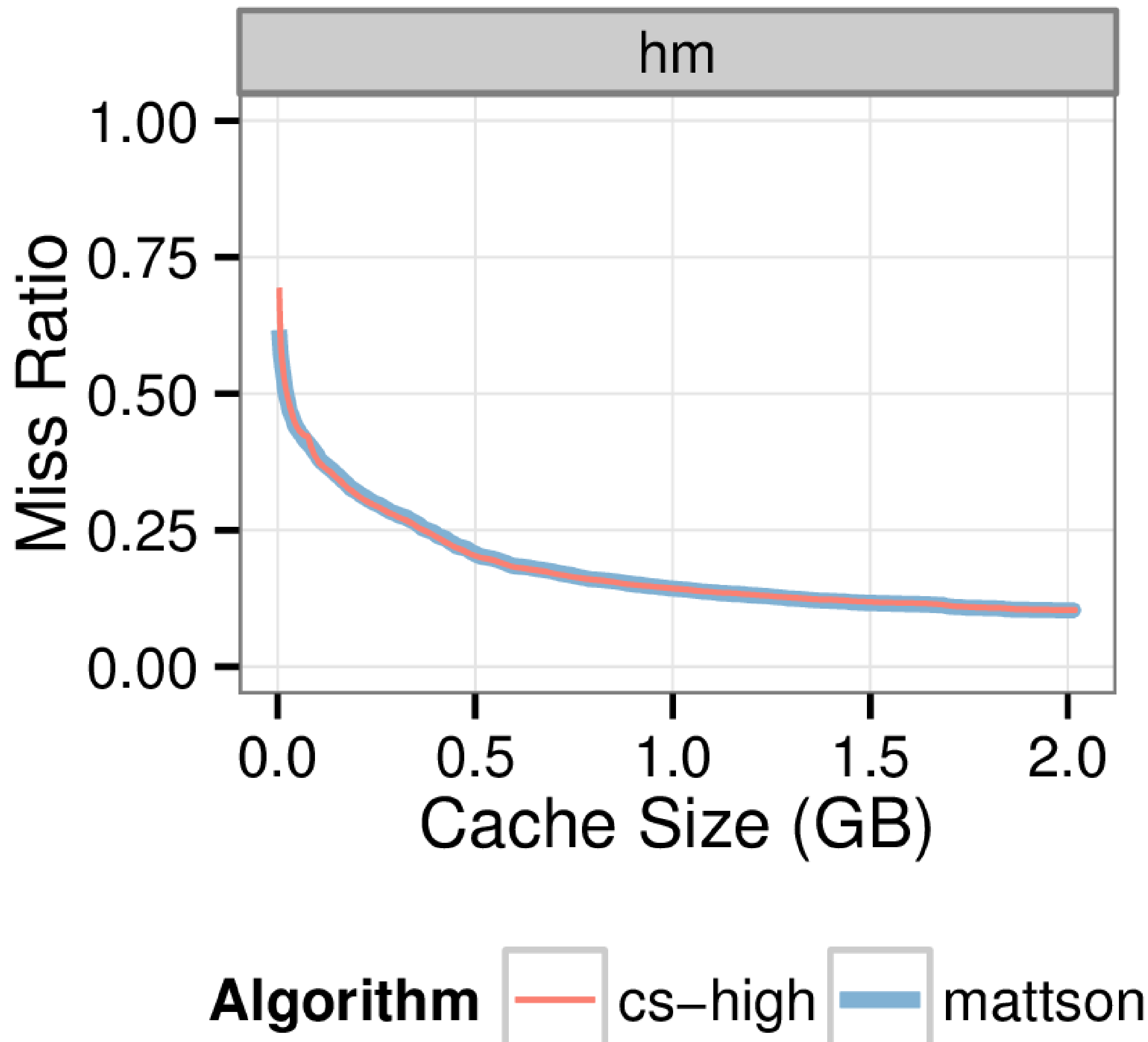
compression parameters are tunable:
high: $k = 10^6$, $p = 98\%$ low: $k = 10^6$, $p = 90\%$

How Well Do They Work?

