Cliffhanger: Scaling Performance Cliffs in Web Memory Caches

(climbing)

NSDI'16

Memory Caches are Essential to Web-scale Application Performance

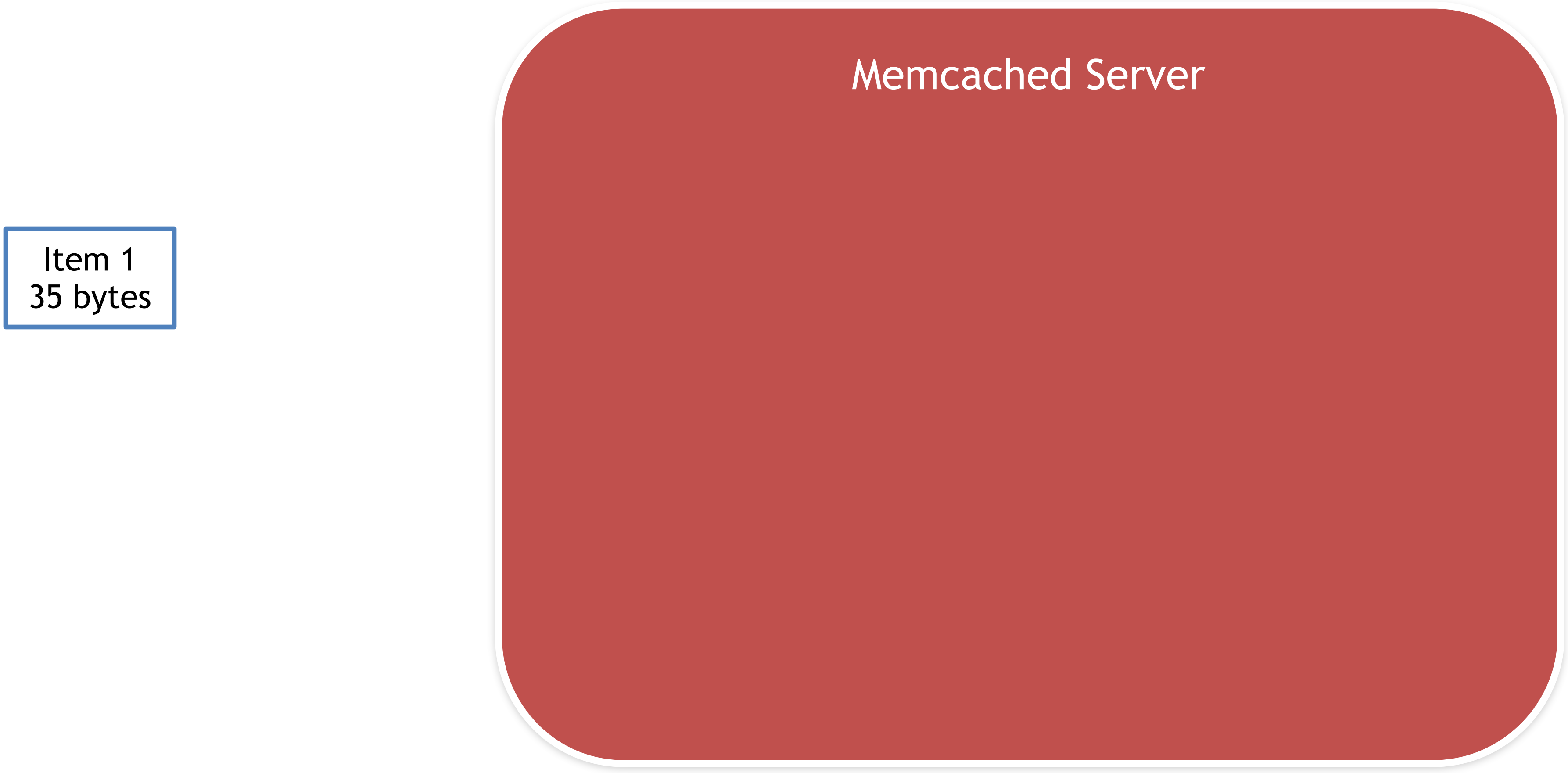
Memory Cache Hit Rate Drives Performance

- Memcached most widely used cache in large data centers
- Small improvements are important, especially when hit rates are high
- +1% cache hit-rate → 35% speedup
  - read latency from cache: 200μs, MySQL: 10ms
  - Old latency: 374 μs
  - New latency: 278 μs

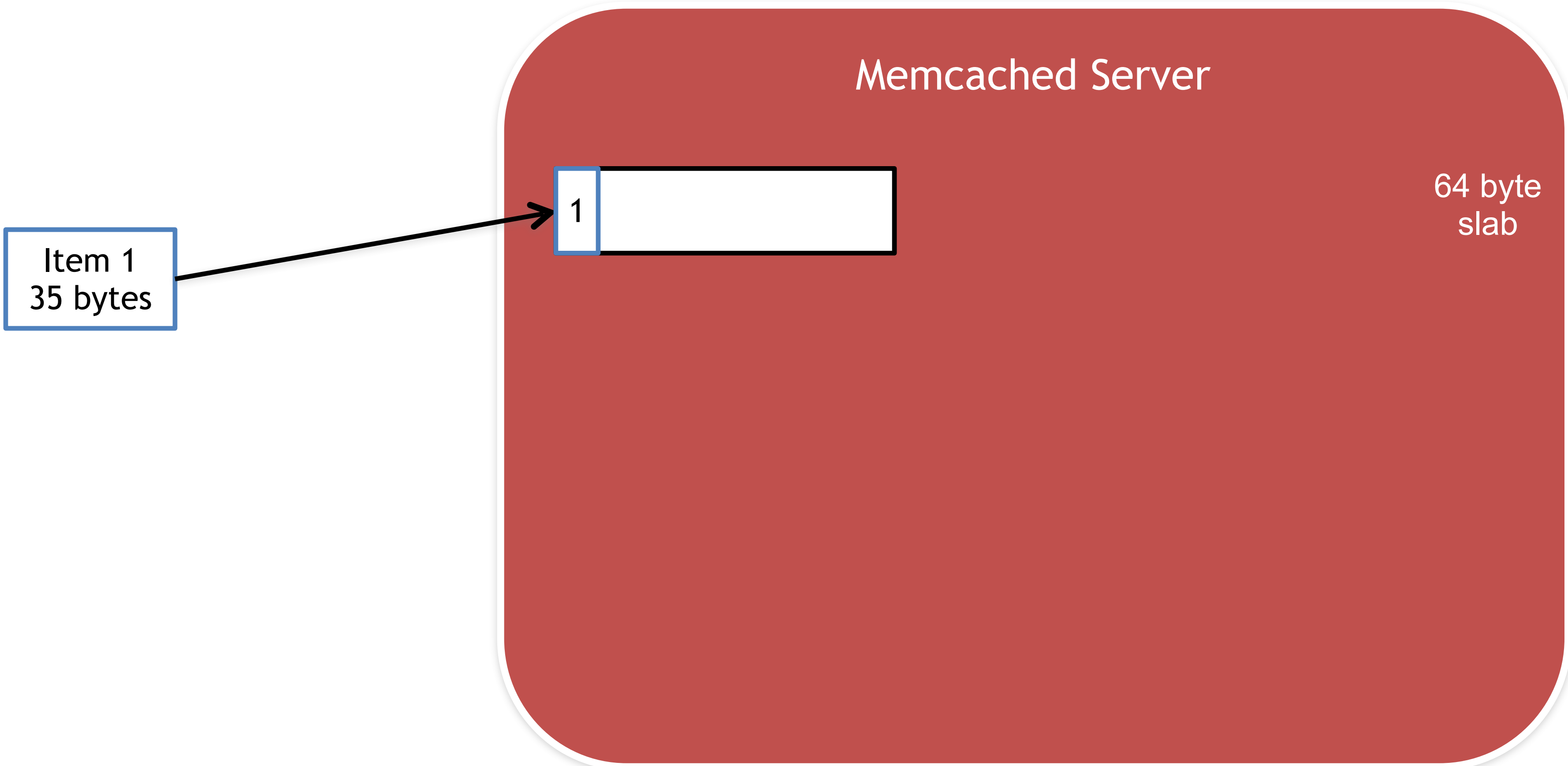
Memory Caches not Optimized for Maximizing Hit Rate

- Does not optimize for hit rate across different request sizes and applications
  - Cache greedily assigns memory to different request sizes and applications
  - Memory assignment remains static

