pRedis: Penalty and Locality Aware Memory Allocation in Redis

Cheng Pan,
Yingwei Luo, Xiaolin Wang
Dept. of CS, Peking University,
Peng Cheng Laboratory,
ICNLAB, Peking University

Zhenlin Wang
Dept. of Computer Science,
Michigan Technological University

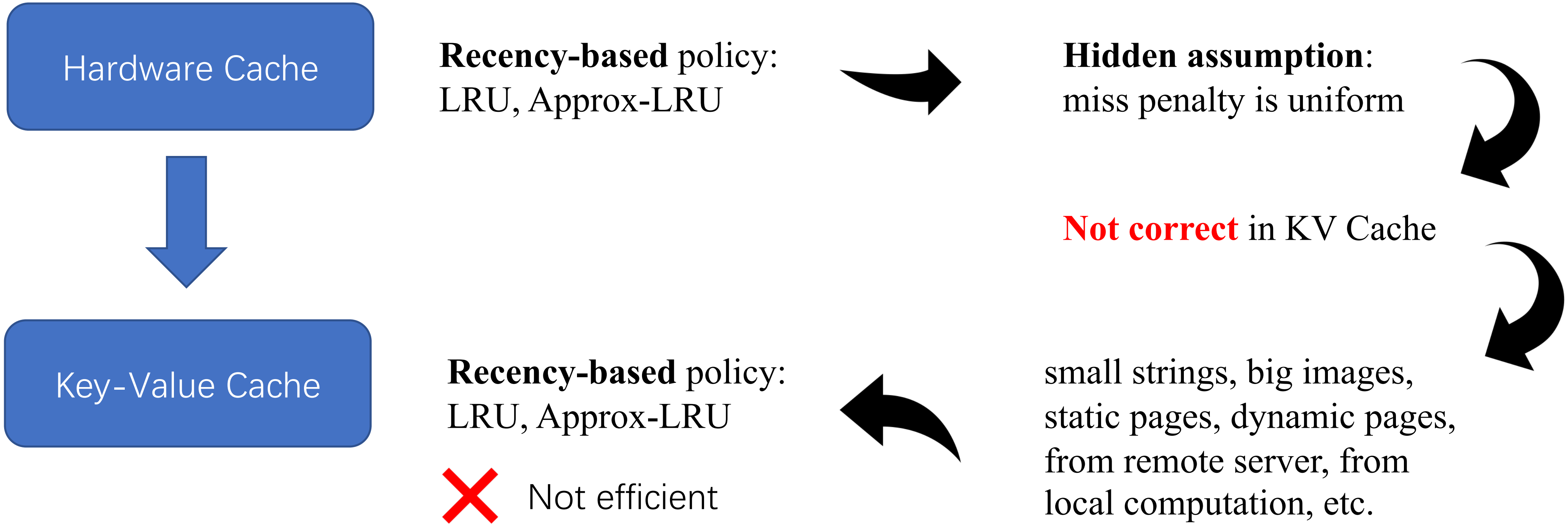
Outline

- Background
- Motivation Example
- pRedis: Penalty and Locality Aware Memory Allocation
- Long-term Locality Handling
- Evaluation
- Conclusion

Background

- In modern web services, the use of KV cache often help improve service performance.
  - Redis
  - Memcached

Background



Hardware Cache

Recency-based policy:
LRU, Approx-LRU

Hidden assumption:
miss penalty is uniform

Not correct in KV Cache

small strings, big images, static pages, dynamic pages, from remote server, from local computation, etc.

Key-Value Cache

Recency-based policy:
LRU, Approx-LRU

✗ Not efficient

Penalty Aware Policies

- The issue of miss penalty has drawn widespread attention:
  - GreedyDual [Young's PhD thesis, 1991]
  - GD-Wheel [EuroSys'15]
  - PAMA [ICPP'15]
  - Hyperbolic Caching [ATC'17]

cost (or miss penalty)

request count

$$p_i = c_i * \frac{n_i}{t_i}$$

residency time

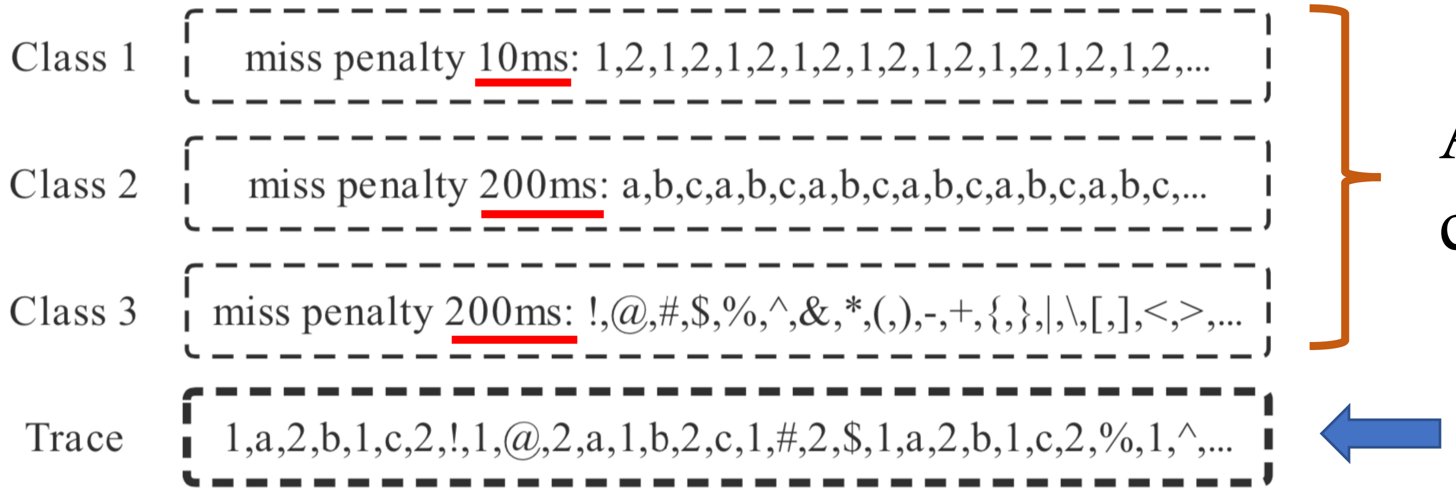
- **Hyperbolic Caching (HC)** delivers a better cache replacement scheme.
  - combines the miss penalty, access count and residency time of data item.
  - shows its advantage over other schemes on request service time.
  - but it is short of a global view of access locality

Outline

- Background
- Motivation Example
- pRedis: Penalty and Locality Aware Memory Allocation
- Long-term Locality Handling
- Evaluation
- Conclusion

Motivation Example

- We define the **miss penalty** as the time interval between the miss of a GET request and the SET of the same key immediately following the GET.



Class 1 miss penalty 10ms: 1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,...

Class 2 miss penalty 200ms: a,b,c,a,b,c,a,b,c,a,b,c,a,b,c,a,b,c,...

Class 3 miss penalty 200ms: !,@,#,$,%,^,&,*,(,),-,+,{,},|,\,[,],<,>,...