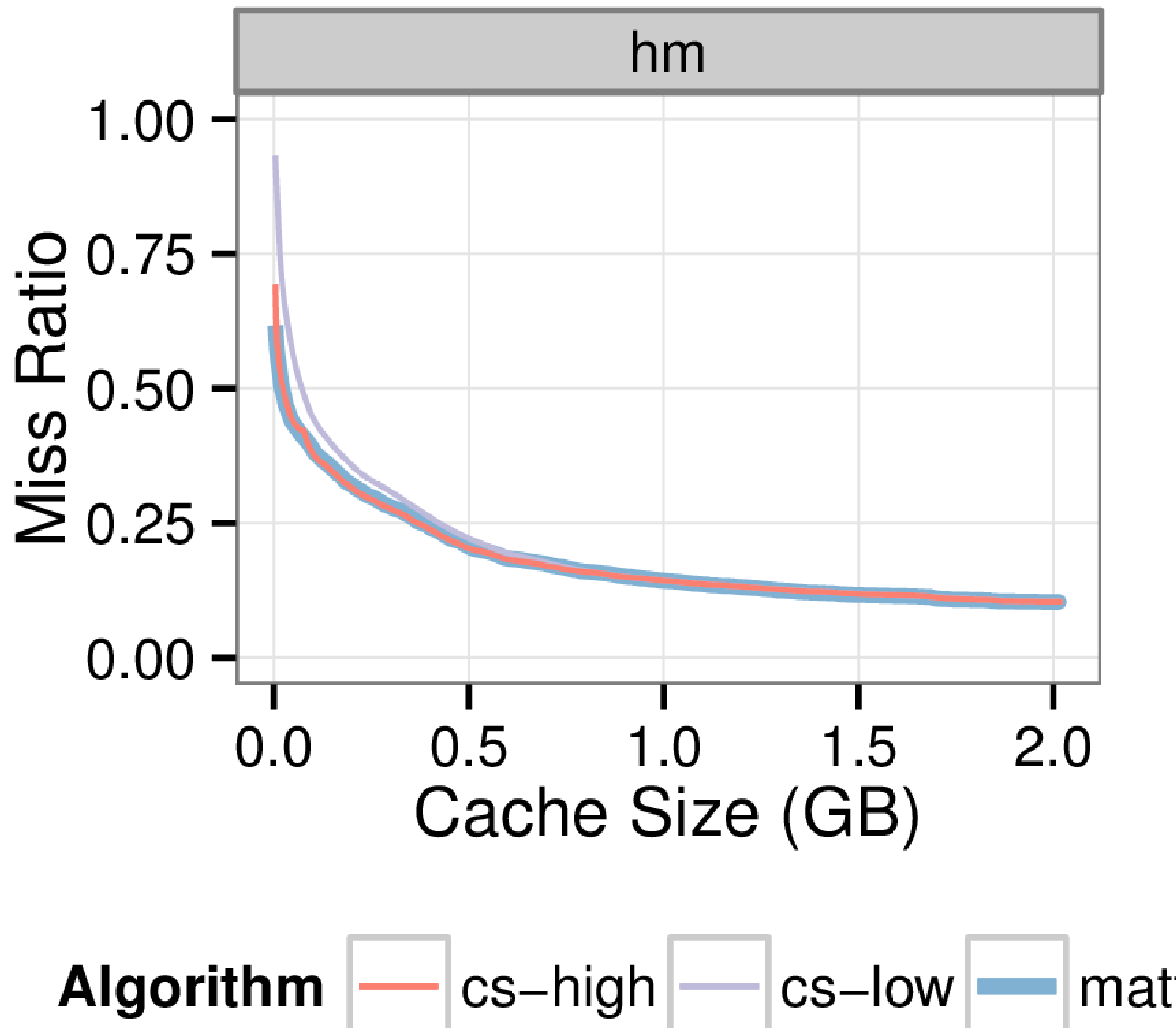
How Well Do They Work?



