mPart: Miss Ratio Curve Guided Partitioning in Key-Value Stores

Daniel Byrne djbyrne@mtu.edu
Nilufer Onder nilufer@mtu.edu
Zhenlin Wang zlwang@mtu.edu

Department of Computer Science
Michigan Technological University

International Symposium on Memory Management
June 2018

Introduction - Multi-Tenant Key-Value Stores

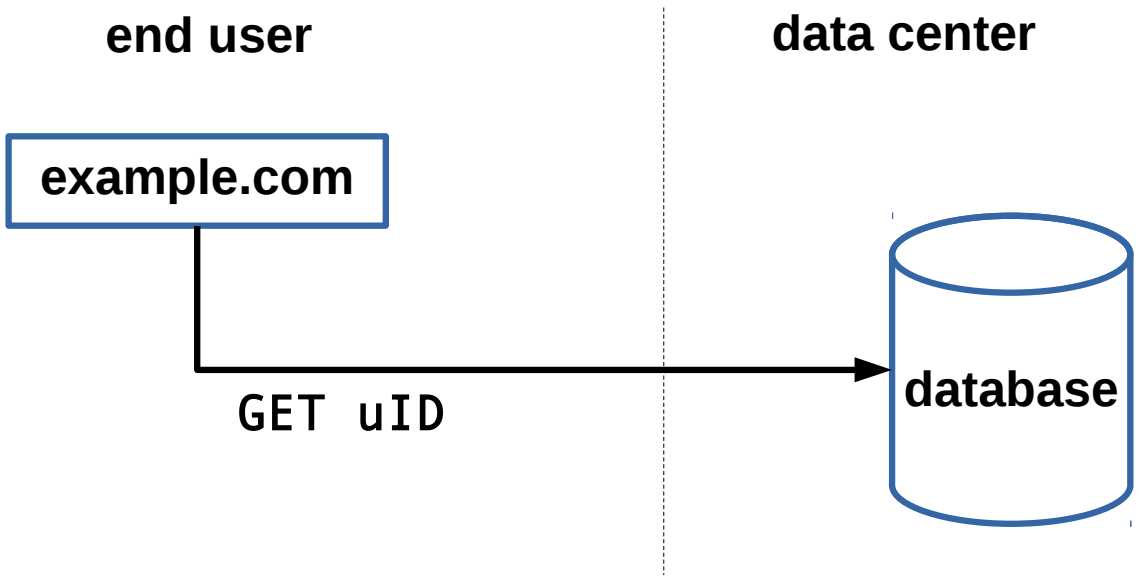
Background - Estimating the Miss Ratio Curve

mPart: Miss Ratio Curve Guided Partitioning

Evaluation

Conclusion

Database Access Times Are Slow

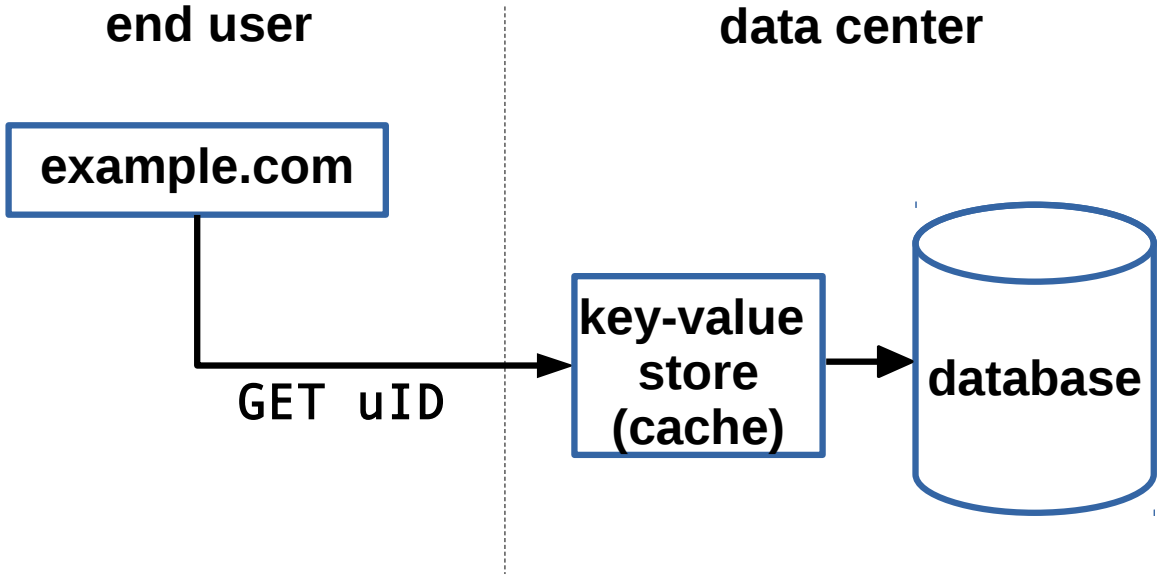


time to fetch data from database:
~1,000 µs

Improve Access Time with Key-Value Store

- Key-Value stores reside in server's main memory
- Data is stored as: `key,value`
- To retrieve an item: `GET user.name`
- To store/update an item: `SET user.name djbyrne`
- Examples: *memcached* and *Redis*