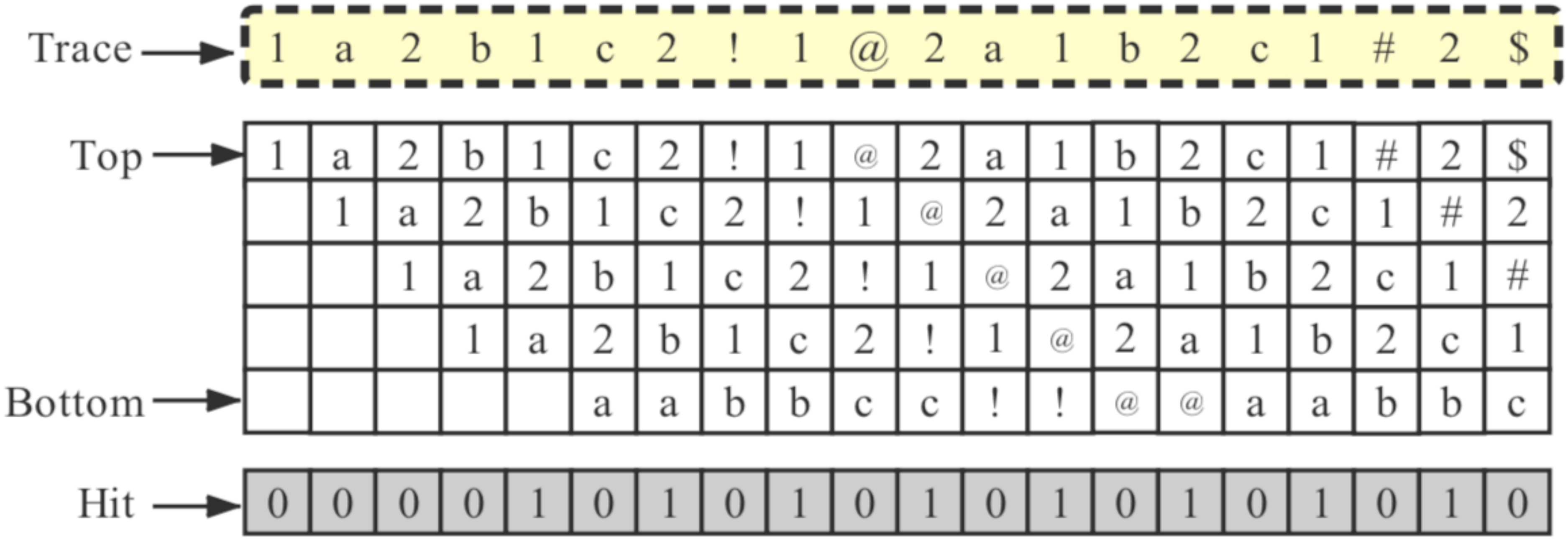
Trace 1,a,2,b,1,c,2,!,1,@,2,a,1,b,2,c,1,#,2,$,1,a,2,b,1,c,2,%,1,^,...

Access rates of these three classes are 5 : 3 : 2.

Combined trace.

Assume that each item's **hit time** is 1 ms, and the **total memory** size is 5.

Motivation Example – LRU Policy



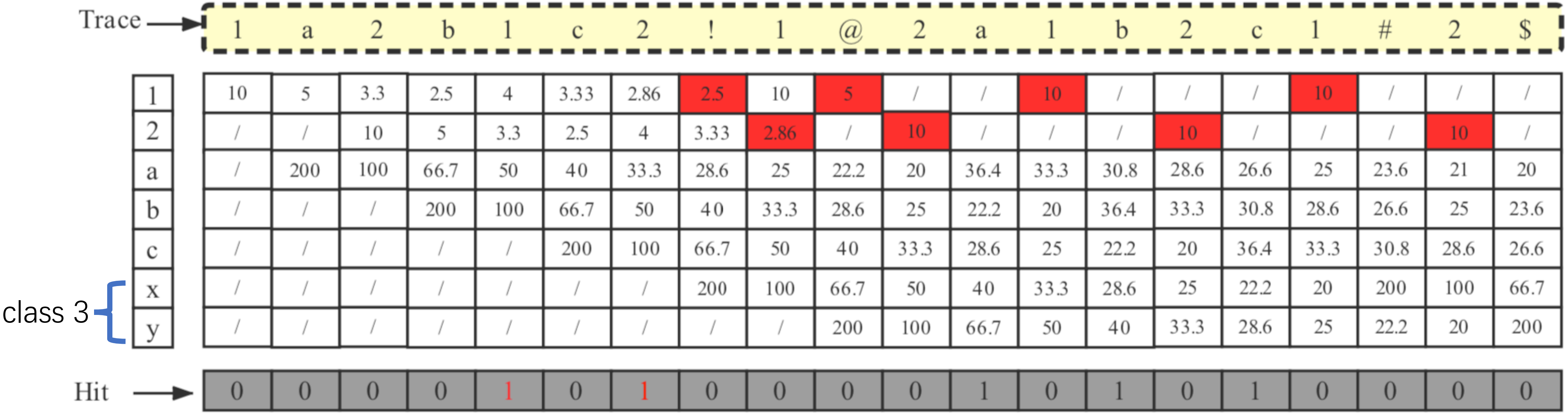
Trace →	1	a	2	b	1	c	2	!	1	@	2	a	1	b	2	c	1	#	2	$
Top →	1	a	2	b	1	c	2	!	1	@	2	a	1	b	2	c	1	#	2	$
		1	a	2	b	1	c	2	!	1	@	2	a	1	b	2	c	1	#	2
			1	a	2	b	1	c	2	!	1	@	2	a	1	b	2	c	1	#
				1	a	2	b	1	c	2	!	1	@	2	a	1	b	2	c	1
Bottom →						a	a	b	b	c	c	!	!	@	@	a	a	b	b	c
Hit →	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

Every access to class 1 will be a hit (except first 2 access).

Other accesses to class 2 and class 3 will all be misses.

Average request latency = $0.5*1 + 0.3*(200+1) + 0.2*(200+1) =$ **101 ms**.

Motivation Example – HC Policy



Trace	1	a	2	b	1	c	2	!	1	@	2	a	1	b	2	c	1	#	2	$
1	10	5	3.3	2.5	4	3.33	2.86	2.5	10	5	/	/	10	/	/	/	10	/	/	/
2	/	/	10	5	3.3	2.5	4	3.33	2.86	/	10	/	/	/	10	/	/	/	10	/
a	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	36.4	33.3	30.8	28.6	26.6	25	23.6	21	20
b	/	/	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	36.4	33.3	30.8	28.6	26.6	25	23.6
c	/	/	/	/	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	36.4	33.3	30.8	28.6	26.6
x	/	/	/	/	/	/	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	200	100	66.7
y	/	/	/	/	/	/	/	/	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	200
Hit	0	0	0	0	1	0	1	0	0	0	0	1	0	1	0	1	0	0	0	0

The elements in class 1 are chosen to evict except for their first load.

The newest class 3 elements stay in cache **even there is no reuse.**

Average request latency = $0.5 * (10 + 1) + 0.3 * 1 + 0.2 * (200 + 1) = 46$ ms

Motivation Example – pRedis Policy

- **Key Problems:**
  - **LRU:** doesn't consider miss penalty (e.g. class 2, class 3)
  - **HC:** doesn't consider locality (e.g. class 3)
- We combine **Locality** (Miss Ratio Curve, MRC) an*d **Miss Penalty**.

$$mr_1(c_1) = \begin{cases} 1 & c_1 < 2 \\ 0 & c_1 \geq 2 \end{cases} \qquad mr_2(c_2) = \begin{cases} 1 & c_2 < 3 \\ 0 & c_2 \geq 3 \end{cases} \qquad mr_3(c_3) = 1$$

$$W = 0.5 * mr_1(c_1) * 10 + 0.3 * mr_2(c_2) * 200 + 0.2 * mr_3(c_3) * 200, \quad \text{s.t.} \quad c_1 + c_2 + c_3 = 5$$