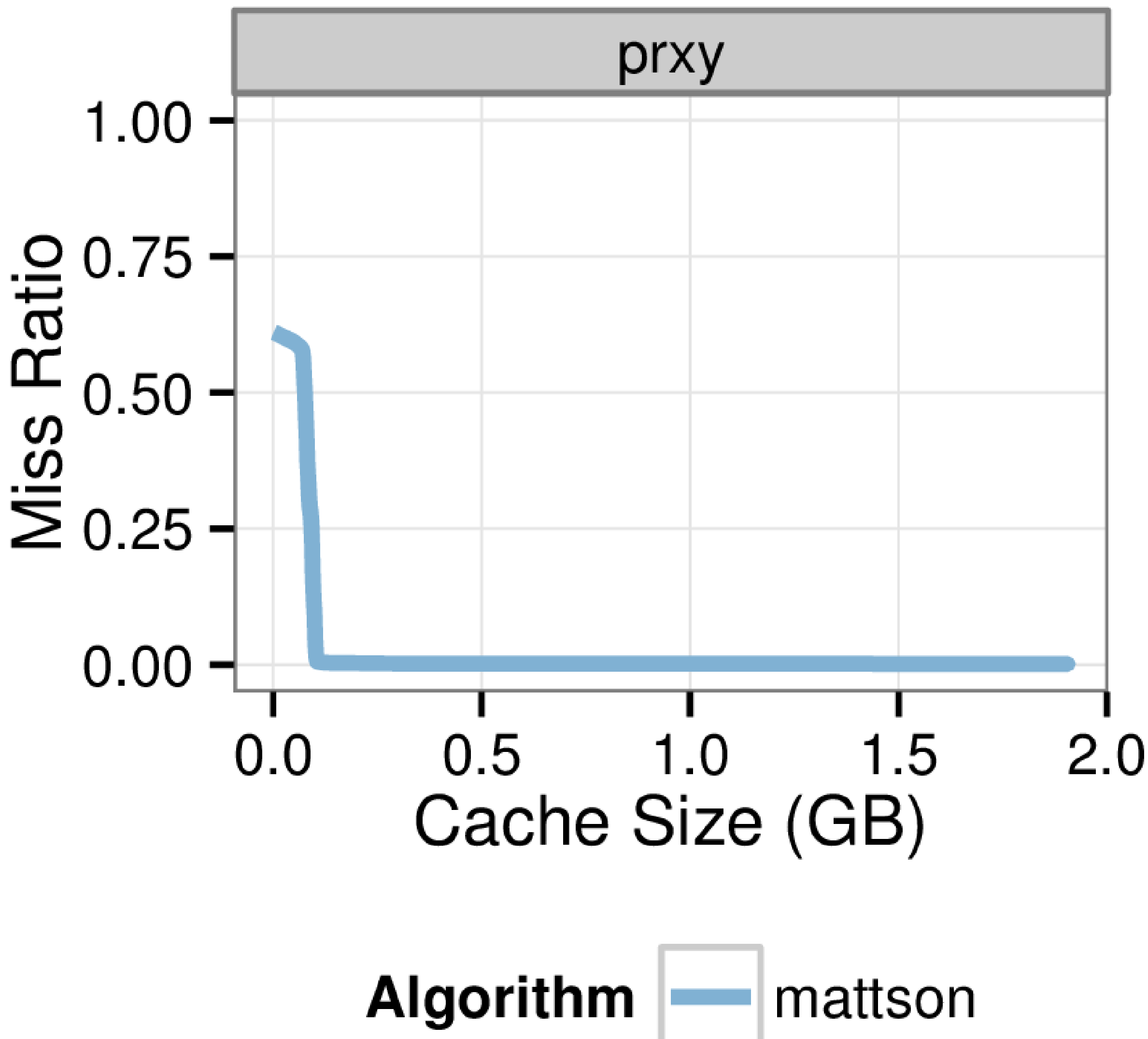
How Well Do They Work?