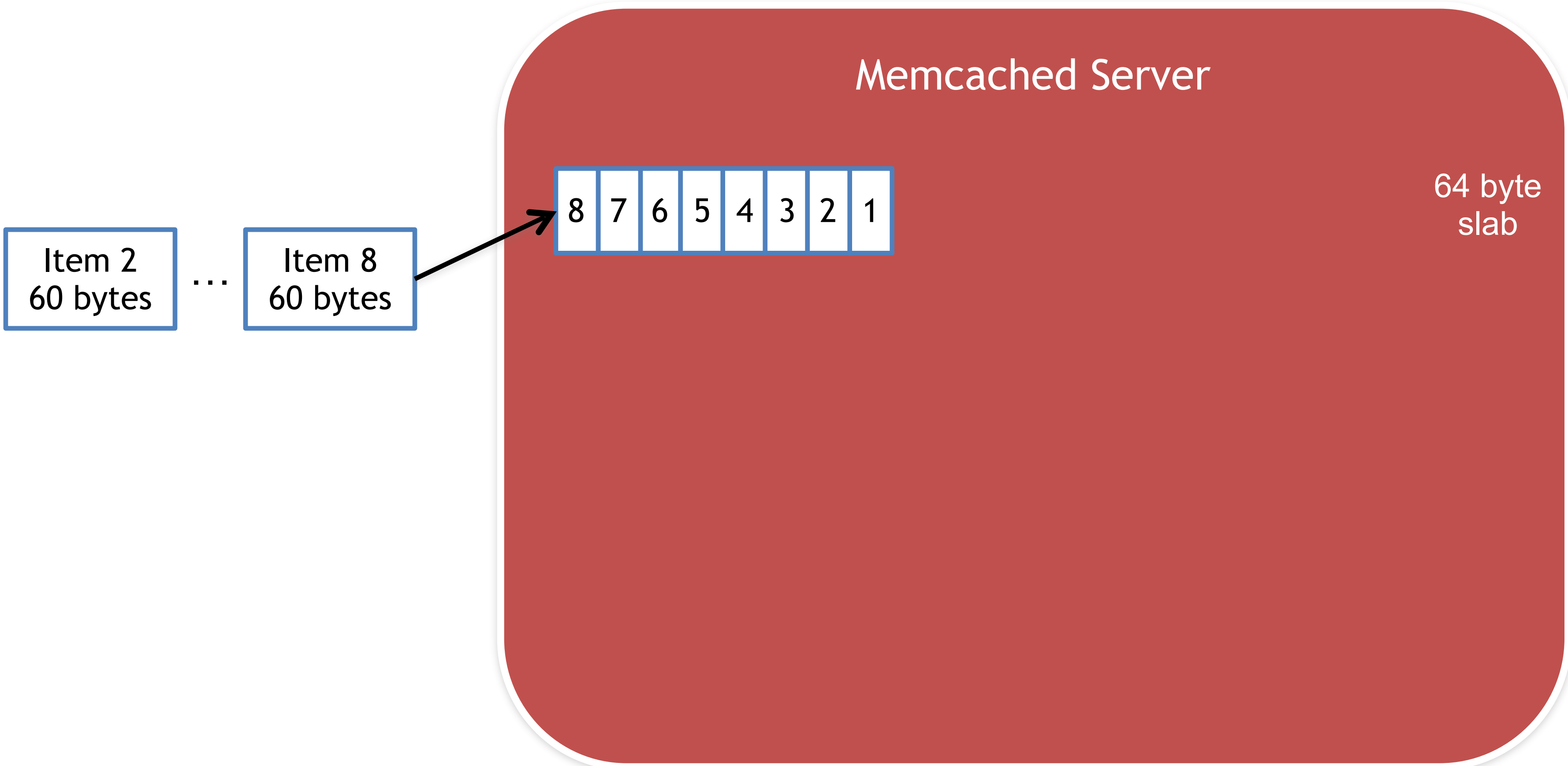
Memcached’s Static Cache Allocation



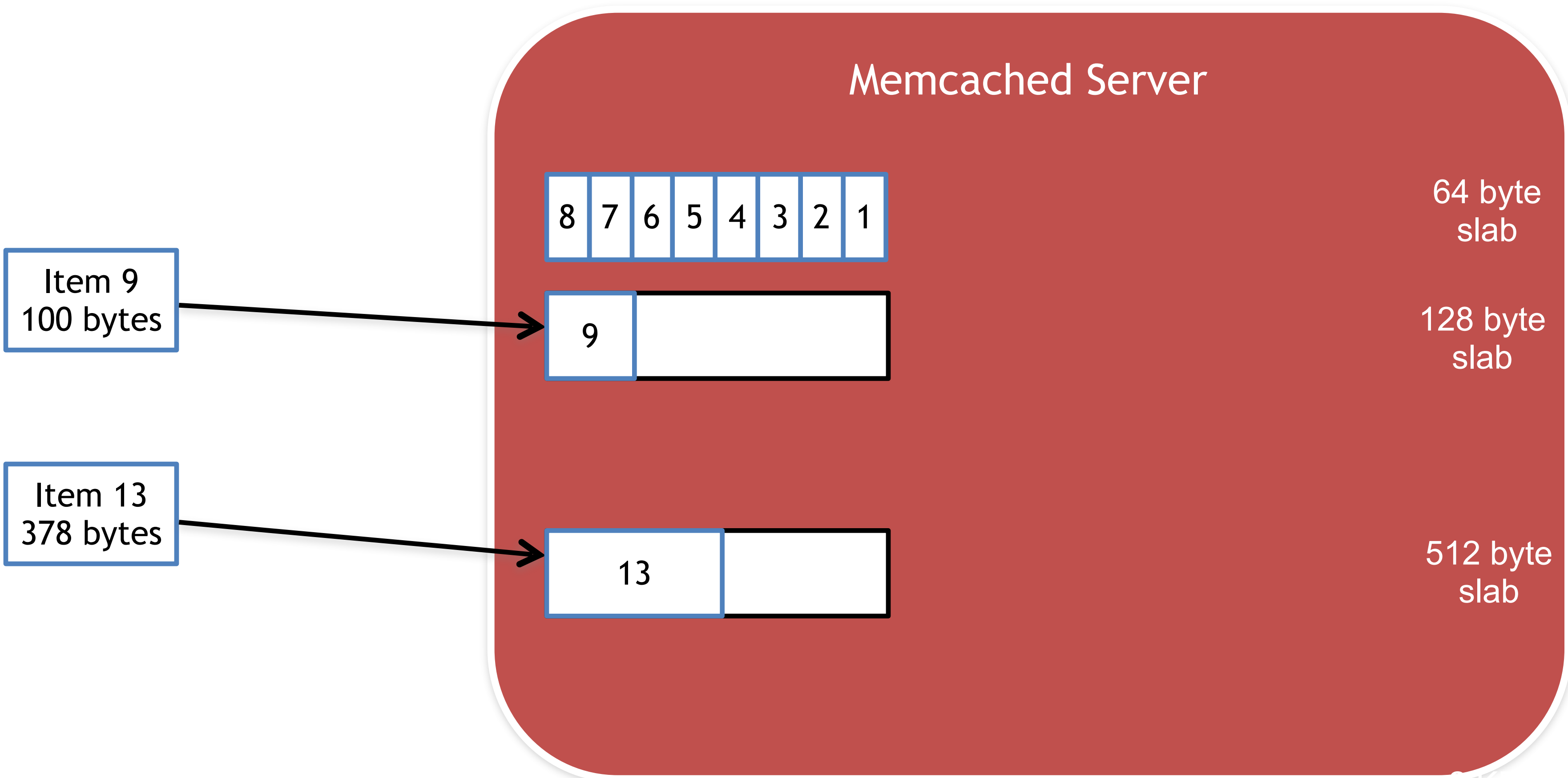
Memcached's Static Cache Allocation



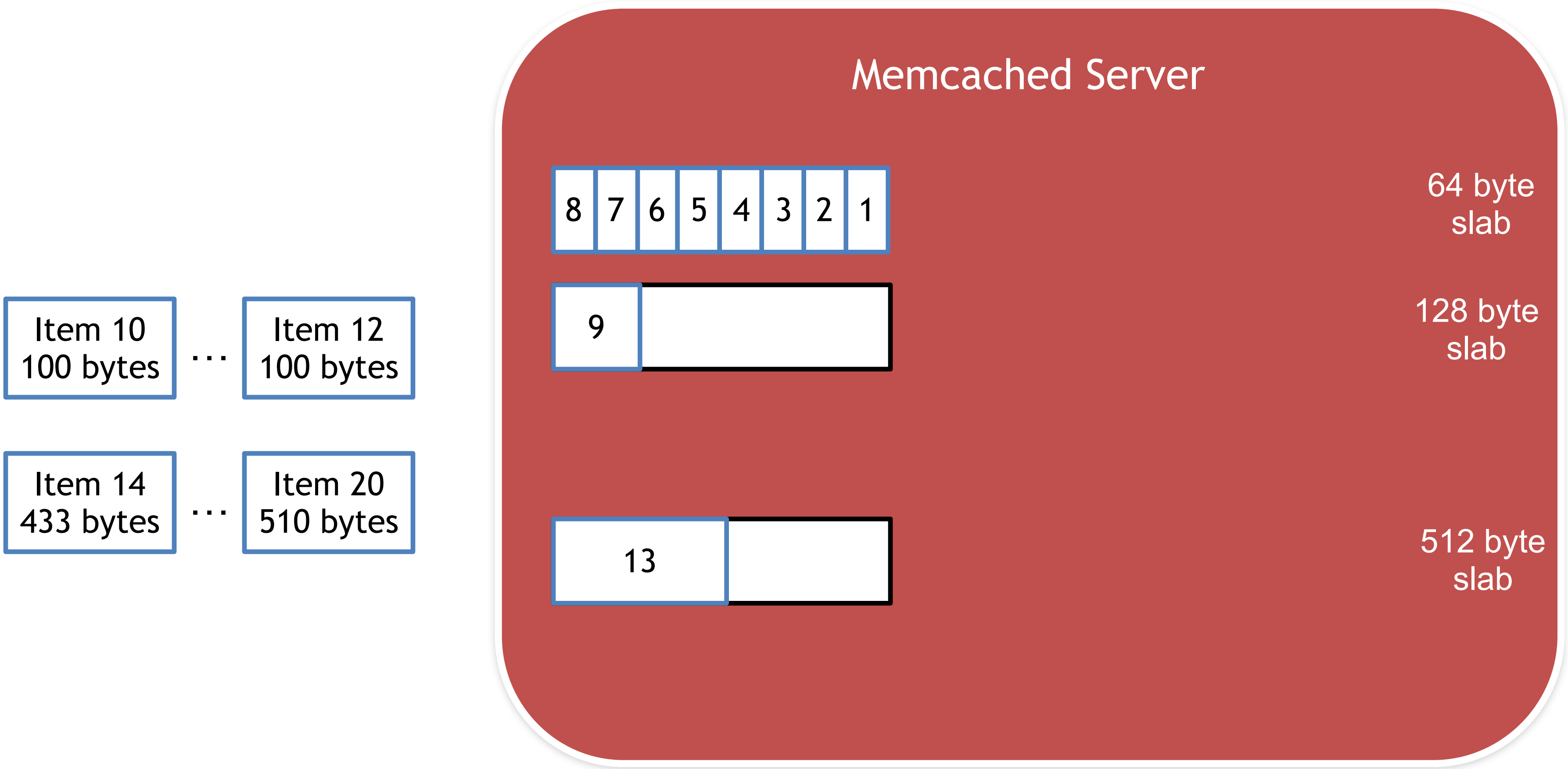
Memcached's Static Cache Allocation