$c_1 = 2$, $c_2 = 3$, $c_3 = 0$, $W_{min} = 40$,

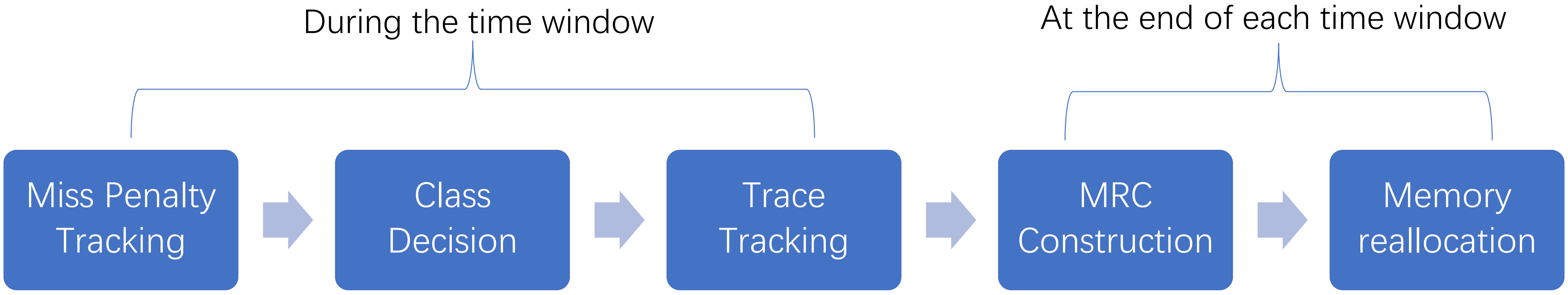
average request latency $= 0.5 * 1 + 0.3 * 1 + 0.2 * (200 + 1) = 41$ ms

Outline

- Background
- Motivation Example
- pRedis: Penalty and Locality Aware Memory Allocation
- Long-term Locality Handling
- Evaluation
- Conclusion

pRedis: Penalty and Locality Aware Memory Allocation

- In pRedis design, a workload can be divided into a series of fixed-size time windows (or phases). In a time window:



Track miss penalty

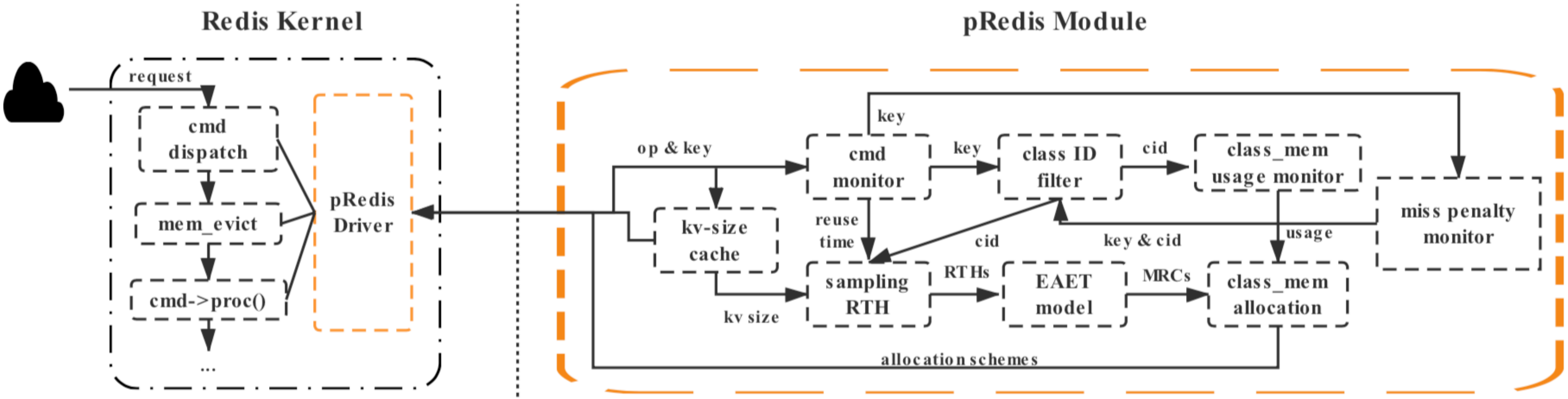
Divide penalty into classes

Generate sub-trace for each class

Use EAET Model

Use dynamic programming

pRedis System Design



Penalty Class ID Filter

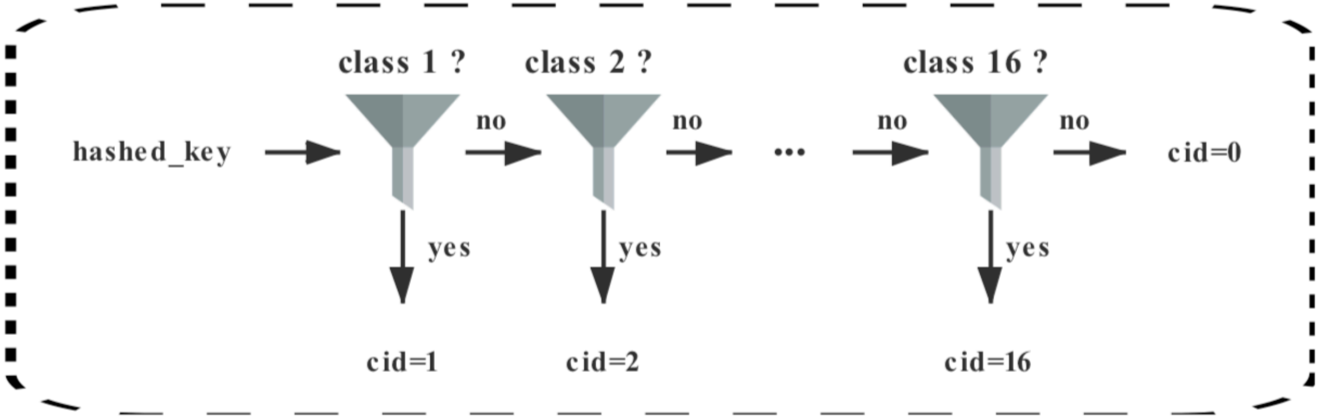
EAET Model

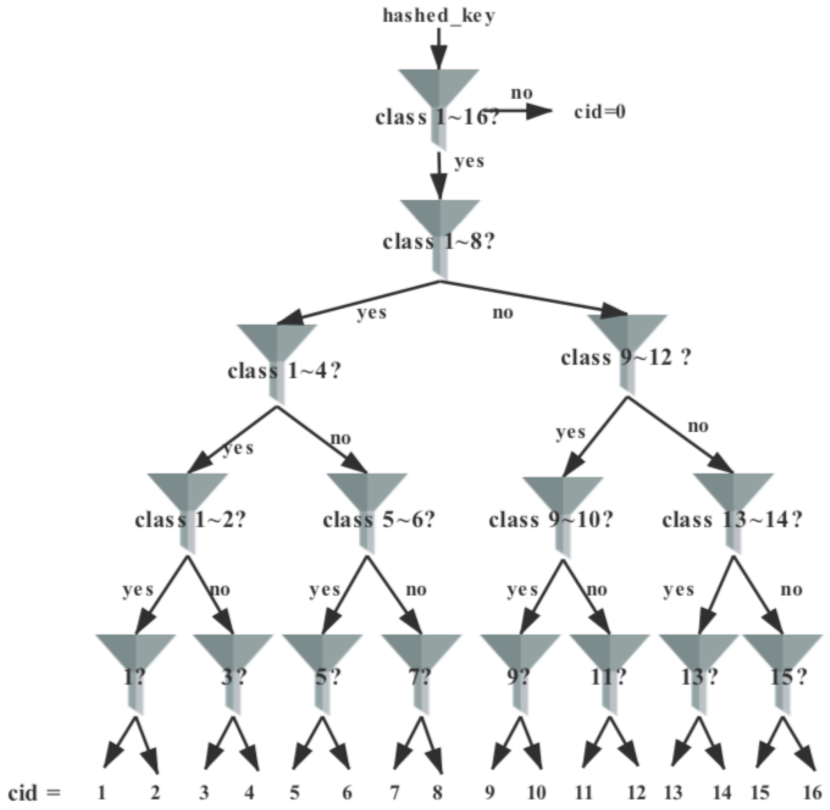
Class Memory Allocation

pRedis – Penalty Class I

- Track the miss penalty for each KV.
- Divide them into different classes.
- But how to maintain these information effici
  - store an additional field for each stored key? too

Penalty Class ID Filter





1 million keys
Pr(false positive) = 0.01

Overhead: 1 MB

pRedis – Penalty Class ID Filter

- Two different ways to decide the Penalty Class ID:
- 1) **Auto-detecting: pRedis(auto)**
  - set the range of each penalty class in advance.
  - each KV will be automatically assigned to the class it belongs to based on the measured miss penalty.