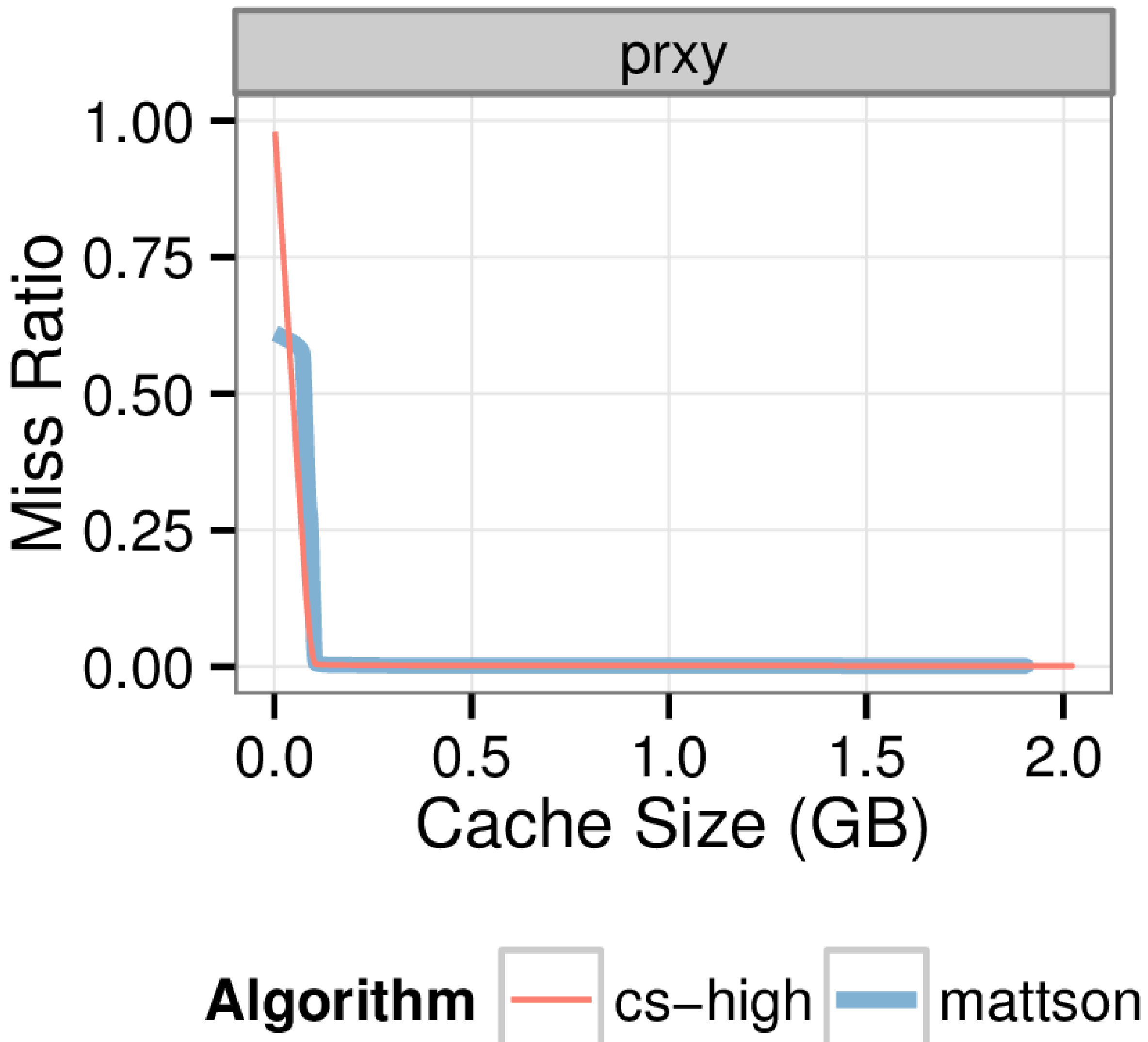
How Well Do They Work?