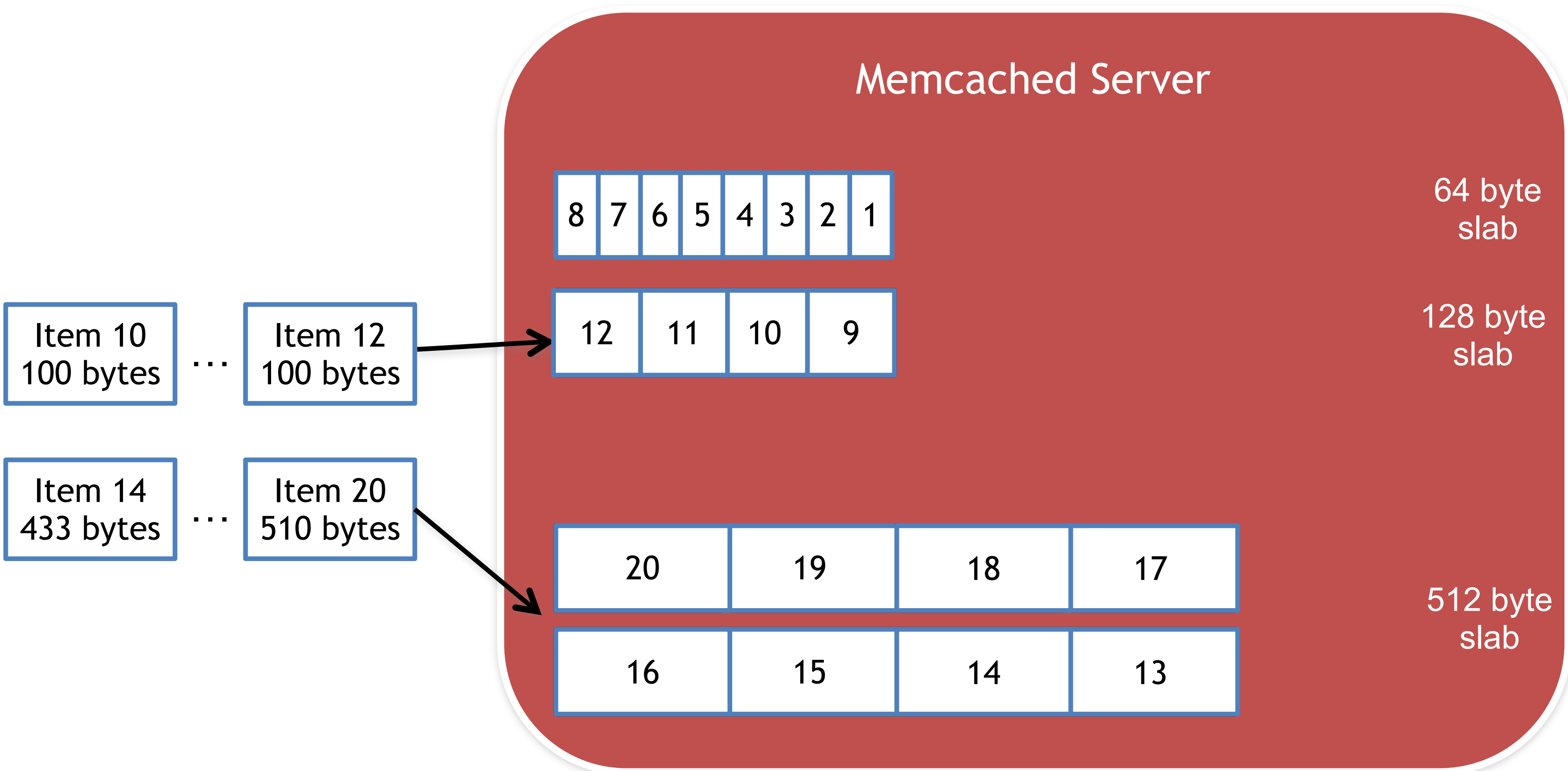
Memcached's Static Cache Allocation



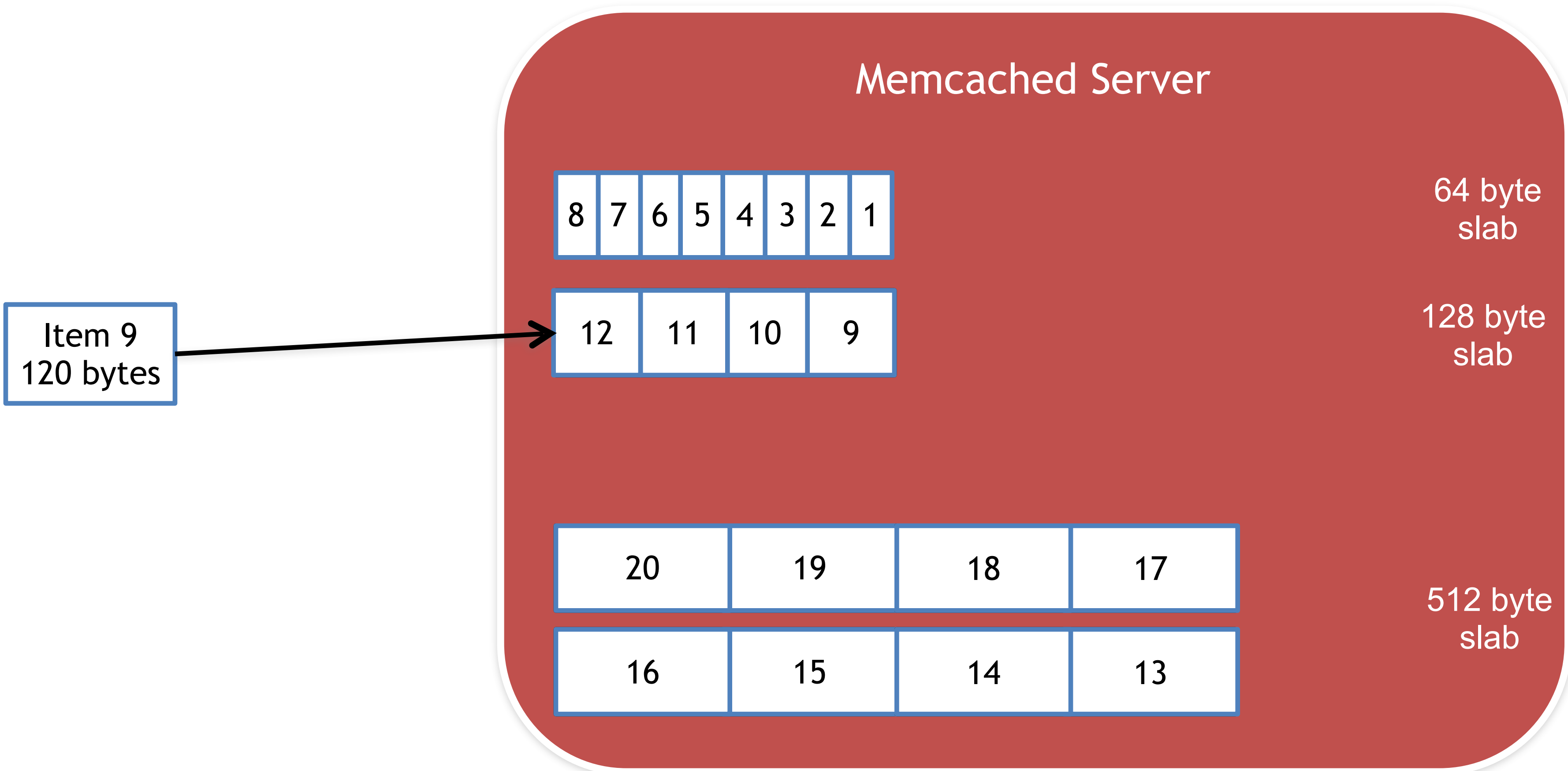
Memcached's Static Cache Allocation



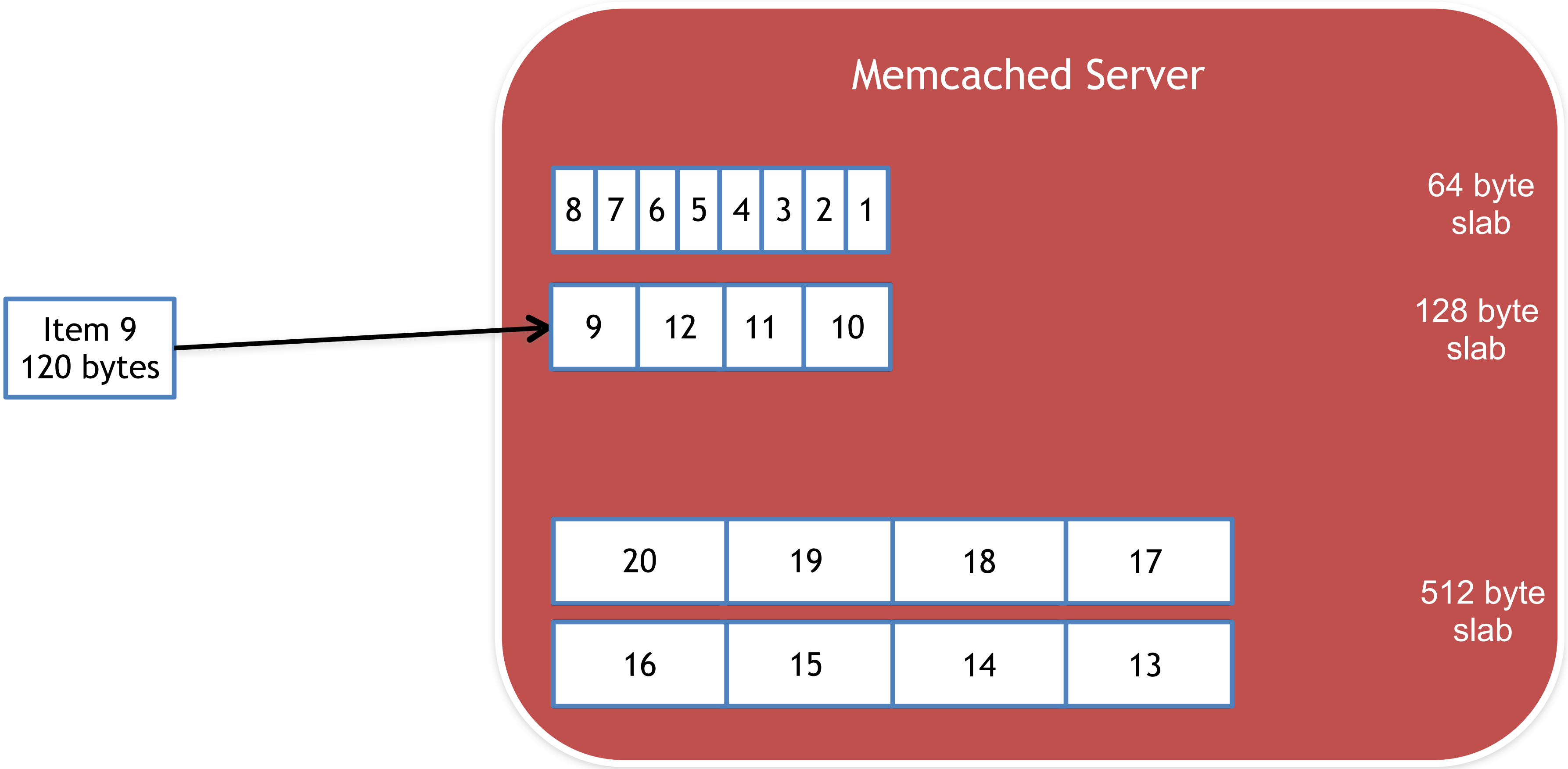