- 2) **User-hinted: pRedis(hint)**
  - provides an interface for user to specify the class of an item.
  - aggregates the latency of all items of a penalty class in a time period.

pRedis – EAET Model

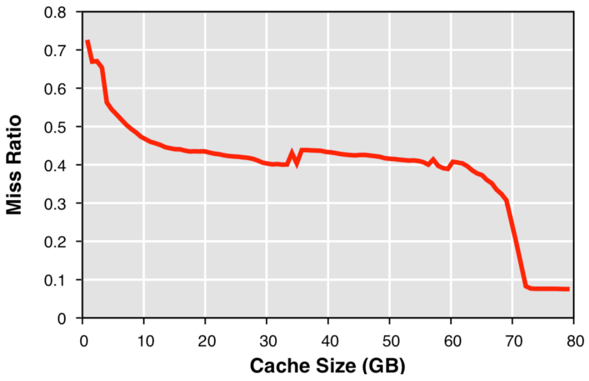
- Enhanced AET (EAET) model is a cache locality model (APSys 2018):
  - support read, write, update, deletion operations
  - support non-uniform object sizes

Input: KVs access workload

EAET Modeling

Output: Miss Ratio Curve (MRC)

```
SET key1 123
GET key1
SET key2 "test"
GET key2
...
```



pRedis – Class Memory Allocation

- If we allocate penalty class i with M_i memory units, then this class's overall miss penalty (or latency) MP_i can be estimated as:

$$MP_i = mr_i(M_i) * p_i * N_i \longrightarrow \text{access count}$$

miss rate given memory size M_i

average miss penalty

- Our final goal:

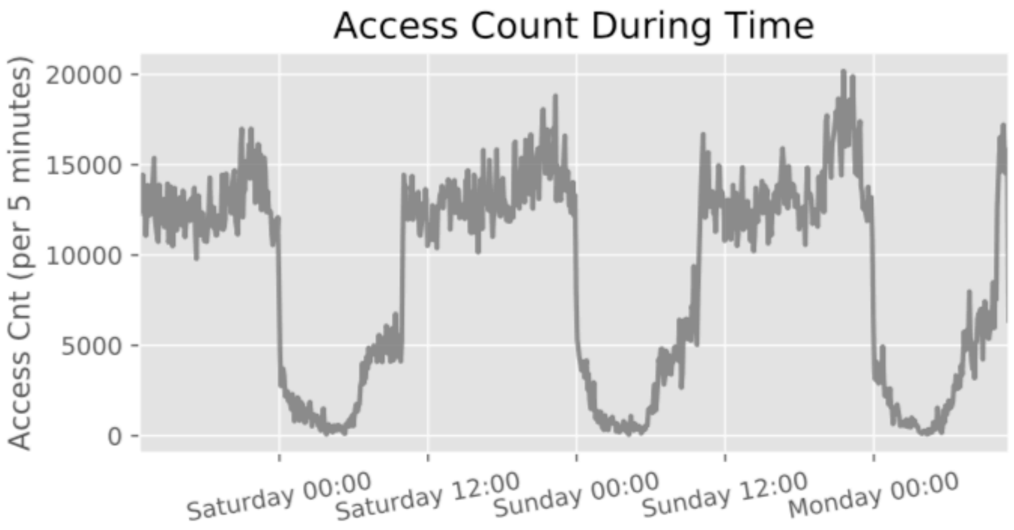
$$min \sum_{i=1}^{n} MP_i = min \sum_{i=1}^{n} mr_i(M_i) * p_i * N_i \qquad s.t. \sum_{i=1}^{n} M_i = M$$

Dynamic programming to obtain the optimal memory allocation: enforced through object replacements.

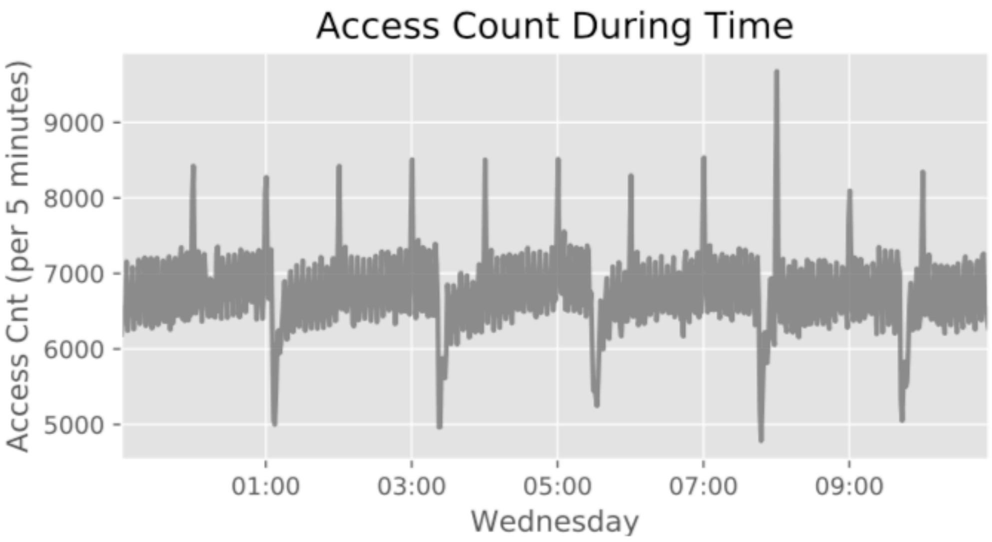
Outline

- Background
- Motivation Example
- pRedis: Penalty and Locality Aware Memory Allocation
- Long-term Locality Handling
- Evaluation
- Conclusion

Long-term Locality Handling



(a) Periodic Pattern



(b) Non-periodic Pattern

Periodic Pattern: The number of requests changes periodically over time, and the long-term reuse is accompanied by the emergence of request peaks.

Non-Periodic Pattern: The number of requests remains relatively stable over time, or there are no long-term reuses.

Auto Load/Dump Mechanism

- Obviously, when these two types of workloads share Redis,
  - with the LRU strategy, the memory usage of the two types of data will change during the access peaks and valleys.
  - the passive evictions during the valley periods and the passive loadings (because of GET misses) during the peak periods will cause considerable latency.

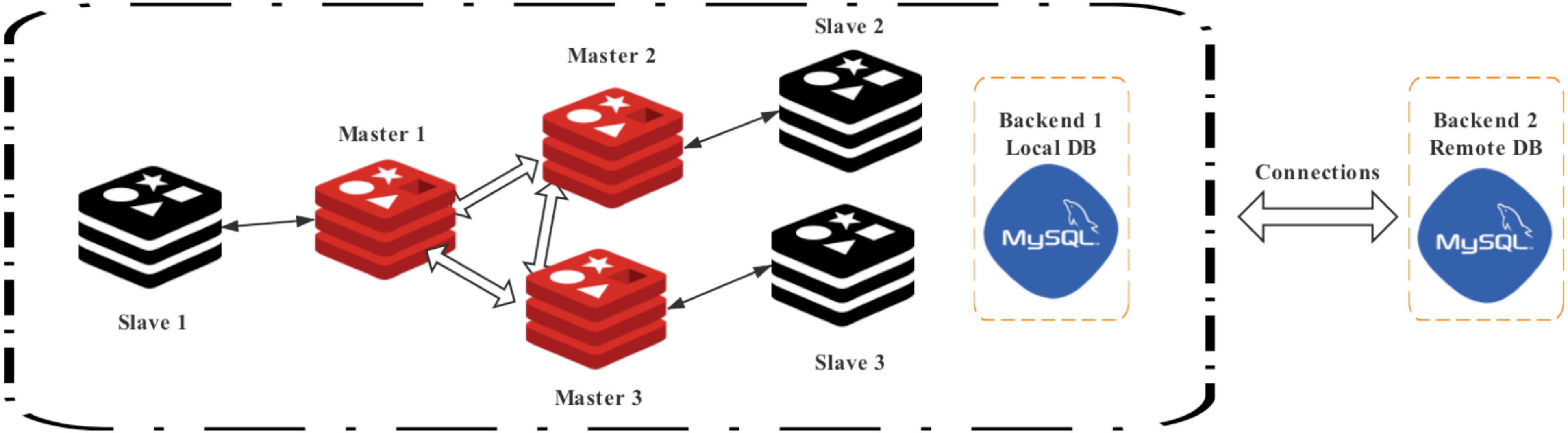
- Auto load/dump mechanism
  - Proactively dump some of the memory to a local SSD (or hard drives) when a valley arrives.
  - Proactively load the previously dumped content before arrival of a peak.