Caching Improves End-to-End Latency



time to fetch data from cache: ~10 µs

Impact of Hit Ratio

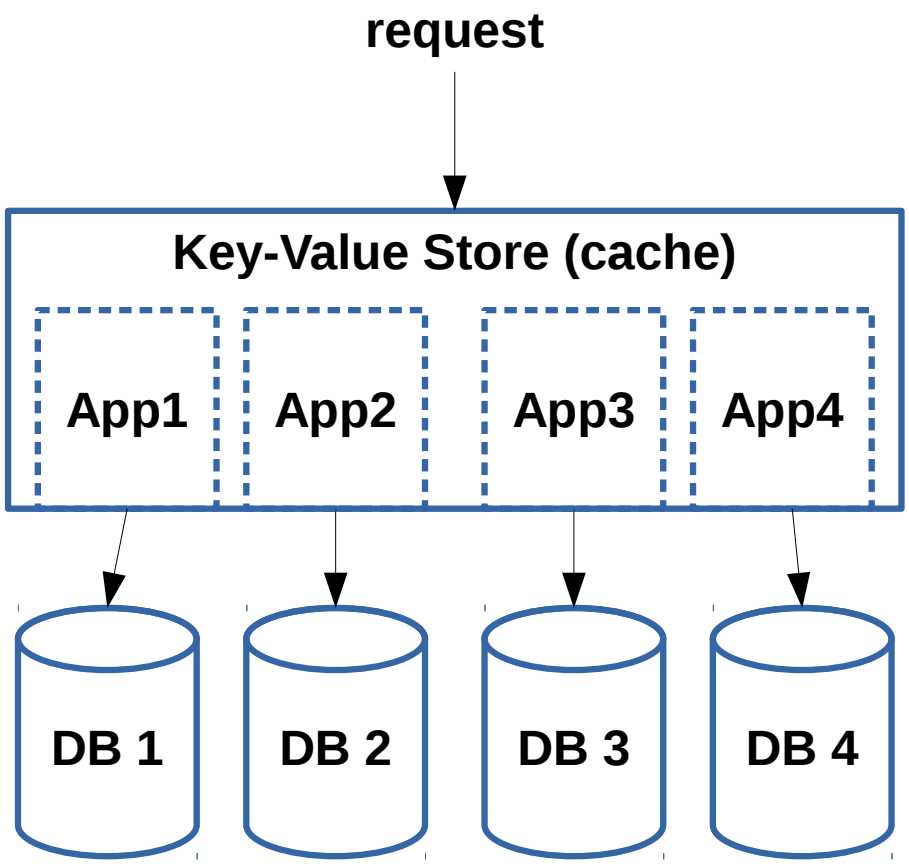
		Access Time (µs)		
System	**Hit Ratio**	**Cache**	**Database**	**End-to-End**
A	95%	10	1,000	59.5
B	96%	10	1,000	49.6

Impact of Hit Ratio

		Access Time (µs)		
System	**Hit Ratio**	**Cache**	**Database**	**End-to-End**
A	95%	10	1,000	59.5
B	96%	10	1,000	49.6

1% increase in hit ratio $\rightarrow$ 16% increase in end-to-end!

Data Centers Host Multiple Apps



Sharing the Cache with Multiple Apps

- Efficient Space Utilization — Cache space is not given to apps that do not benefit
- Near Optimal Miss Ratio — Overall cache miss ratio is minimized

Sharing the Cache with Multiple Apps

- Efficient Space Utilization — Cache space is not given to apps that do not benefit
- Near Optimal Miss Ratio — Overall cache miss ratio is minimized

Allocation	Efficient Utilization	Near Optimal Miss Ratio
Static	✗	✓
Pooled (no guidance)	✓	✗
Pooled (w/ guidance)	✓	✓

Introduction - Multi-Tenant Key-Value Stores

Background - Estimating the Miss Ratio Curve

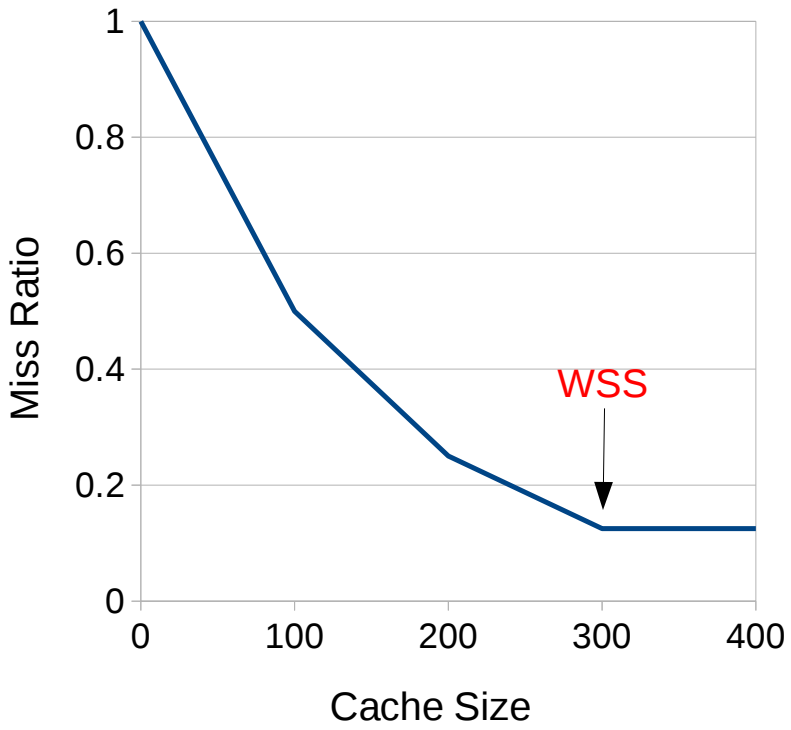
mPart: Miss Ratio Curve Guided Partitioning