Memcached's Static Cache Allocation



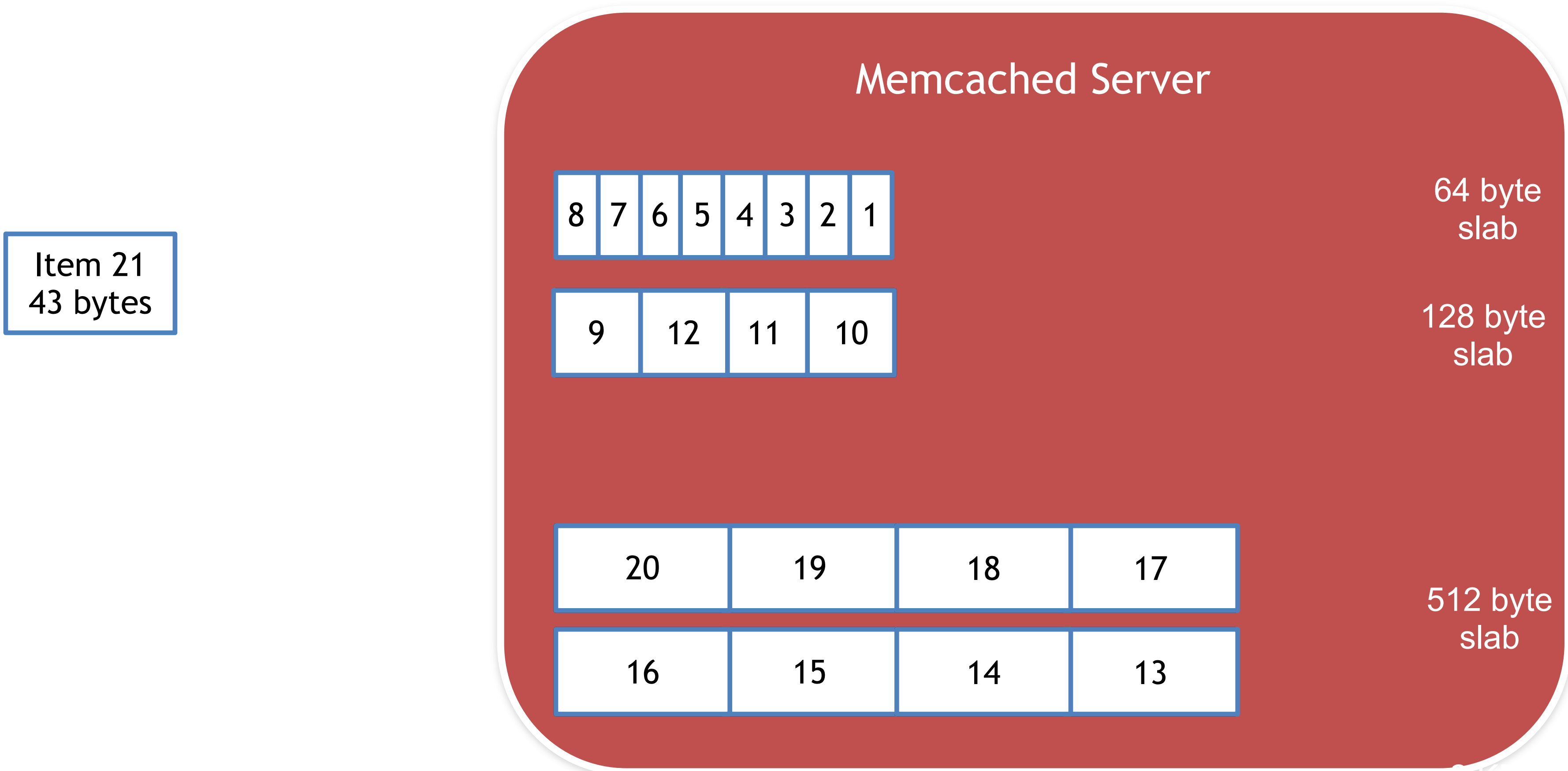
Memcached's Static Cache Allocation



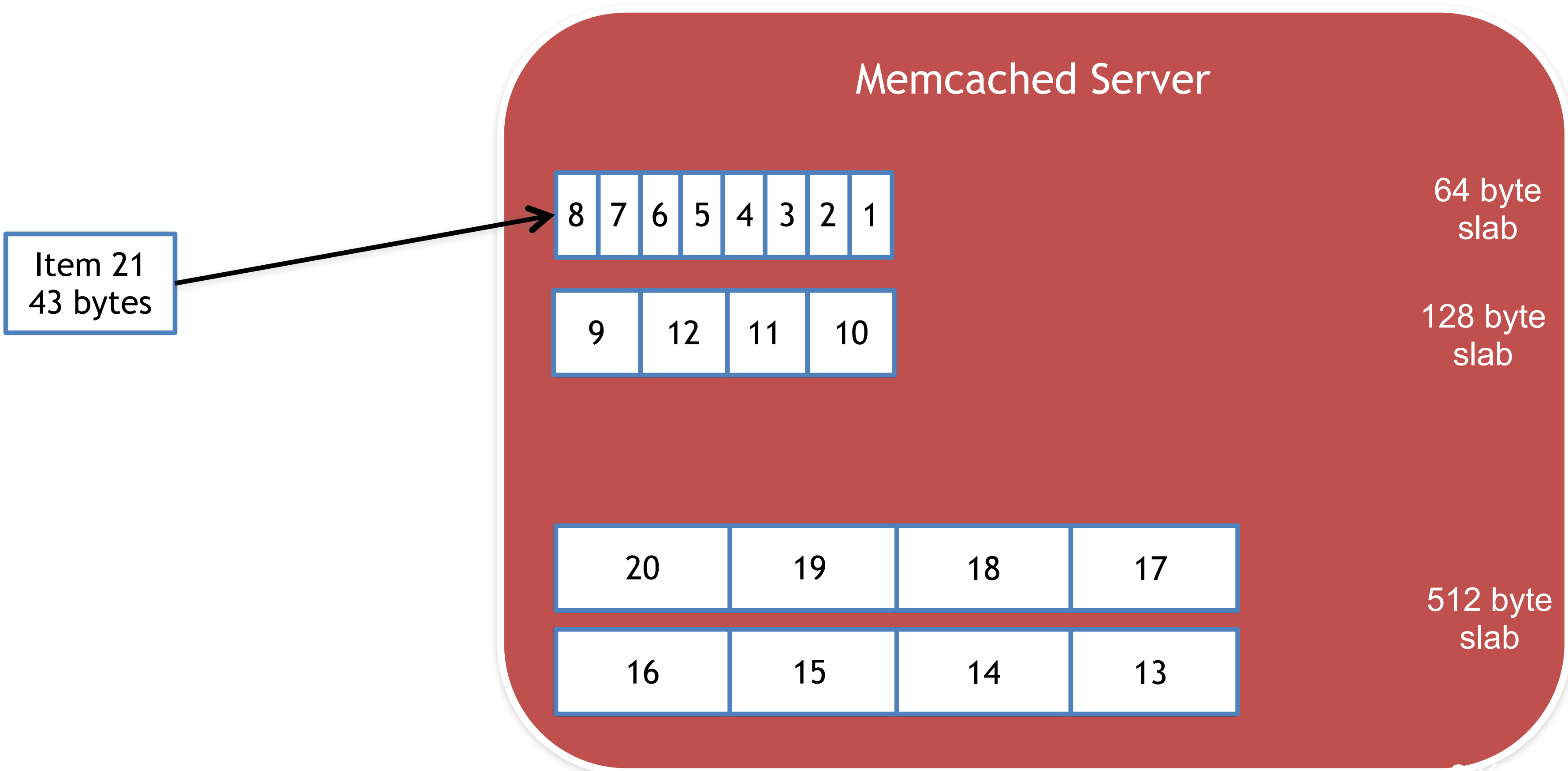
Memcached’s Static Cache Allocation



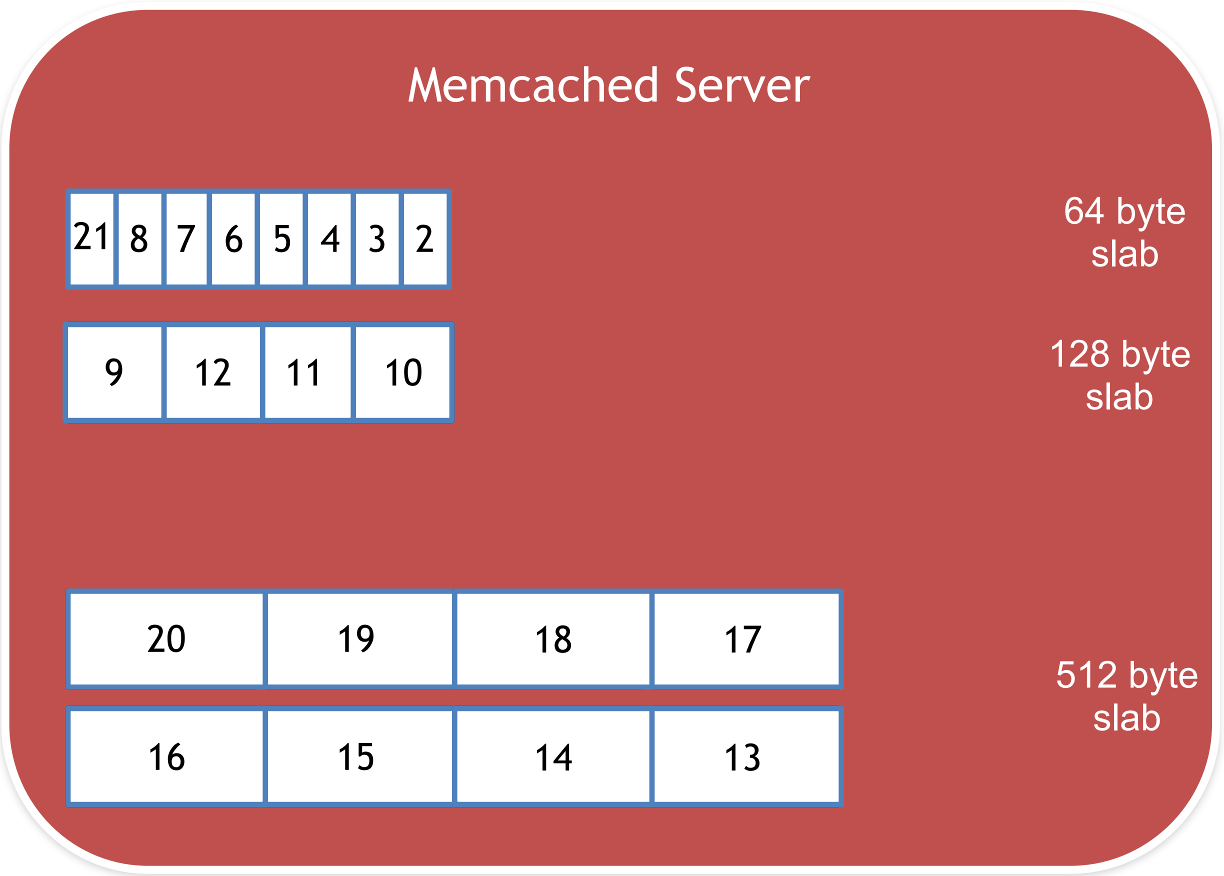