Outline

- Background
- Motivation Example
- pRedis: Penalty and Locality Aware Memory Allocation
- Long-term Locality Handling
- Evaluation
- Conclusion

Experimental Setup

- We evaluate pRedis and other strategies using **six cluster nodes**.
- Each node: Intel(R) Xeon(R) E5-2670 v3 2.30GHz processor with 30MB shared LLC and 200 GB of memory, the OS is Ubuntu 16.04 with Linux-4.15.0.



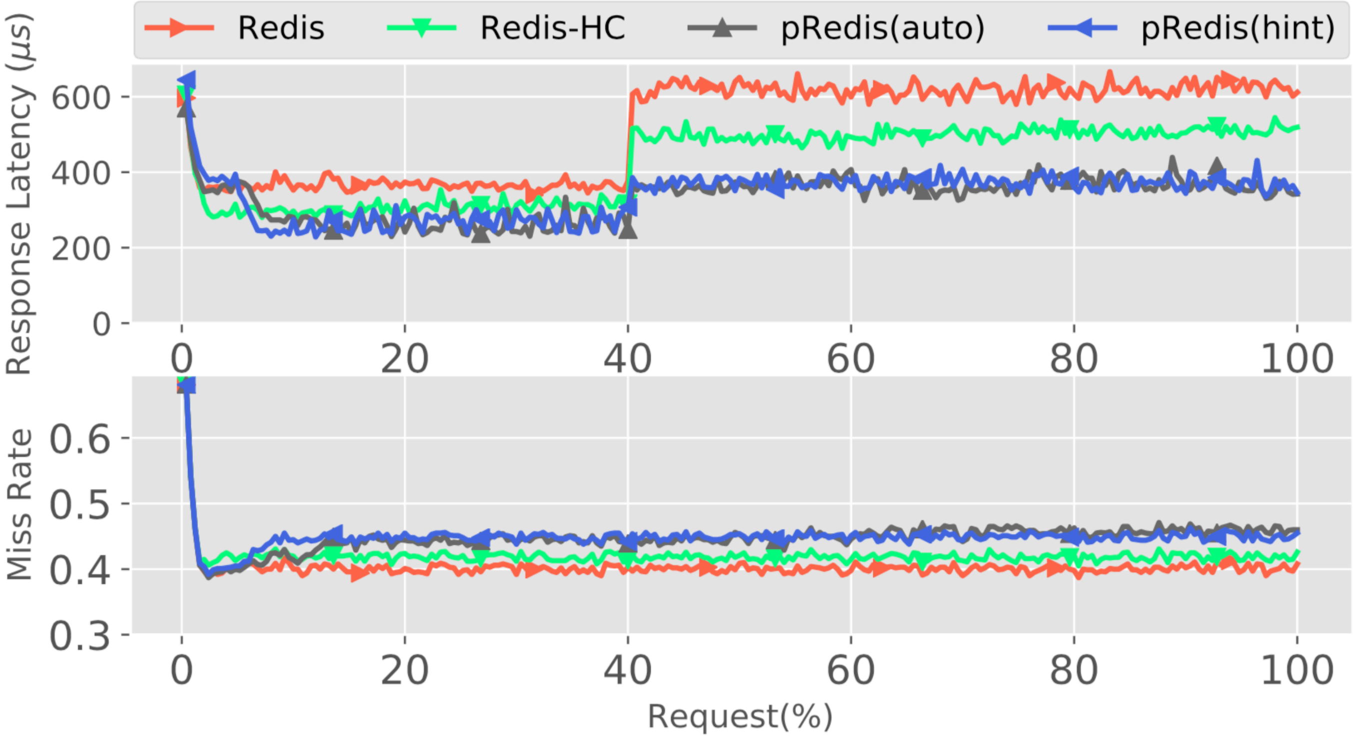
Latency – Experimental Design

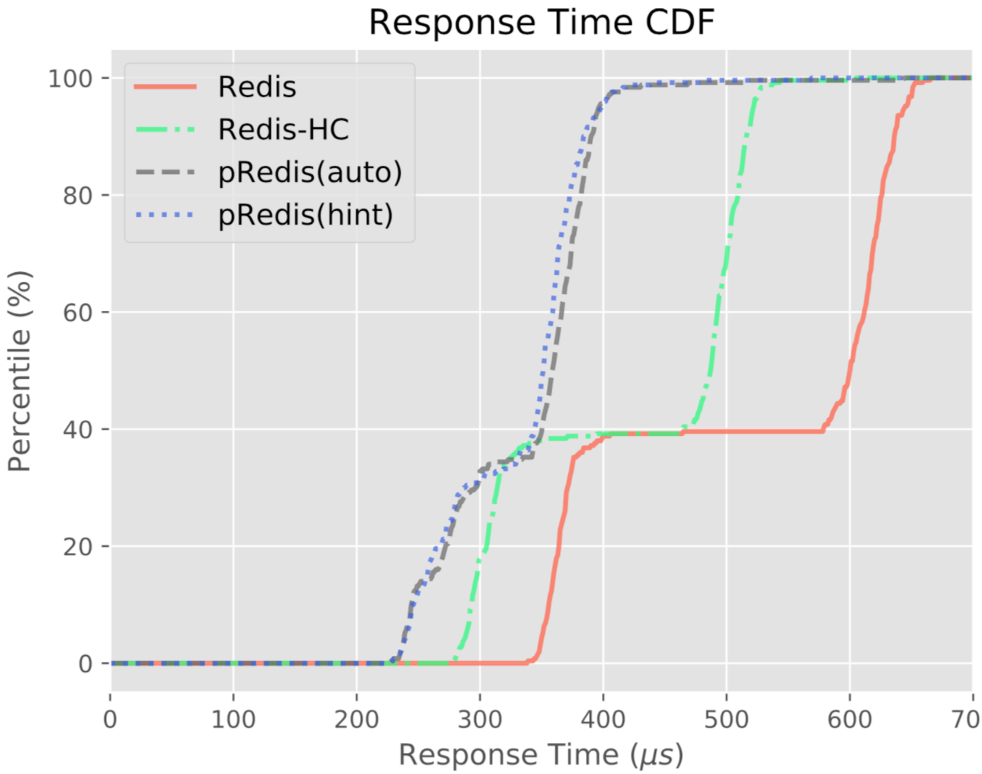
- We use the MurmurHash3 function to randomly distribute the data to two backend MySQL servers, one **local** and one **remote**.
  - access latency are ~120 µs and ~1000 µs, respectively.

- We set a series of ranges, [1µs, 10µs), [10µs, 30µs), [30µs, 70µs), ..., [327670µs, 655350µs), 16 penalty classes in total.

- Additionally, in order to compare two different variants of pRedis, we run **a stress test** (mysqlslap) in the remote MySQL server after the workload reaches 40% of the trace.
  - causing the remote latency to rise from ~1000 µs to ~2000 µs.