Evaluation

Conclusion

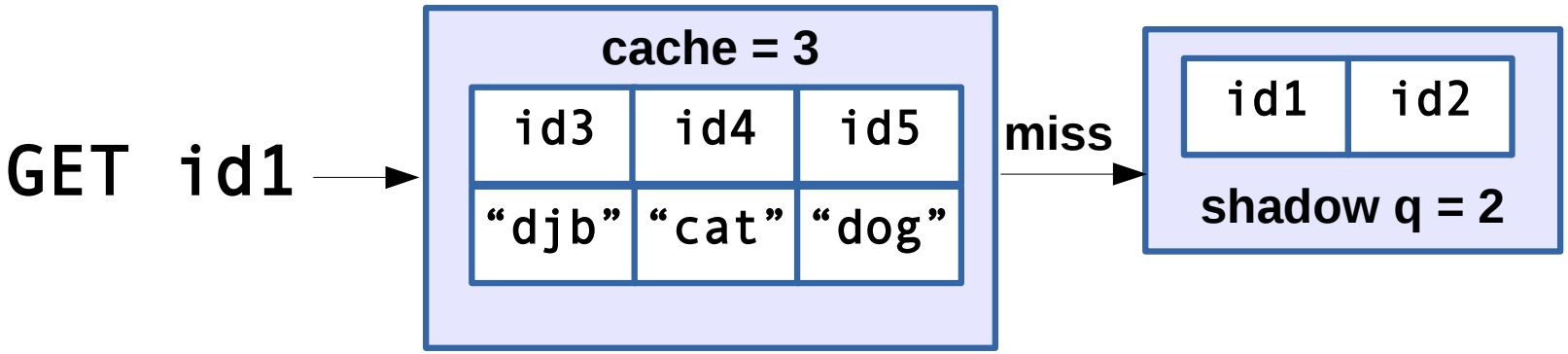
Miss Ratio Curves

- Miss Ratio Curve (MRC) — Quantifies the utility of increasing/decreasing cache size in terms of miss ratio
- Working Set Size (WSS) — The point that increasing cache size is not significant



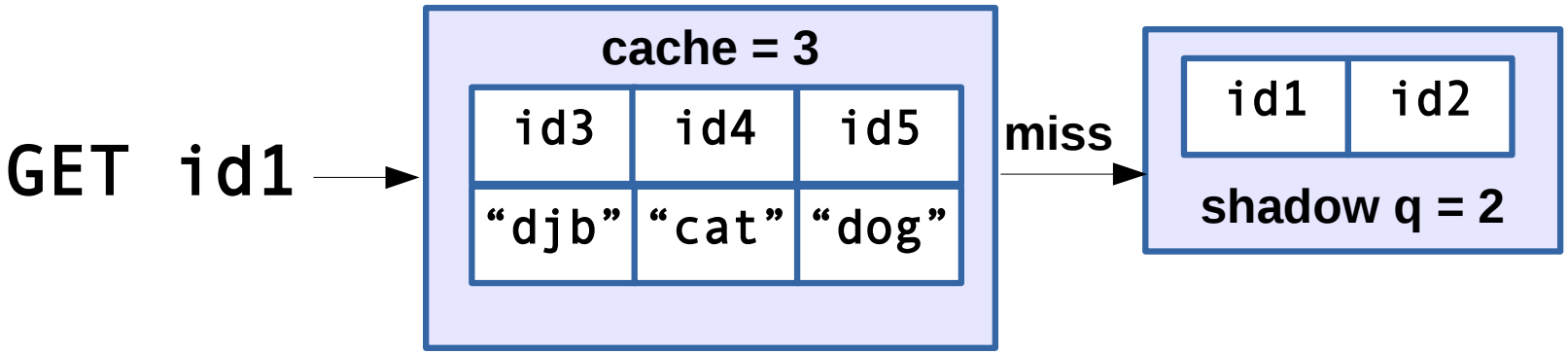
Memshare - Estimate the MRC Slope

- *Memshare* by Cidon et al. (USENIX ATC '17) use Shadow Queues
- Shadow Queues are logical victim caches, keep track of evicted keys



Memshare - Estimate the MRC Slope

- *Memshare* by Cidon et al. (USENIX ATC '17) use Shadow Queues
- Shadow Queues are logical victim caches, keep track of evicted keys



- When an app hits in the shadow queue, it is asssigned 1 credit
- 1 credit is taken from another app with lower shadow queue hit rate

Drawbacks of Shadow Queues

- Critical path overhead — must check on every miss
- Has limited scope — can only estimate gradient for `sizeof(shadow queue)` beyond the current allocation
- No way of quantifying decrease in cache size

Drawbacks of Shadow Queues

- Critical path overhead — must check on every miss
- Has limited scope — can only estimate gradient for `sizeof(shadow queue)` beyond the current allocation
- No way of quantifying decrease in cache size

Our approach: Use app's complete MRC to guide allocation

Introduction - Multi-Tenant Key-Value Stores

Background - Estimating the Miss Ratio Curve

mPart: Miss Ratio Curve Guided Partitioning

Evaluation

Conclusion

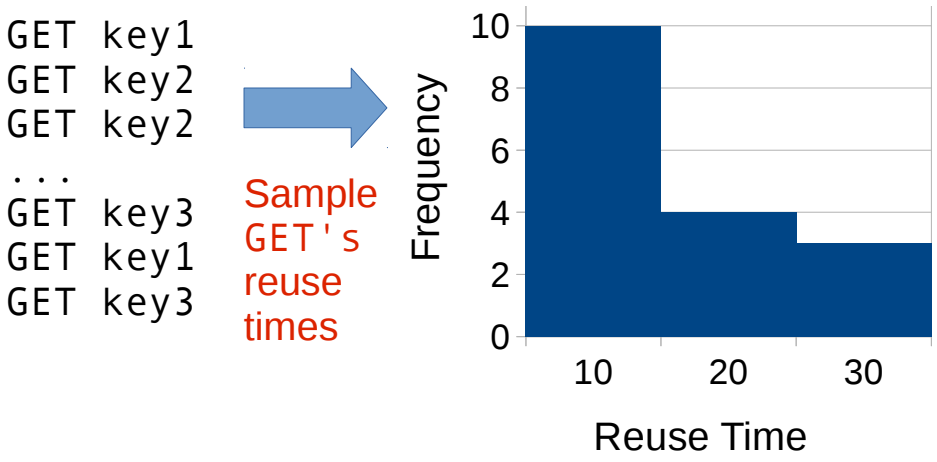
Building Miss Ratio Curve

- Track *reuse time* — Logical time between a request's *last use* and *current use*
- *Average Eviction Time* (Hu et al. USENIX ATC '16) — constructs accurate MRC by sampling reuse times

```
GET key1
GET key2
GET key2
...
GET key3
GET key1
GET key3
```