Memcached's Static Cache Allocation



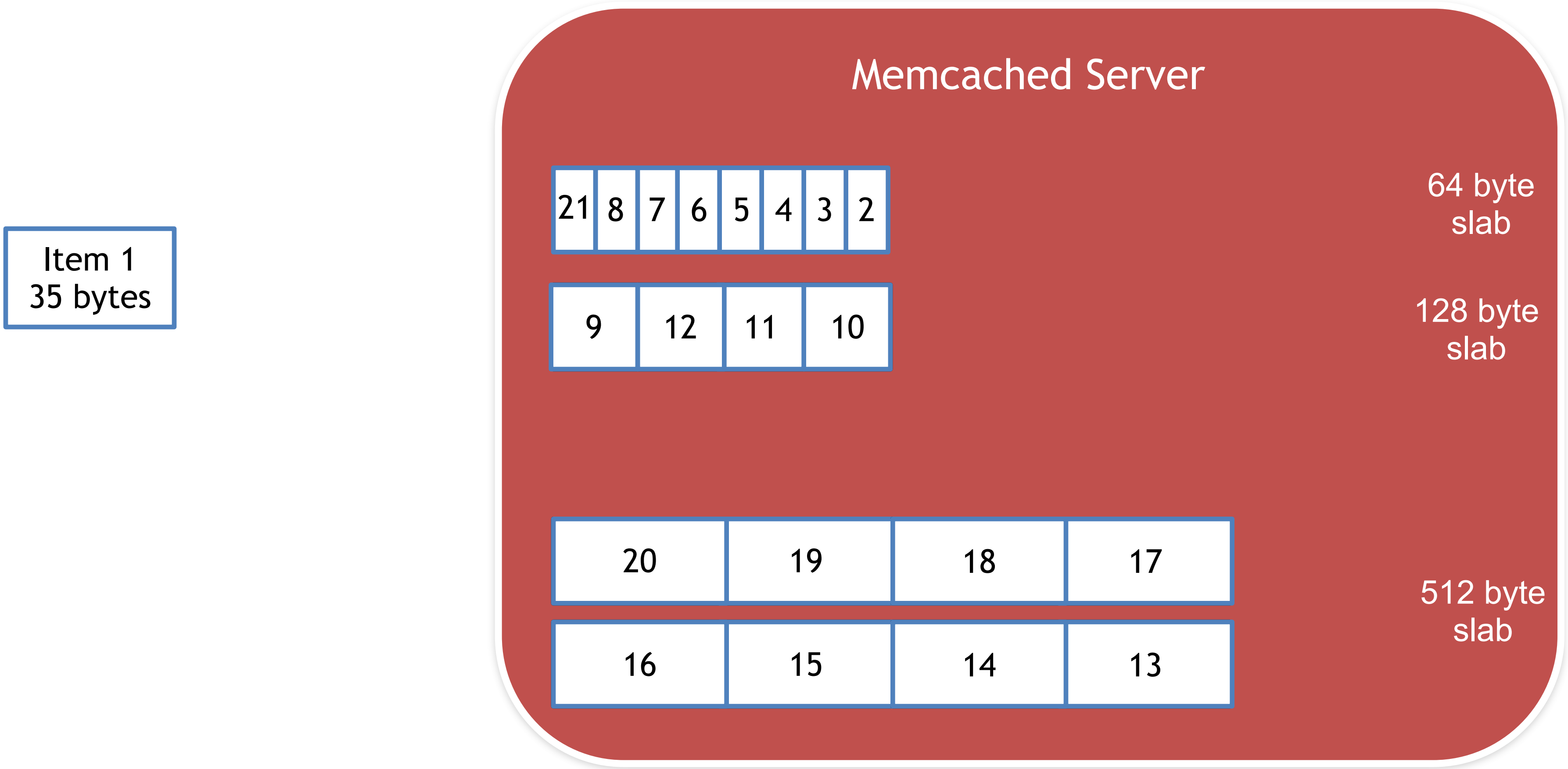
Memcached's Static Cache Allocation



Memcached's Static Cache Allocation



Memcached's Static Cache Allocation

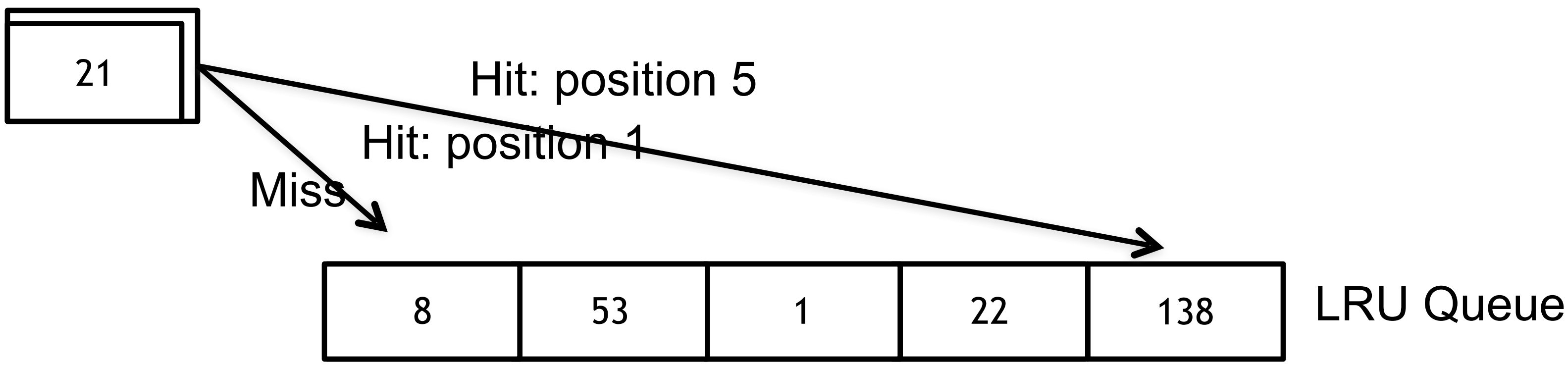