Latency – YCSB Workload A





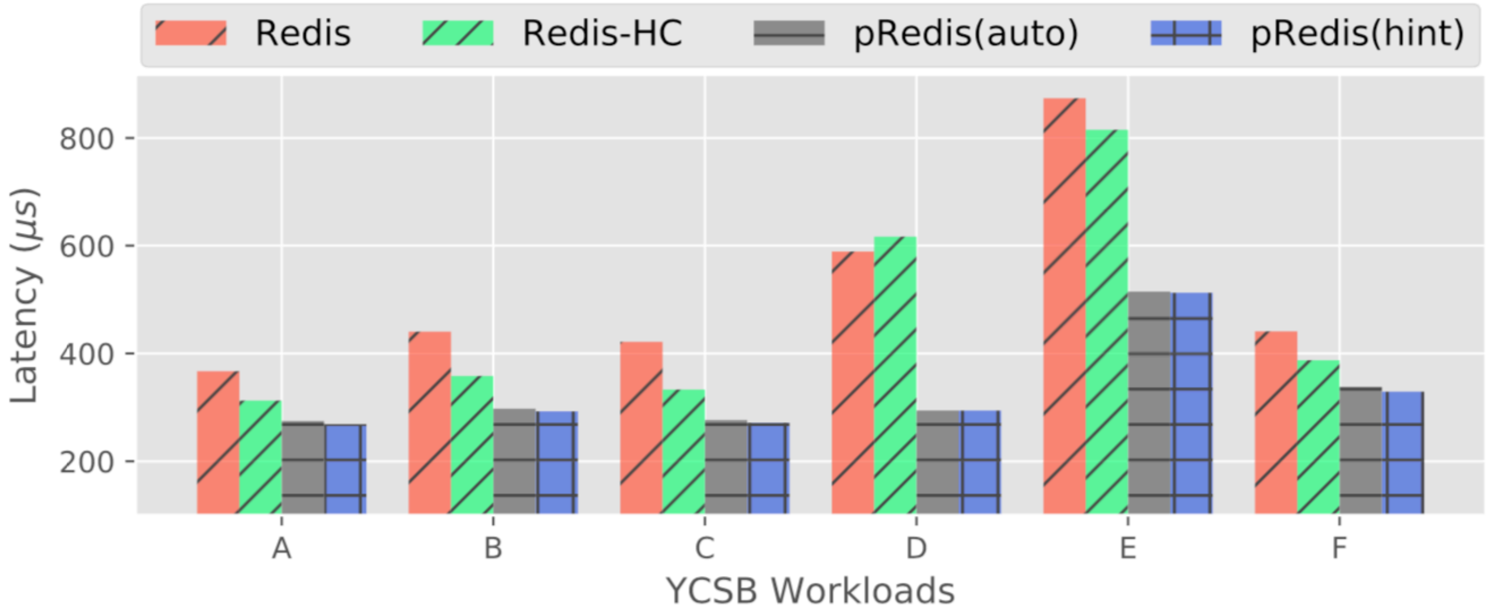
	Redis	Redis-HC	pRedis(auto)	pRedis(hint)
Avg (μs)	519.0	425.4	338.1	332.7

pRedis(auto) is 34.8% and 20.5% lower than Redis and Redis-HC, pRedis(hint) cuts another 1.6%.

Latency

- We summarize the average response latency of the six YCSB workloads in the right figure.

- pRedis(auto) vs. Redis-HC: 12.1% ~ 51.9%.
- pRedis(hint) vs. Redis-HC: 14.0% ~ 52.3%.

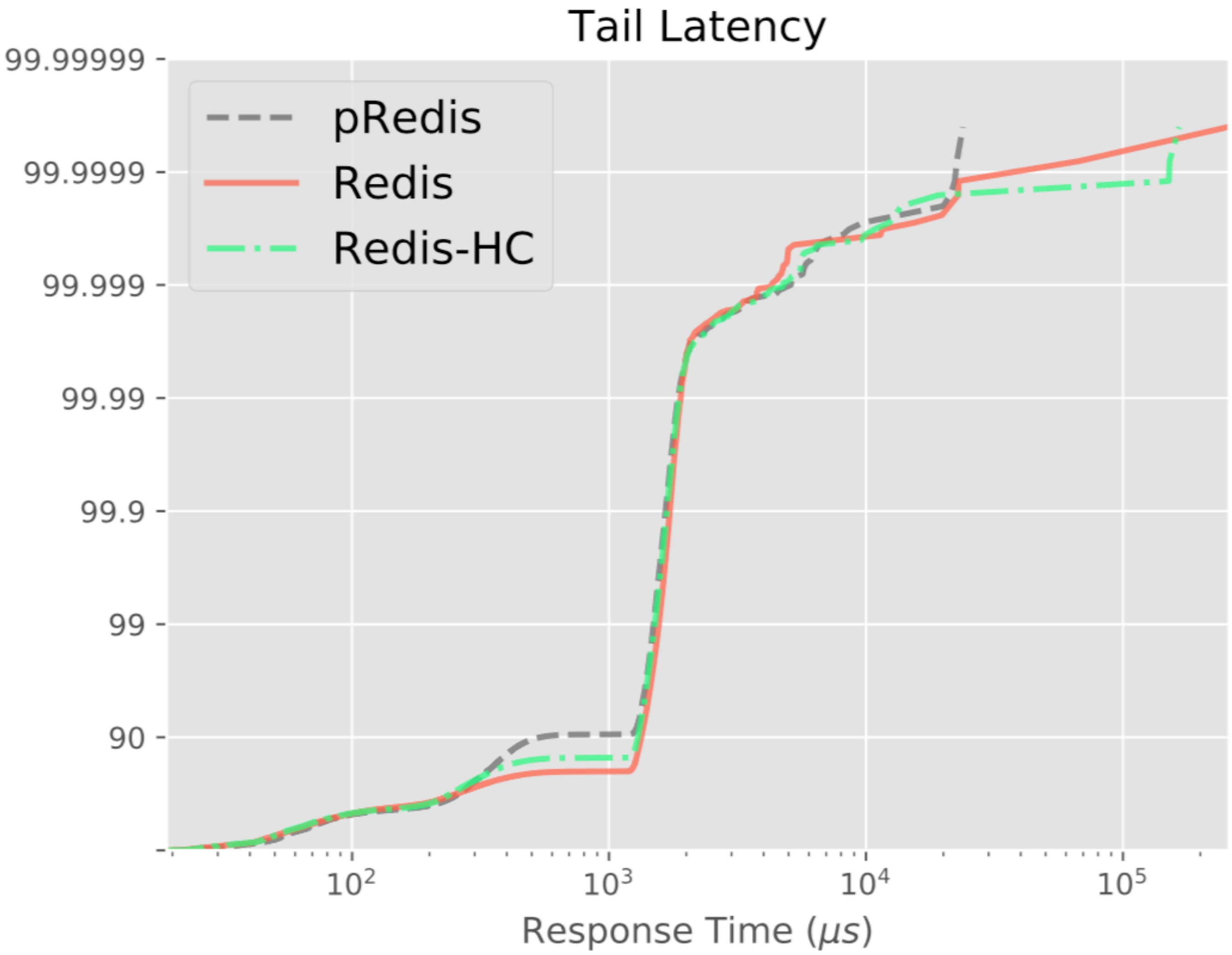


	pRedis(auto)					
	A	B	C	D	E	F
Redis	25.7%	33.6%	34.9%	49.7%	40.1%	25.2%
Redis-HC	12.8%	18.3%	17.6%	51.9%	35.9%	12.1%

	pRedis(hint)					
	A	B	C	D	E	F
Redis	26.8%	33.6%	35.6%	50.1%	41.3%	26.8%
Redis-HC	14.0%	18.3%	18.5%	52.3%	37.1%	15.0%

Tail Latency

- YCSB Workload A
- using pRedis(hint)
- **0~99.99%:** pRedis are the same as or lower than Redis and Redis-HC.
- **99.999%~99.9999%:** three methods have their pros and cons.
- **next 0.00009%:** pRedis performs better than others.