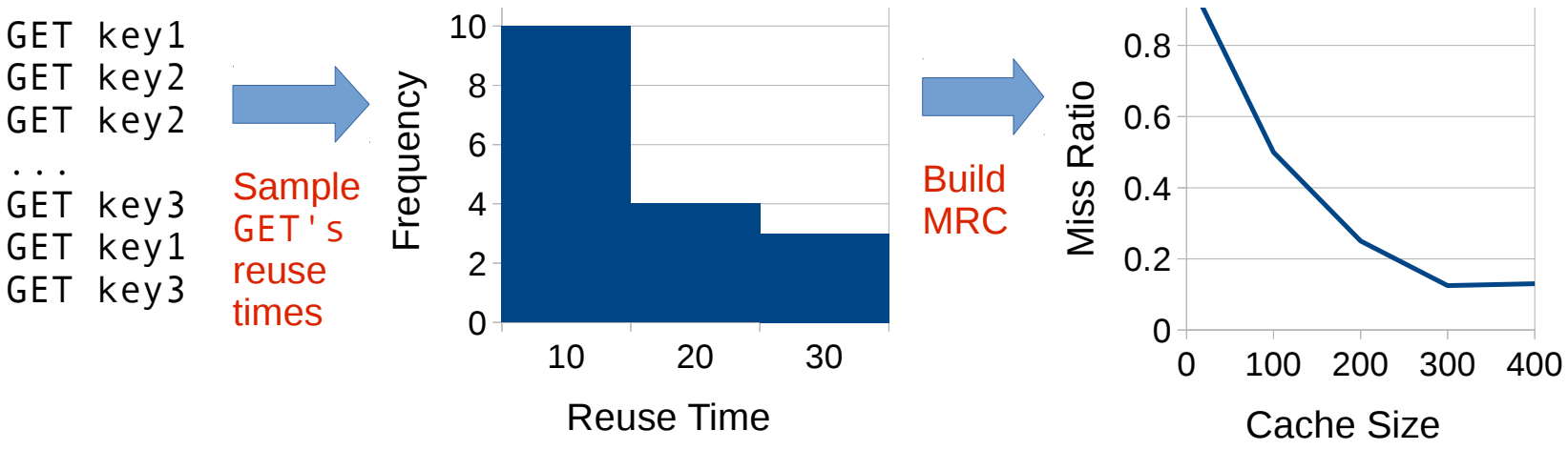
Building Miss Ratio Curve

- Track *reuse time* — Logical time between a request's *last use* and *current use*
- *Average Eviction Time* (Hu et al. USENIX ATC '16) — constructs accurate MRC by sampling reuse times



Building Miss Ratio Curve

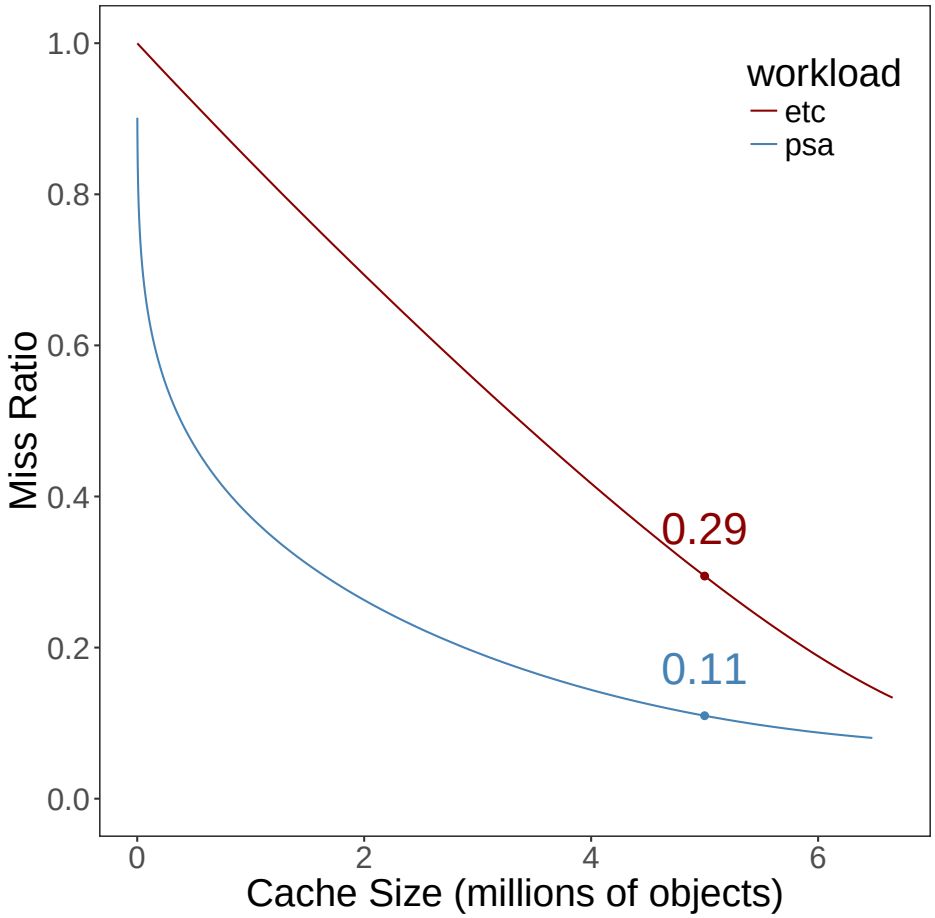
- Track *reuse time* — Logical time between a request's *last use* and *current use*
- *Average Eviction Time* (Hu et al. USENIX ATC '16) — constructs accurate MRC by sampling reuse times