Problems with Memcached Static Cache Allocation

1. Greedy page allocation favors large slab classes
2. The distribution of request sizes changes over time

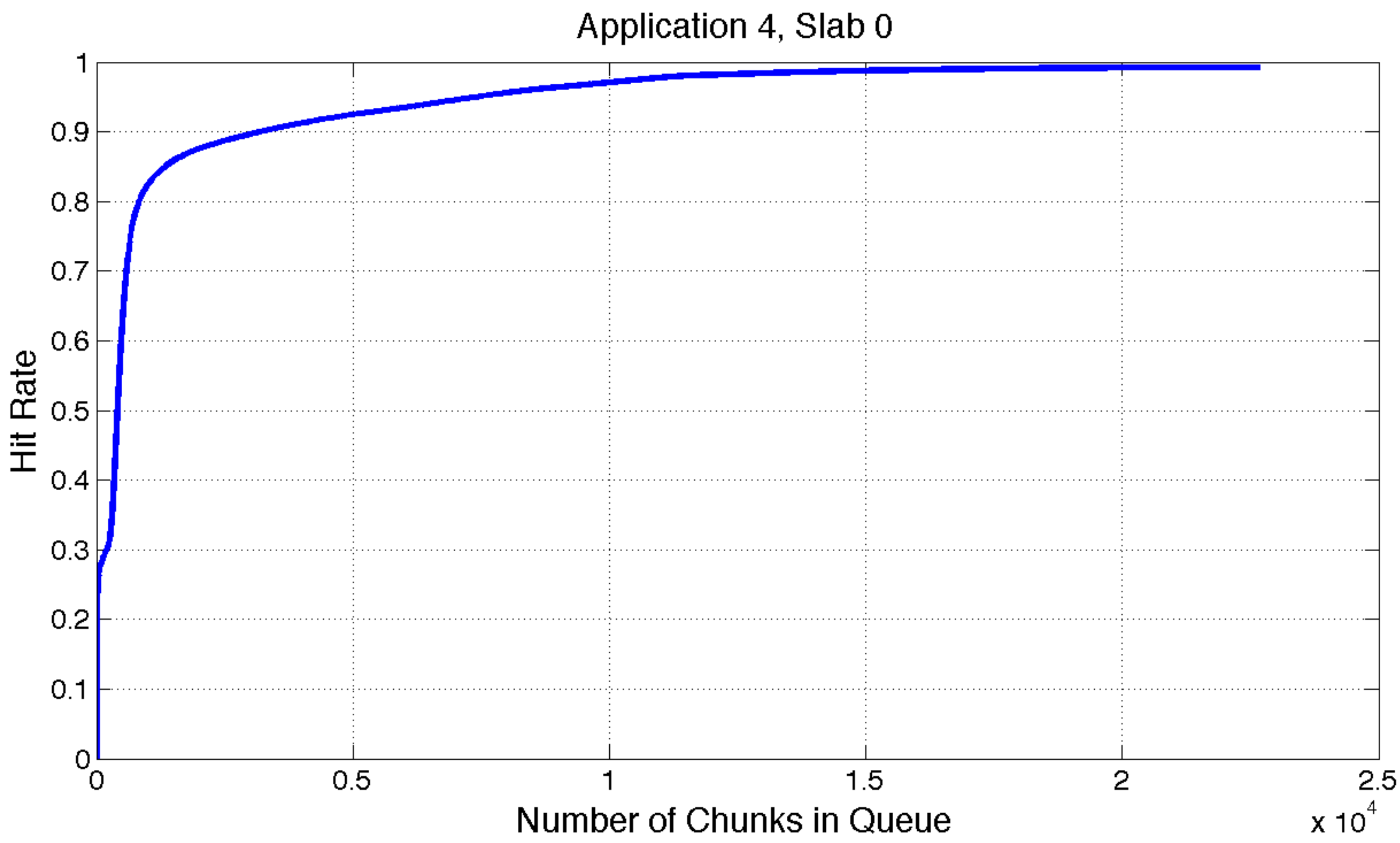
- Can we do better?