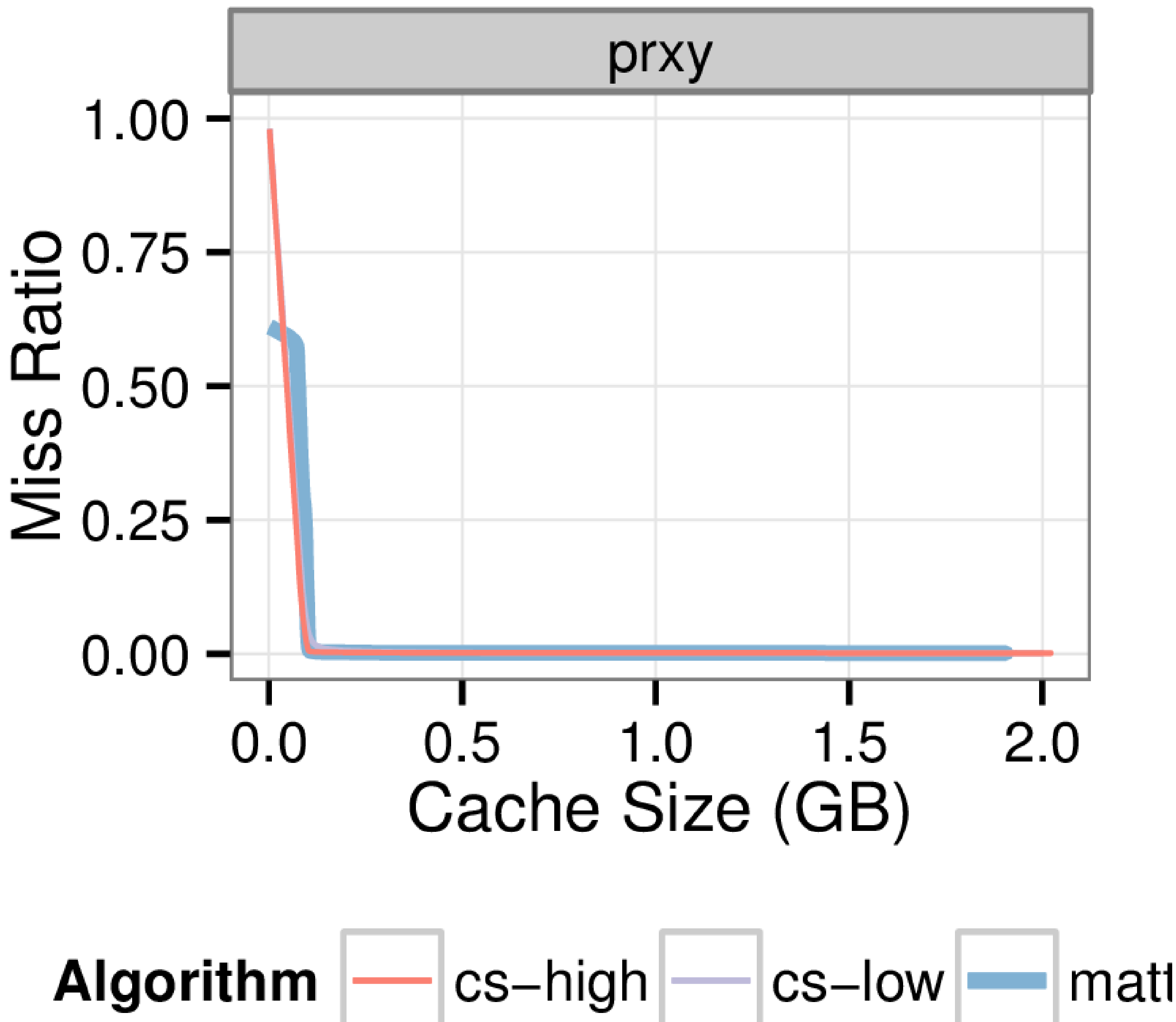
How Well Do They Work?