- See paper for more details on how build MRC

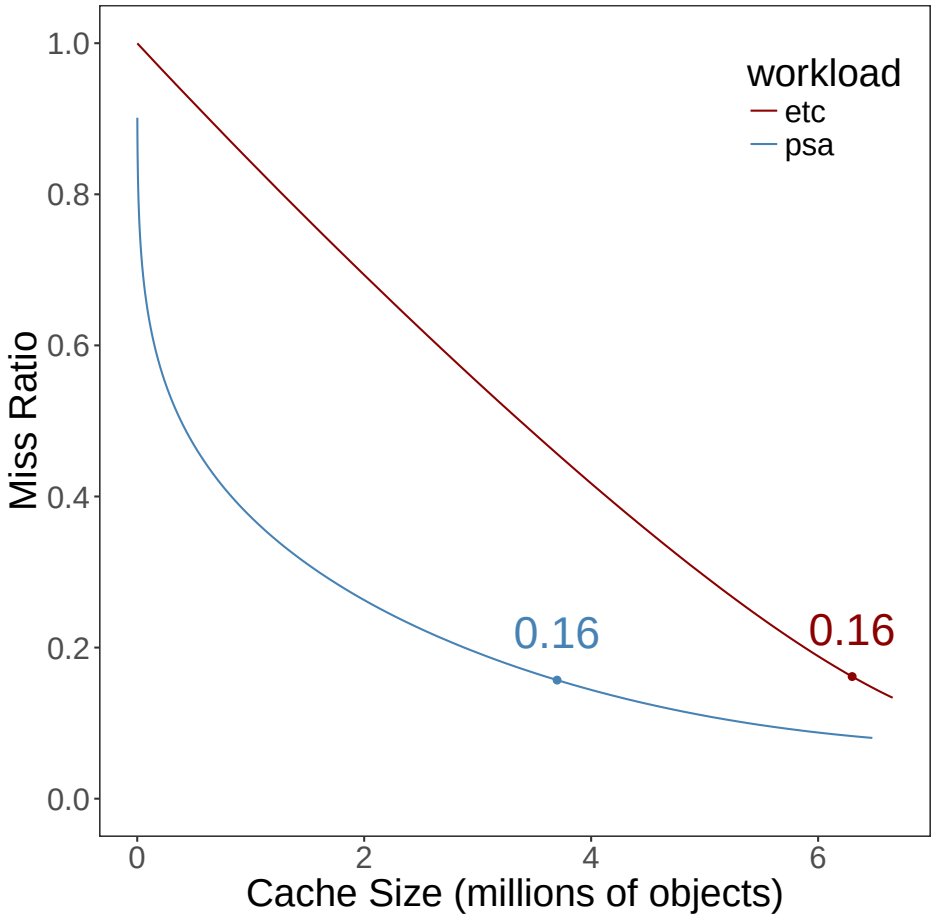
Example Allocation Using MRCs

- Cache size of 10 million objects
- 20% overall miss ratio



Example Allocation Using MRCs

- Cache size of 10 million objects
- 20% overall miss ratio → 16% overall miss ratio



Minimizing the Number of Misses

$\mathbf{m} = [m_1, m_2, ..., m_N]$ is a memory assignment to N apps

Minimizing the Number of Misses

$\mathbf{m} = [m_1, m_2, ..., m_N]$ is a memory assignment to N apps

$$\underset{F(\mathbf{m})}{\text{minimize}} \quad F(\mathbf{m}) = \sum_{i=1}^{N} NR_i * \underbrace{mrc_i(m_i)}_{\text{miss ratio for memory } m_i} \tag{1}$$

Minimizing the Number of Misses

$\mathbf{m} = [m_1, m_2, ..., m_N]$ is a memory assignment to N apps

$$\underset{F(\mathbf{m})}{\text{minimize}} \quad F(\mathbf{m}) = \sum_{i=1}^{N} \underbrace{NR_i * mrc_i(m_i)}_{\text{expected misses for app } i} \tag{1}$$

Minimizing the Number of Misses

$\mathbf{m} = [m_1, m_2, ..., m_N]$ is a memory assignment to N apps

$$\underset{F(\mathbf{m})}{\text{minimize}} \quad F(\mathbf{m}) = \underbrace{\sum_{i=1}^{N} NR_i * mrc_i(m_i)}_{\text{total misses}} \tag{1}$$

Minimizing the Number of Misses

$\mathbf{m} = [m_1, m_2, ..., m_N]$ is a memory assignment to N apps

$$\begin{aligned} \underset{F(\mathbf{m})}{\text{minimize}} \quad & F(\mathbf{m}) = \sum_{i=1}^{N} NR_i * mrc_i(m_i) \\ \text{subject to} \quad & \underbrace{\sum_{i=1}^{N} m_i \leq M}_{\text{total memory } M} \end{aligned} \tag{1}$$

Minimizing the Number of Misses

$\mathbf{m} = [m_1, m_2, ..., m_N]$ is a memory assignment to N apps

$$\begin{aligned} \underset{F(\mathbf{m})}{\text{minimize}} \quad & F(\mathbf{m}) = \sum_{i=1}^{N} NR_i * mrc_i(m_i) \\ \text{subject to} \quad & \underbrace{\sum_{i=1}^{N} m_i \leq M}_{\text{total memory } M} \end{aligned} \tag{1}$$

Problem: Brute force search is too costly

Minimizing the Number of Misses

$\mathbf{m} = [m_1, m_2, ..., m_N]$ is a memory assignment to N apps

$$\begin{aligned} \underset{F(\mathbf{m})}{\text{minimize}} \quad & F(\mathbf{m}) = \sum_{i=1}^{N} NR_i * mrc_i(m_i) \\ \text{subject to} \quad & \underbrace{\sum_{i=1}^{N} m_i \leq M}_{\text{total memory } M} \end{aligned} \tag{1}$$

Problem: Brute force search is too costly

Solution: Use Dynamic Programming for polynomial time search

Implementation

- Implemented AET and allocation algorithm in C++ on top of Memshare
  - Special thanks to Cidon, Rushton, and Stutsman for source code
- Sampling for AET is done on incoming `GET` requests
- Constructing MRC occurs in separate thread

Introduction - Multi-Tenant Key-Value Stores

Background - Estimating the Miss Ratio Curve

mPart: Miss Ratio Curve Guided Partitioning

Evaluation

Conclusion

System Parameters

Memshare

- Increment/decrement 500 objects on a hit in the shadow queue
- Choose the application with lowest shadow queue hit rate to remove from
- Shadow queue represents 10MB of objects