Profiling Hit Rate Curves

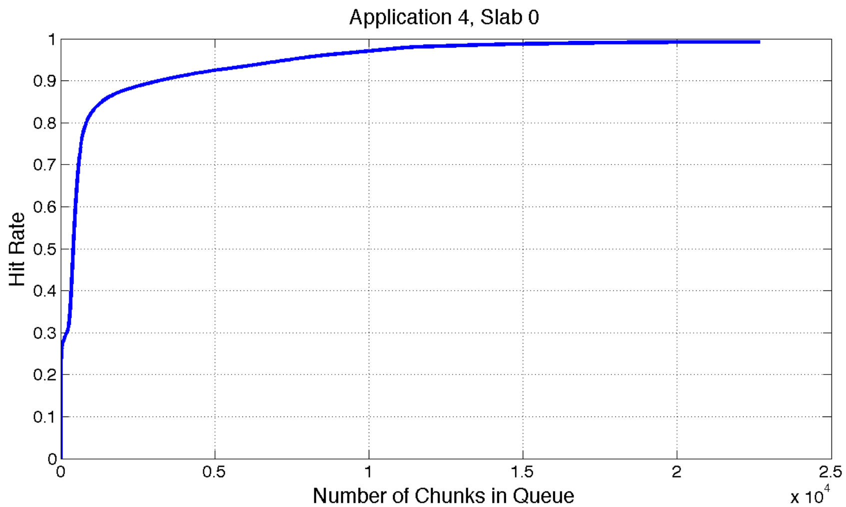


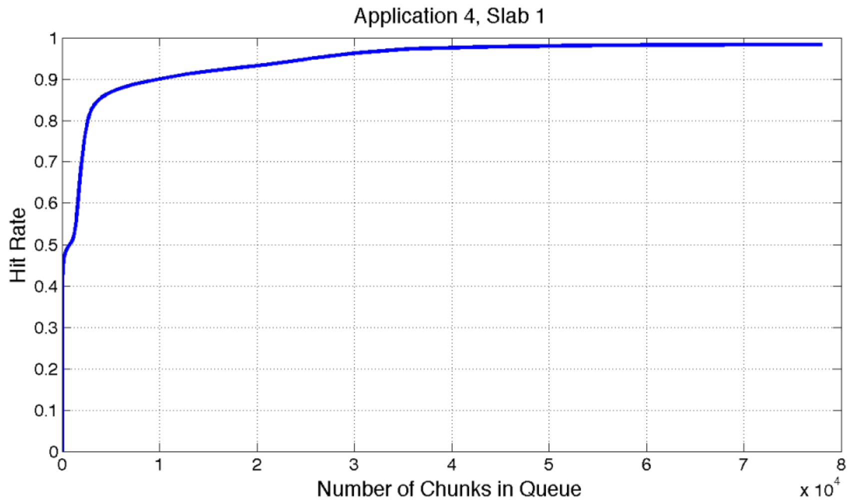
Stack distances:
5
1
∞

Hit Rate Curve Profiling



Optimizing Hit Rate Curves





Memory Allocation Solver Using Hit-rate Curves

$$\underset{m}{\text{maximize}} \quad \sum_{i=1}^{s} f_i h_i(m_i)$$

$$\text{subject to} \quad \sum_{i=1}^{s} m_i \leq M$$

f – frequency of requests

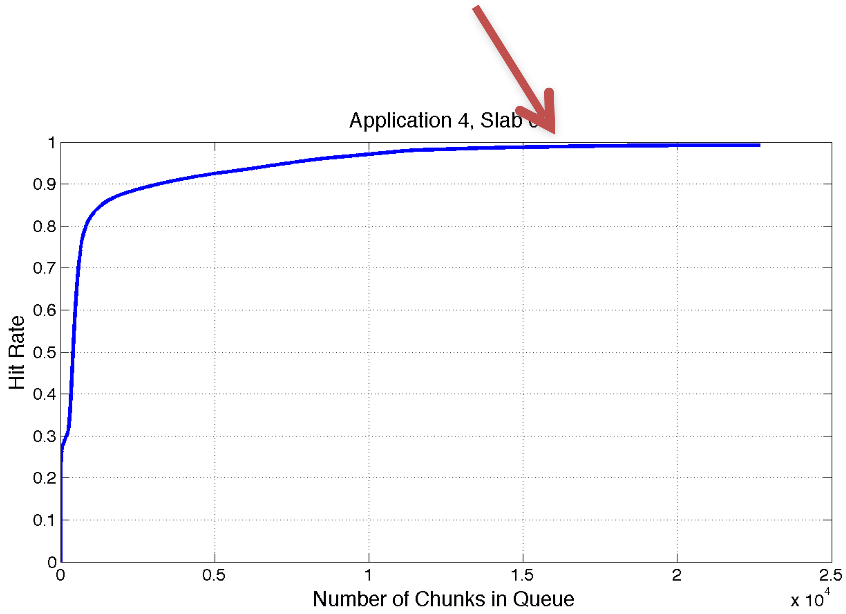
h – hit-rate of requests

m – memory allocated to slab class

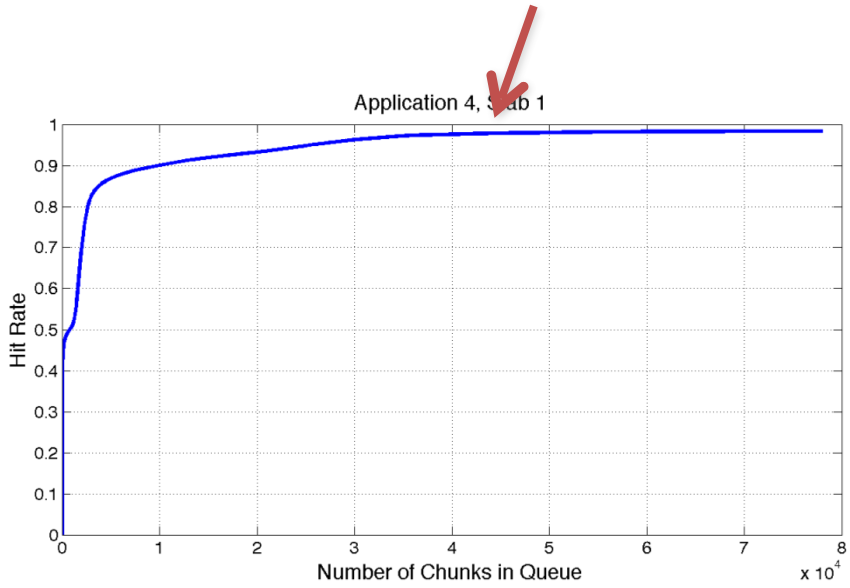
M – memory allocated to application

Solver Output

Allocate 1178 Items



Allocate 41381 Items