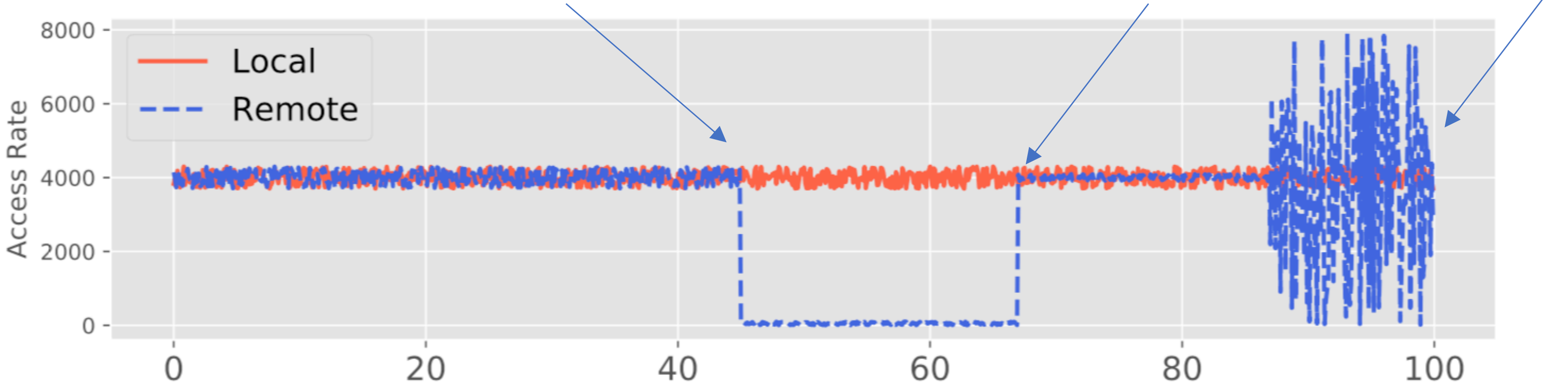


	min	90th	95th	99th	99.9th	99.99th
Redis	19	1335	1415	1561	1712	1789
Redis-HC	19	1295	1369	1506	1664	1766
pRedis	19	540	1349	1490	1647	1747

Auto Dump/Load in Periodic Pattern

- We use two traces from the collection of Redis traces
  - one trace has periodic pattern (the e-commerce trace),
  - the other has non-periodic pattern (a system monitoring service trace).

- The data objects are also distributed to both the local and remote MySQL databases.



Remote access pause

Remote access pause

access thrash

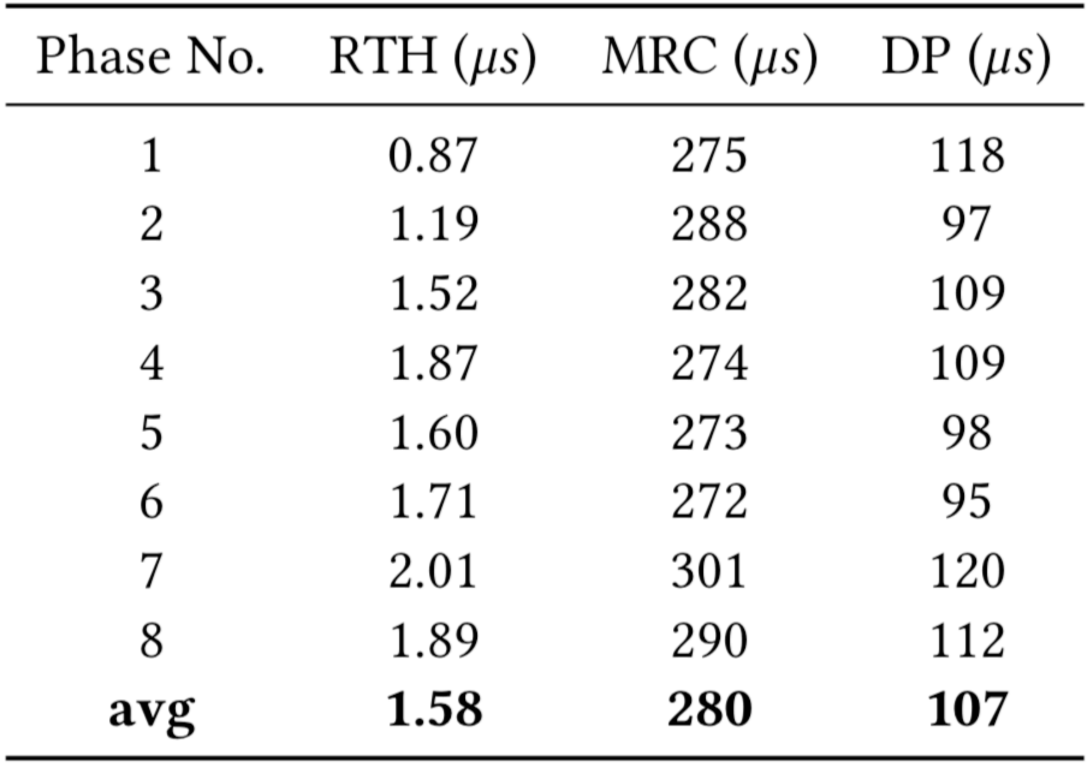
Auto Dump/Load in Periodic Pattern

- In general, the use of auto-dump/load can smooth the access latency caused by periodic pattern switching.
- pRedis(with d/l) vs. Redis-HC: 13.3%
- pRedis(with d/l) vs. pRedis(without d/l): 8.4%

	Redis	Redis-HC	pRedis (with d/l)	pRedis (without d/l)
Avg (μs)	317.3	277.2	240.3	262.6

Overhead

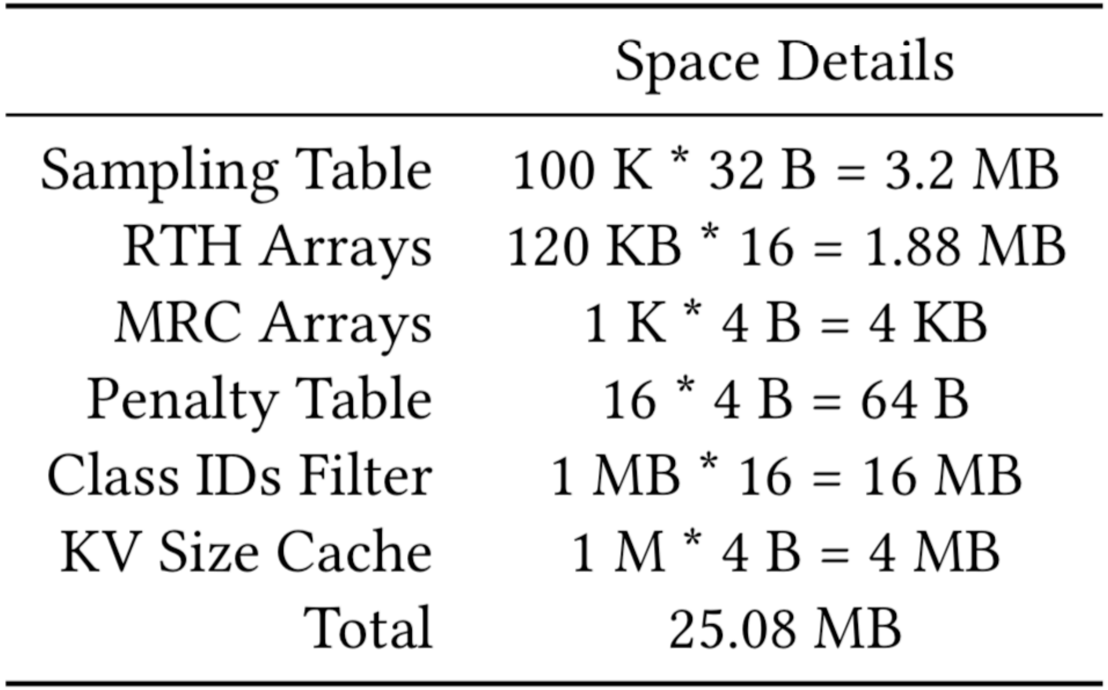
Time Overhead



Phase No.	RTH (μs)	MRC (μs)	DP (μs)
1	0.87	275	118
2	1.19	288	97
3	1.52	282	109
4	1.87	274	109
5	1.60	273	98
6	1.71	272	95
7	2.01	301	120
8	1.89	290	112
avg	**1.58**	**280**	**107**

RTH sampling time takes about 0.01% of access time, MRC construction and re-allocation DP occur at the end of each phase (in minutes), that’s negligible.