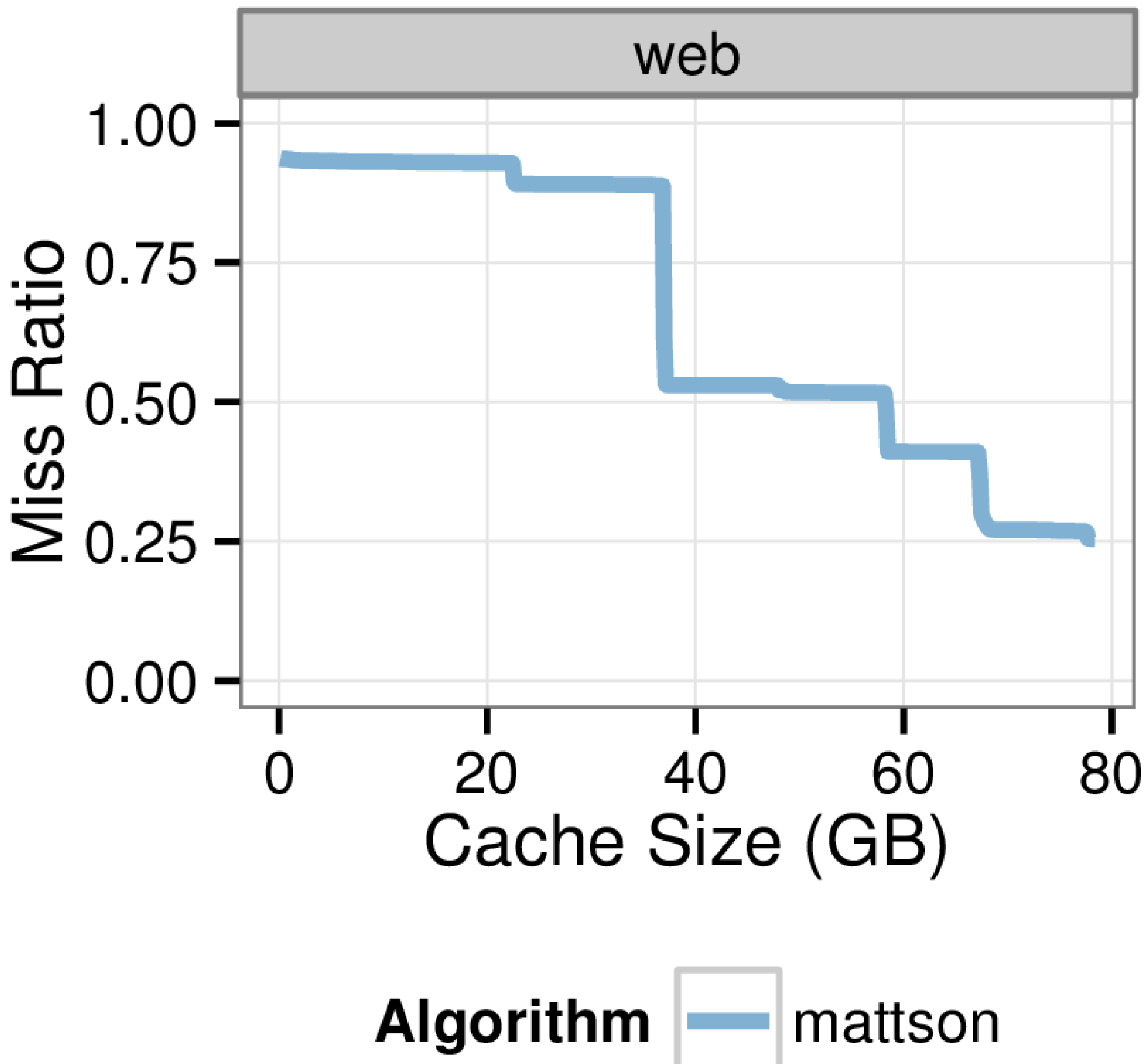
How Well Do They Work?