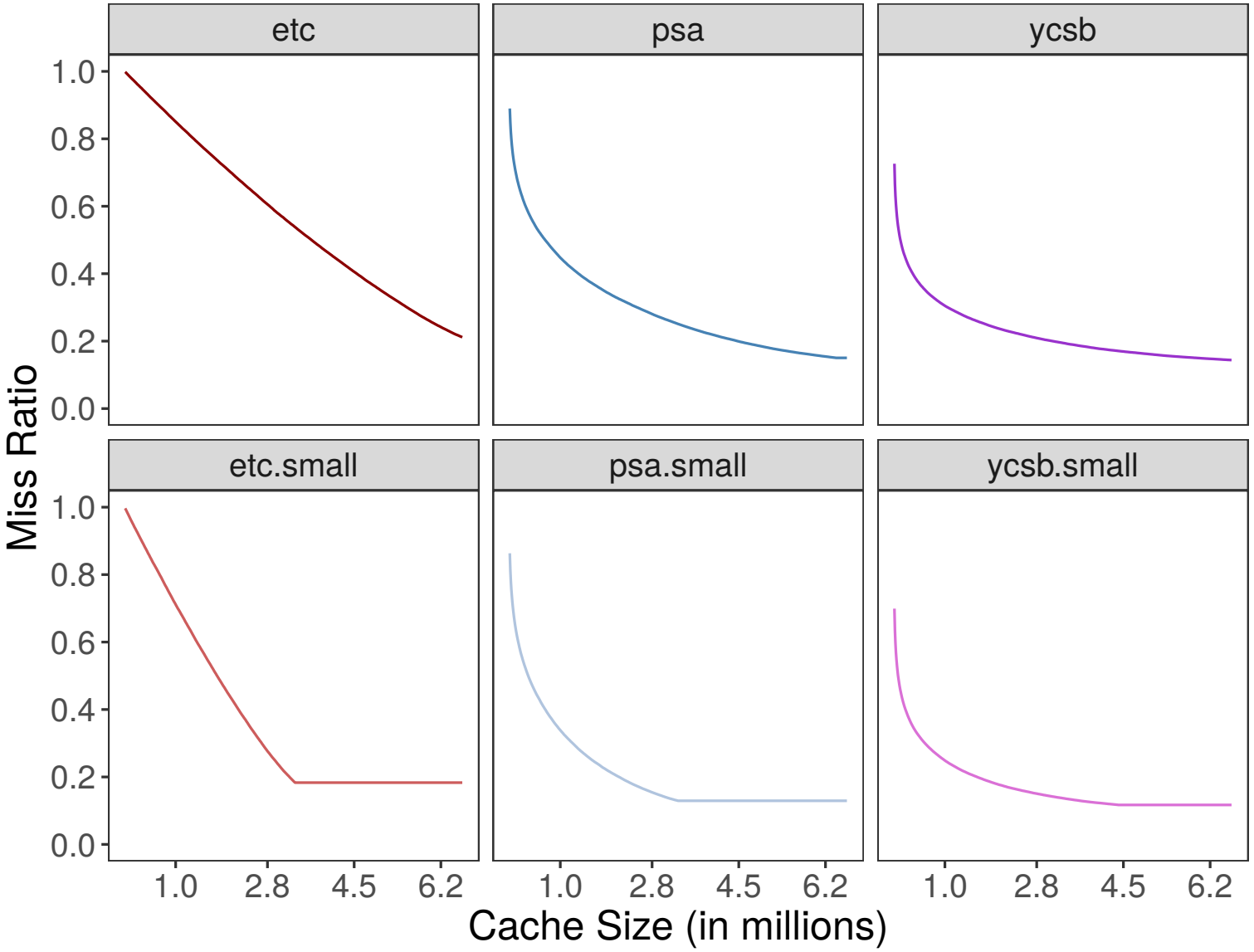
mPart

- Re-allocate every 2 million requests
- Sample rate of 1/10,000 `GET` requests

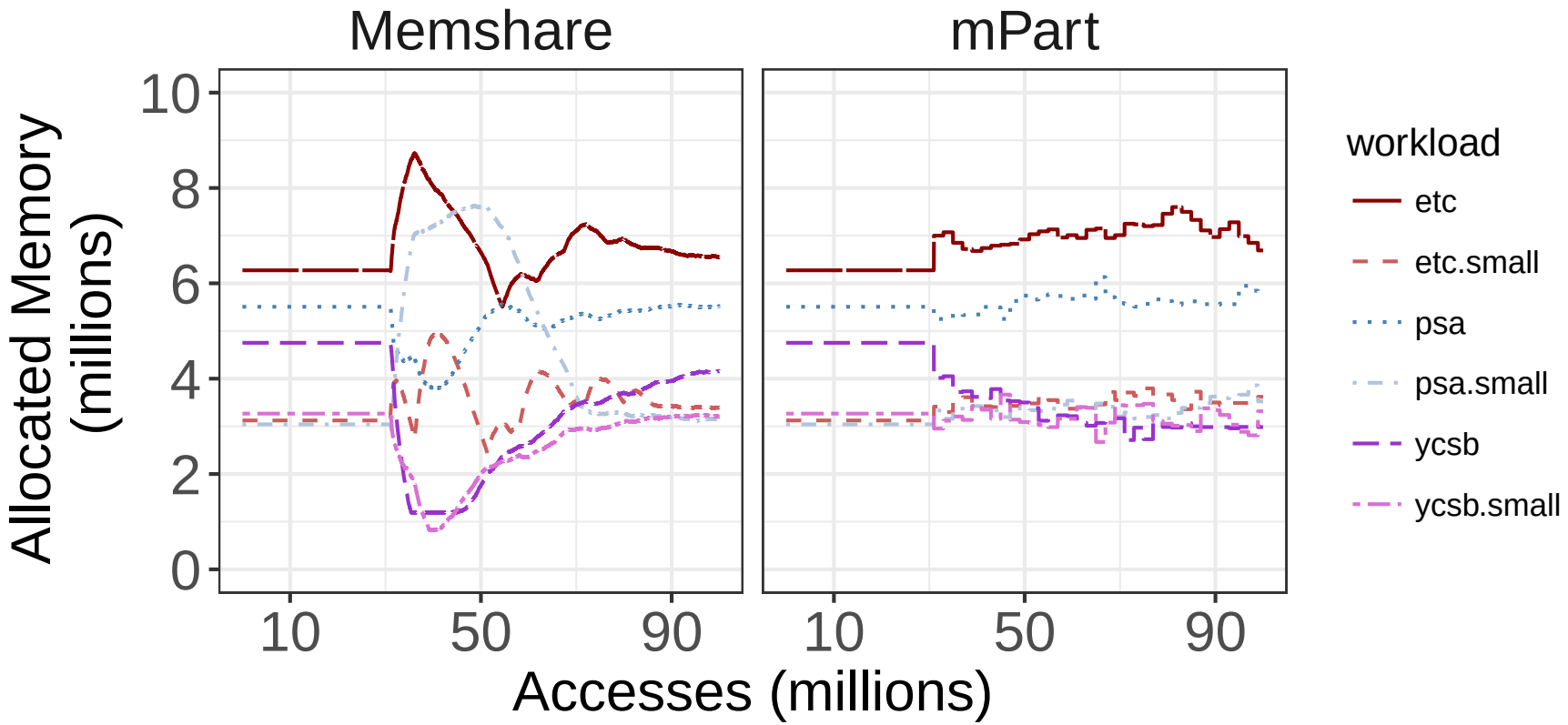
Workloads Used

- *etc* and *etc.small* — Modeled after Facebook workload
- *psa* and *psa.small* — Used in prior key-value store works
- *ycsb* and *ycsb.small* — Yahoo Cloud Suite Benchmark workload C
- 100 million requests each
- Have distinct working set sizes

Workload MRCs