Space Overhead



	Space Details
Sampling Table	100 K * 32 B = 3.2 MB
RTH Arrays	120 KB * 16 = 1.88 MB
MRC Arrays	1 K * 4 B = 4 KB
Penalty Table	16 * 4 B = 64 B
Class IDs Filter	1 MB * 16 = 16 MB
KV Size Cache	1 M * 4 B = 4 MB
Total	25.08 MB

working set is 10 GB (using YCSB Workload A), total space overhead is 25.08 MB, 0.24% of the total working set size, that’s acceptable.

Outline

- Background
- Motivation Example
- pRedis: Penalty and Locality Aware Memory Allocation
- Long-term Locality Handling
- Evaluation
- Conclusion

Conclusion

- We have presented a systematic design and implementation of pRedis:
  - A penalty and locality aware memory allocation scheme for Redis.
  - It exploits the data locality and miss penalty, in a quantitative manner, to guide the memory allocation in Redis.

- pRedis shows good performance:
  - It can predict MRC for each penalty class with a 98.8% accuracy and has the ability to adapt the phase change.
  - It outperforms a state-of-the-art penalty aware cache management scheme, HC, by reducing 14~52% average response time.
  - Its time and space overhead is low.

PEKING
UNIVERSITY

Thanks for your attention !

Q & A

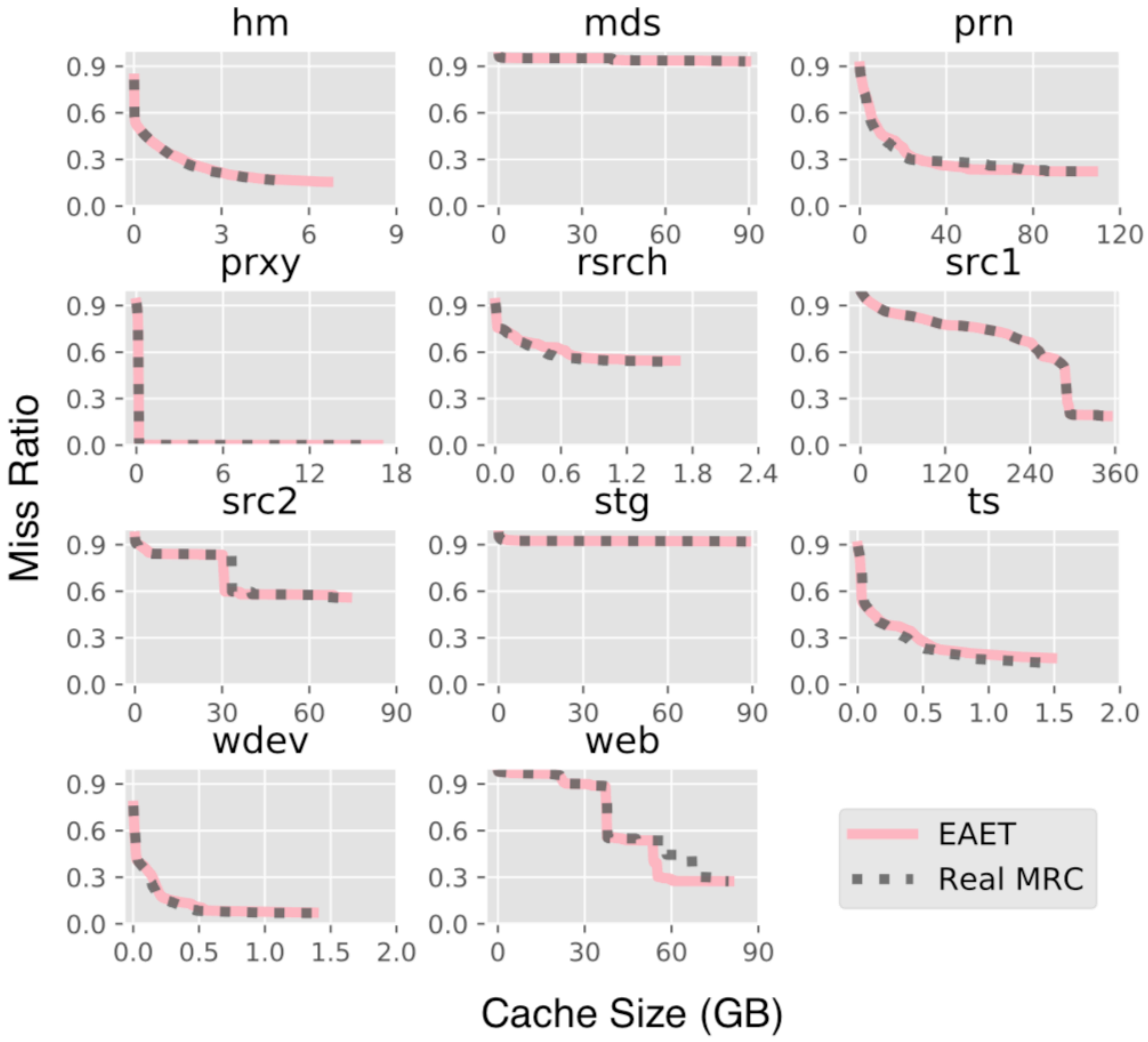
pancheng@pku.edu.cn

Workloads

- MSR Workloads
  - One week of block I/O traces from the Microsoft Research Cambridge Enterprise servers
- YCSB Workloads
  - A framework and common set of workloads for evaluating the performance of different "key-value" and "cloud" serving stores.
- A Collection of Real-world Redis Workloads
  - They are obtained from a set of Redis servers used for E-commerce, cluster performance monitoring, and other services.
- Memtier Benchmark
  - A high throughput benchmarking tool for Redis and Memcached.

MRC Accuracy

- pRedis relies on accurate MRCs.
- We compare the pRedis MRC, obtained by EAET using 1% set sampling, with the actual MRC, obtained by measuring the full-trace reuse distances.
- The average absolute error of EAET is **1.2%**, which is accurate enough.

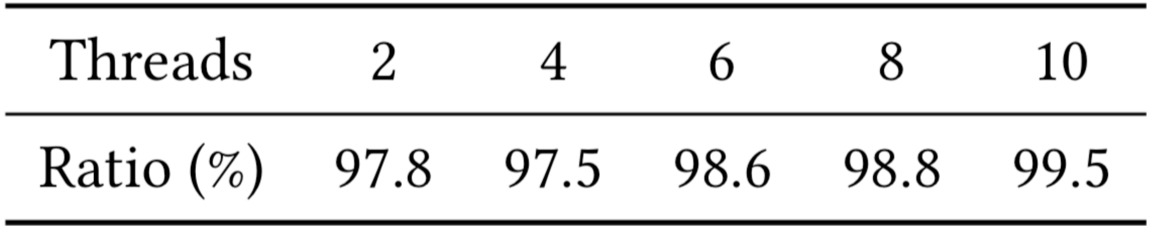


	hm	mds	prn	prxy	rsrch	src1
abs err	0.58%	0.06%	1.9%	0.06%	1.6%	0.05%
	src2	stg	ts	wdev	web	**avg**
abs err	1.2%	0.07%	3.0%	1.1%	3.6%	**1.2%**

Throughput – Worst Case

- A stress test using Memtier benchmark
- The memory-limit is set to ∞, so all of the GET queries will be hits.
- We setup 2 to 10 threads to send requests, each thread will drive 50 clients, each client send 1000000 requests total. The ratio of SET and GET is 1:10, and default data size is 32 bytes.

Table: pRedis vs. Redis on Throughput



Threads	2	4	6	8	10
Ratio (%)	97.8	97.5	98.6	98.8	99.5

The average degradation is **only 1.5%**