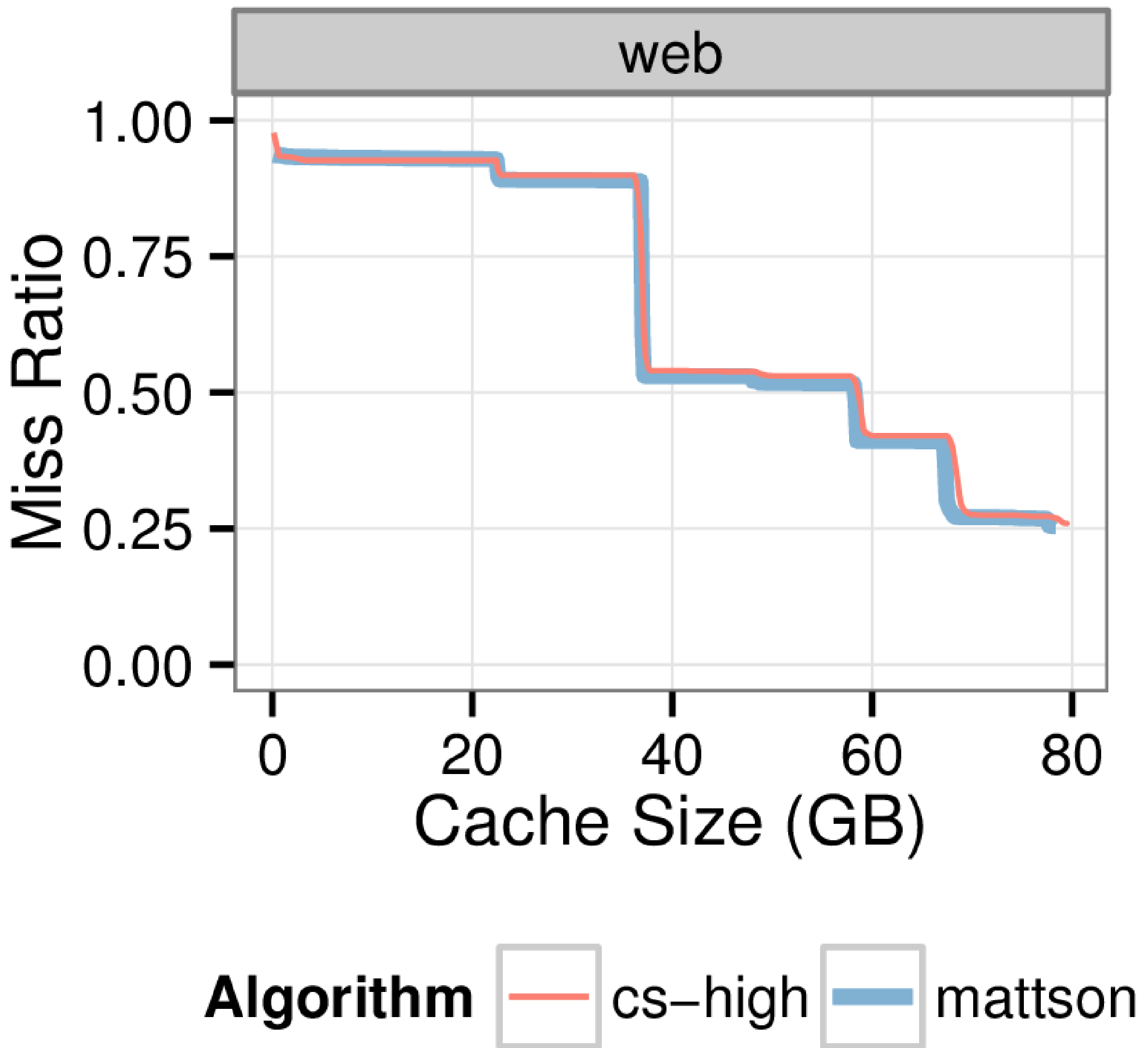
How Well Do They Work?