Arrive at Stable Allocation Faster



Decrease in Miss Ratio

Over 3 different initial allocation settings

- WSS Based — Set to 90% of WSS
- Objects — Set to 75% of unique objects
- Equal — Each application gets the same number of objects

Decrease in Miss Ratio

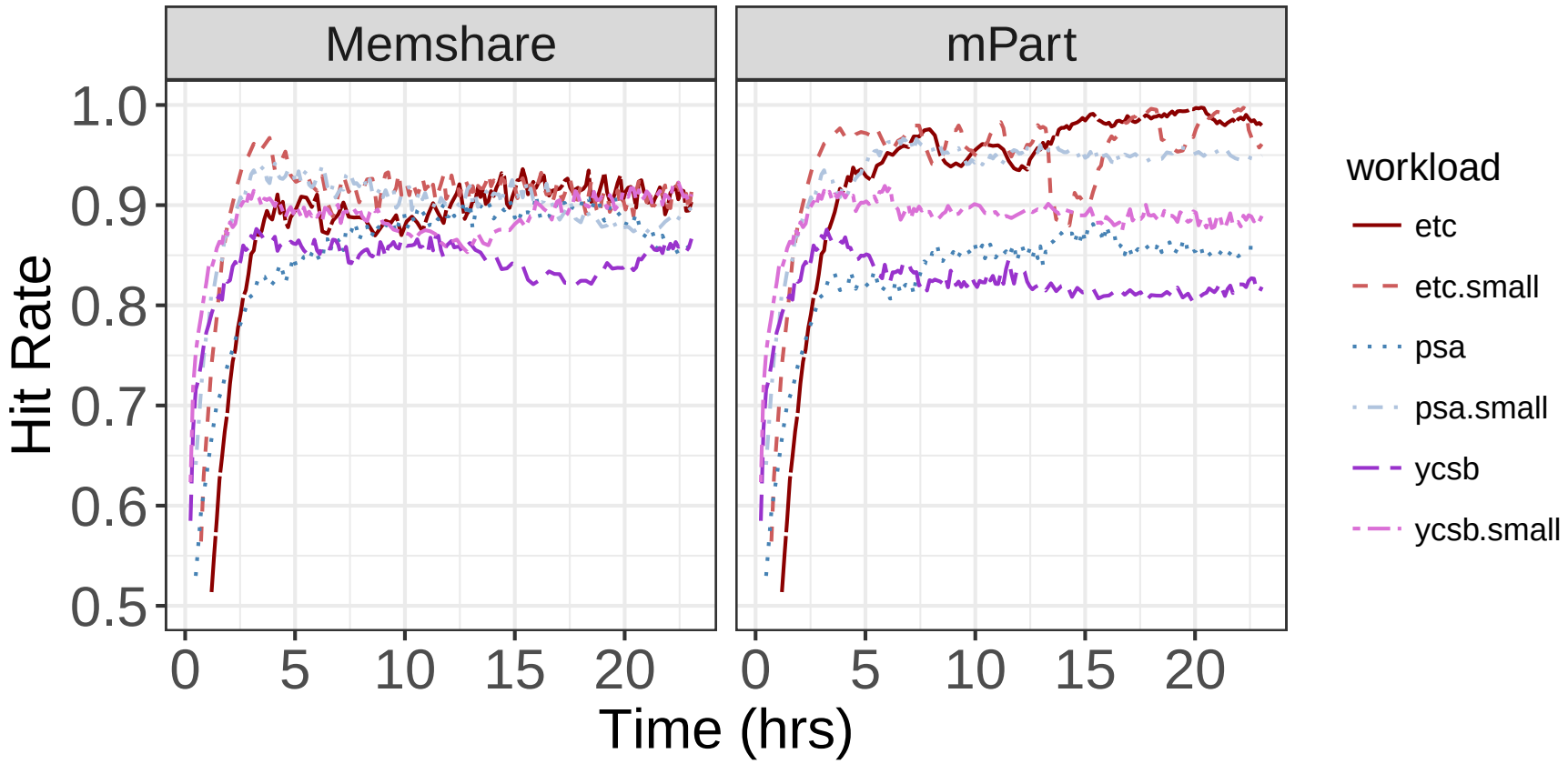
Over 3 different initial allocation settings

- WSS Based — Set to 90% of WSS
- Objects — Set to 75% of unique objects
- Equal — Each application gets the same number of objects

	Initial Setting		
Miss Ratio	**WSS**	**Objects**	**Equal**
Memshare	4.8%	4.2%	4.9%
mPart	3.6%	3.2%	3.9%
Reduction	-25.0%	-23.0%	-20.4%

Time-Varying Workloads - Hit Ratio

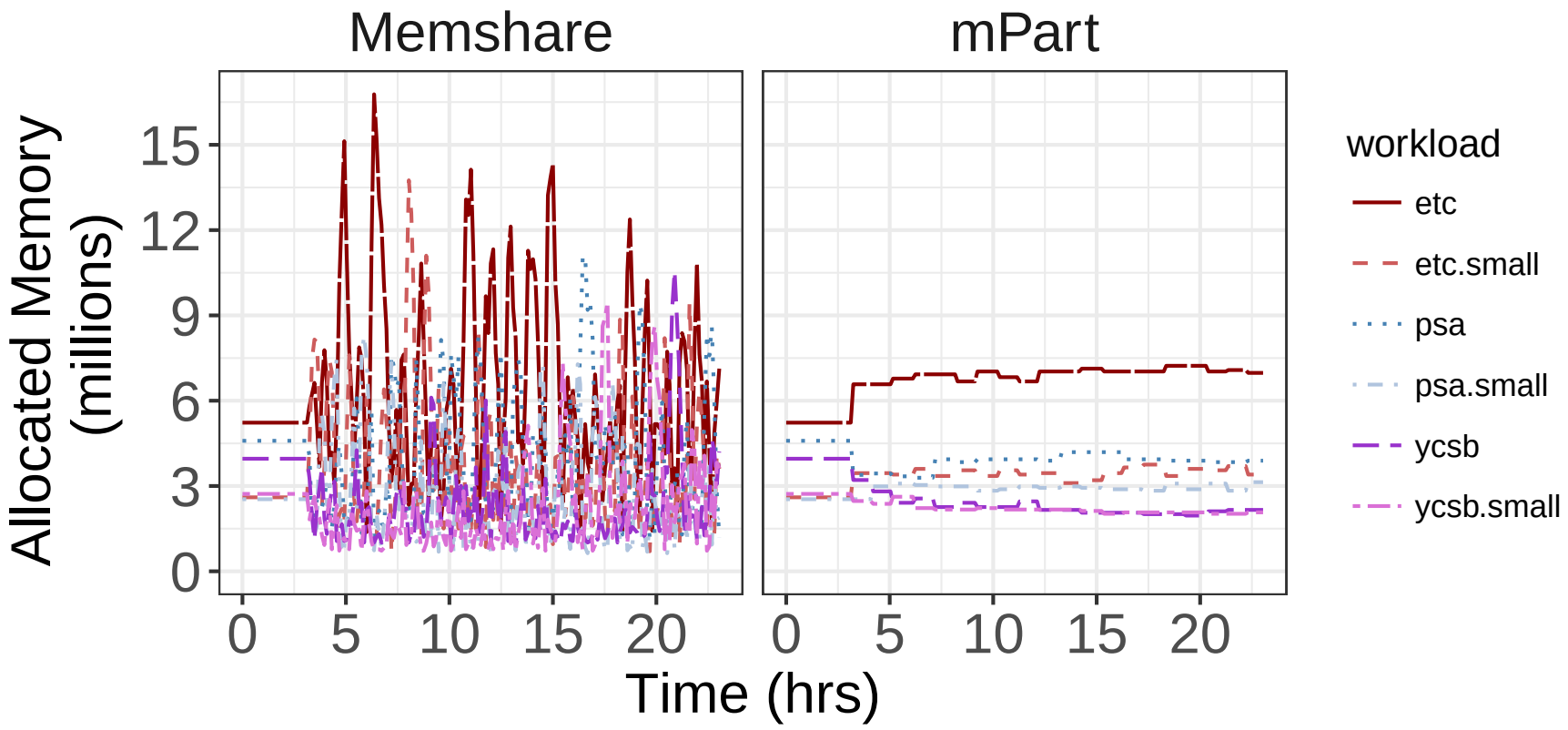
Set workloads to follow the inter-arrival rate distribution seen at Facebook over 24 hour period (Atikoglu, SIGMETRICS '12)



Overall Miss Ratio: Memshare: 11% → mPart: 8.9%

Time-Varying Workloads - Allocation

Set workloads to follow the inter-arrival rate distribution seen at Facebook over 24 hour period (Atikoglu, SIGMETRICS '12)



Cache Latency Improvement - GET

`GET` request latencies measured in µs

- `GET` may be a miss, cause a check in shadow queue

System	95%	99%	99.9%
Memshare	38.27	45.13	55.41
mPart	37.42	44.48	53.81
Improvement	2.22%	1.44%	2.89%

Cache Latency Improvement - SET

`SET` request latencies µs

- `SET` may cause eviction, need to update shadow queue in Memshare

System	95%	99%	99.9%
Memshare	48.69	85.99	126.69
mPart	47.98	82.92	123.77
Improvement	1.46%	3.57%	2.30%

Cache Throughput Improvement

Throughput	**ycsb**	
Memshare	560.1	kop/s
mPart	610.8	kop/s
Improvement	8.8%	

Cache Throughput Improvement

Throughput	ycsb	
Memshare	560.1	kop/s
mPart	610.8	kop/s
Improvement	8.8%	

No need to check shadow queue; only sample incoming requests

Introduction - Multi-Tenant Key-Value Stores

Background - Estimating the Miss Ratio Curve

mPart: Miss Ratio Curve Guided Partitioning

Evaluation

Conclusion

Conclusion

Recent advancements in MRC construction allow them to be used in online allocation

- Improved overall hit ratio
- Increased cache throughput
- Lower tail-latency

Thank you, questions?

mPart: Miss Ratio Curve Guided Partitioning in Key-Value Stores

Daniel Byrne djbyrne@mtu.edu
Nilufer Onder nilufer@mtu.edu
Zhenlin Wang zlwang@mtu.edu

Department of Computer Science
Michigan Technological University

International Symposium on Memory Management
June 2018