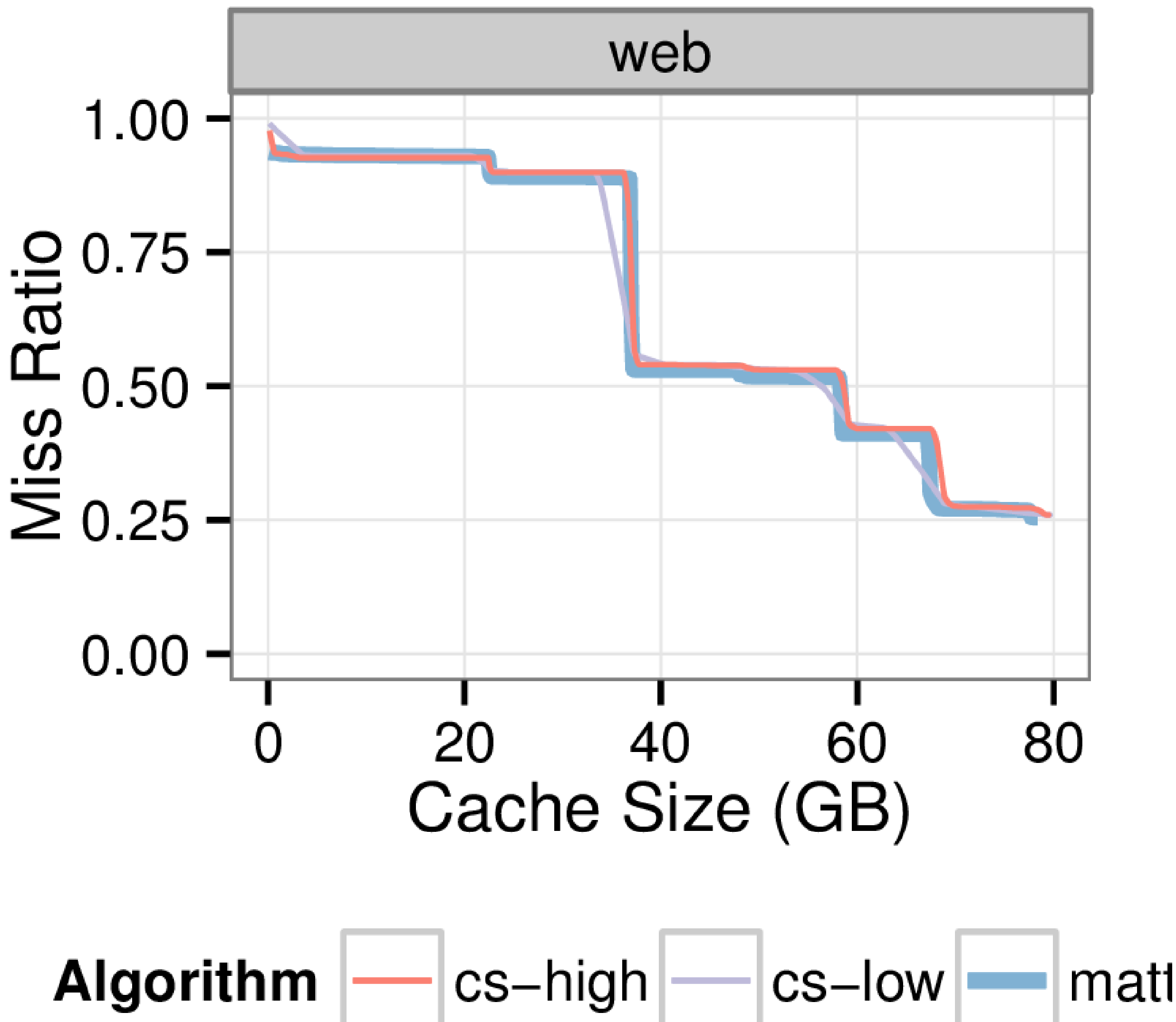
How Well Do They Work?



How Well Do They Work?



How Well Do They Work?



Conclusions

- Managing data can be data-intensive!
- Counter Stacks measure uniqueness over time
  - Low memory and storage overheads
  - Easy to capture, process, and store workload histories
- Used in production:
  - Collecting traces from the field
  - Making online placement decisions
  - Forecasting benefits of adding more hardware

Thanks!

Questions?

How Well Do They Work?