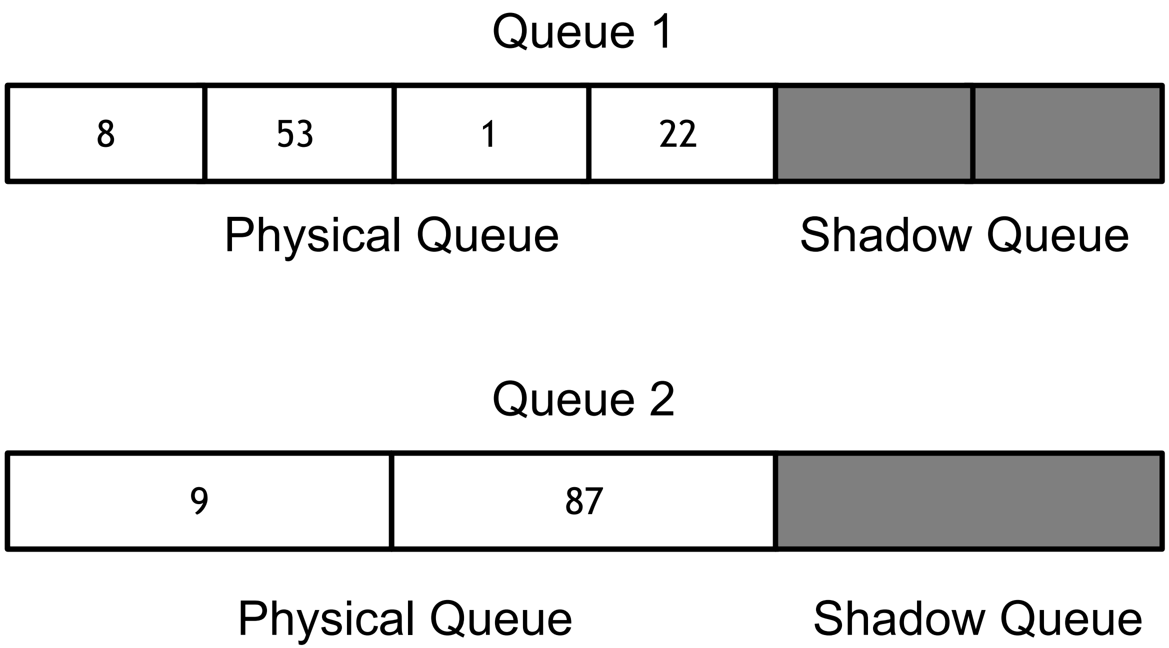


Solver is Expensive and Not Dynamic

- Solver is expensive
  - Requires estimating stack distances for each curve
  - Requires centralized solver
- Solver is static
  - How frequently should we optimize?

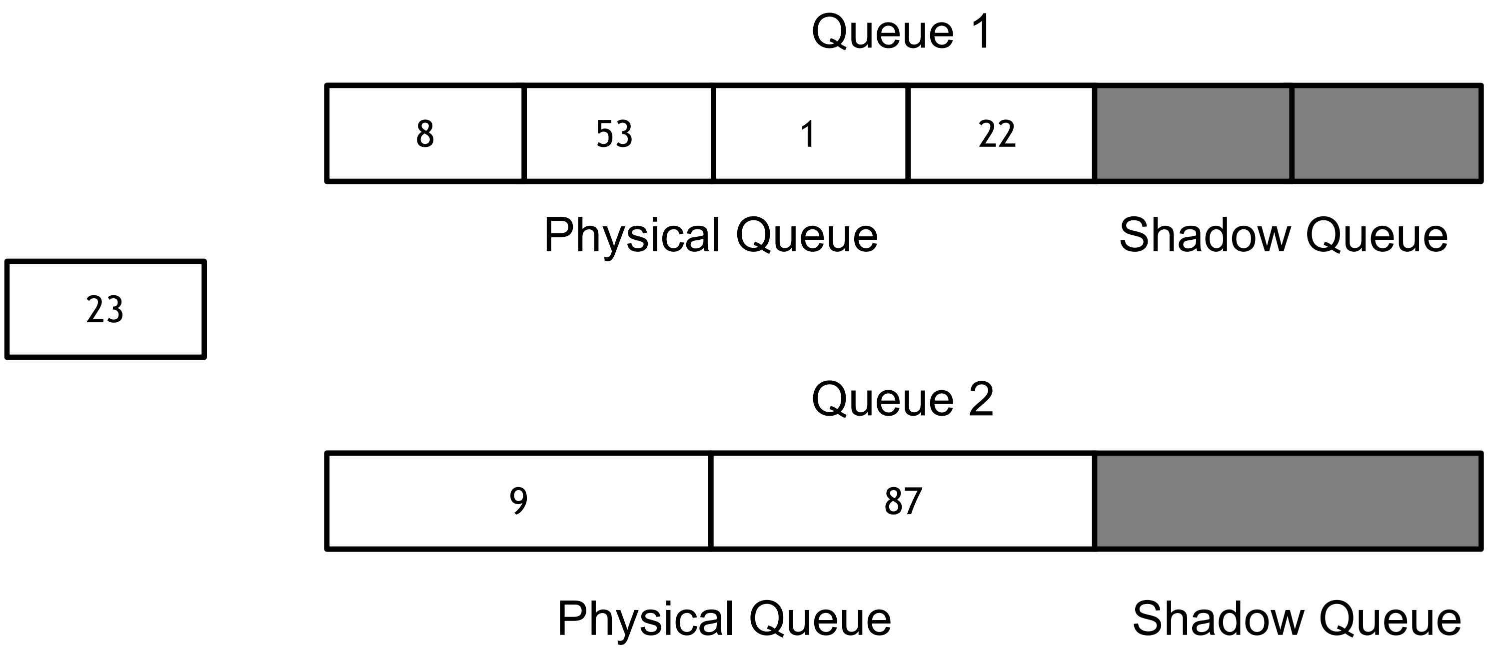
- Instead of optimizing entire hit rate curve, we can optimize incrementally
  - Estimate local gradient for each curve
  - Increase memory for curve with highest gradient

Using Shadow Queues to Estimate Local Gradient



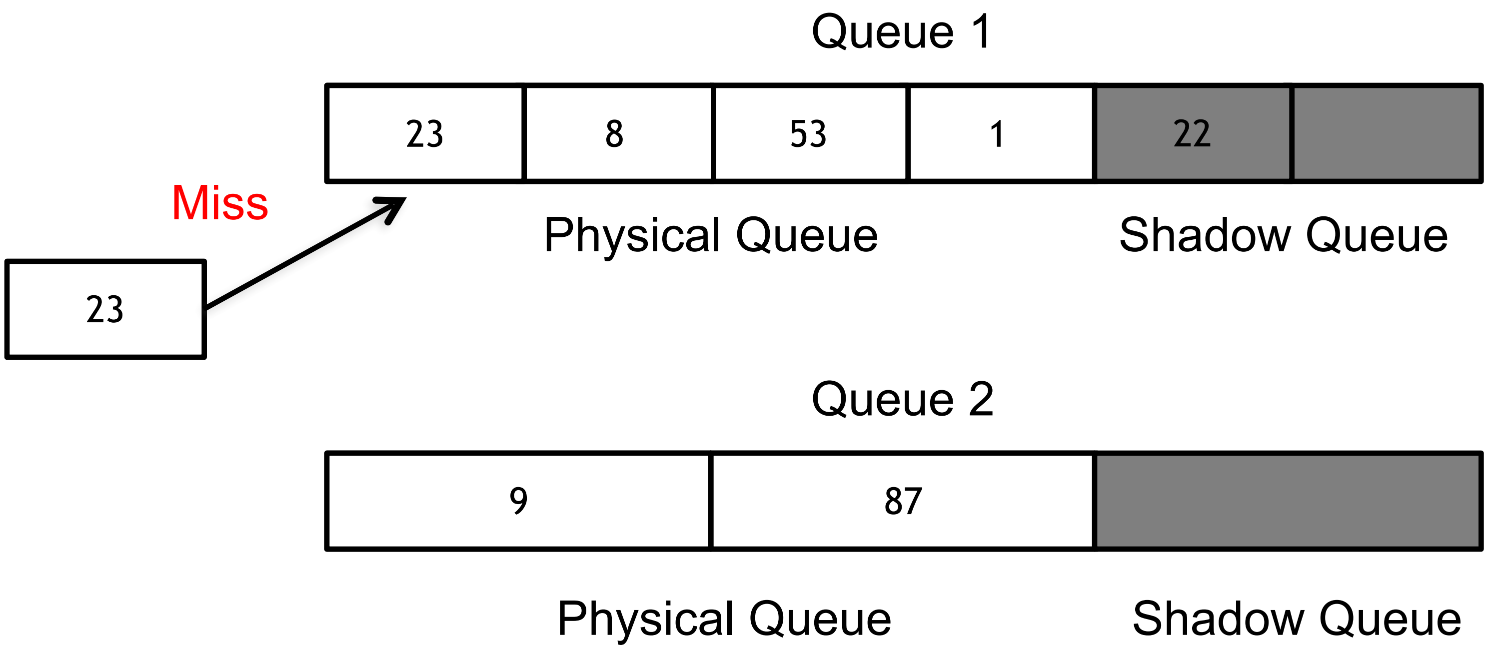
	Credits
Queue 1	0
Queue 2	0

Using Shadow Queues to Estimate Local Gradient



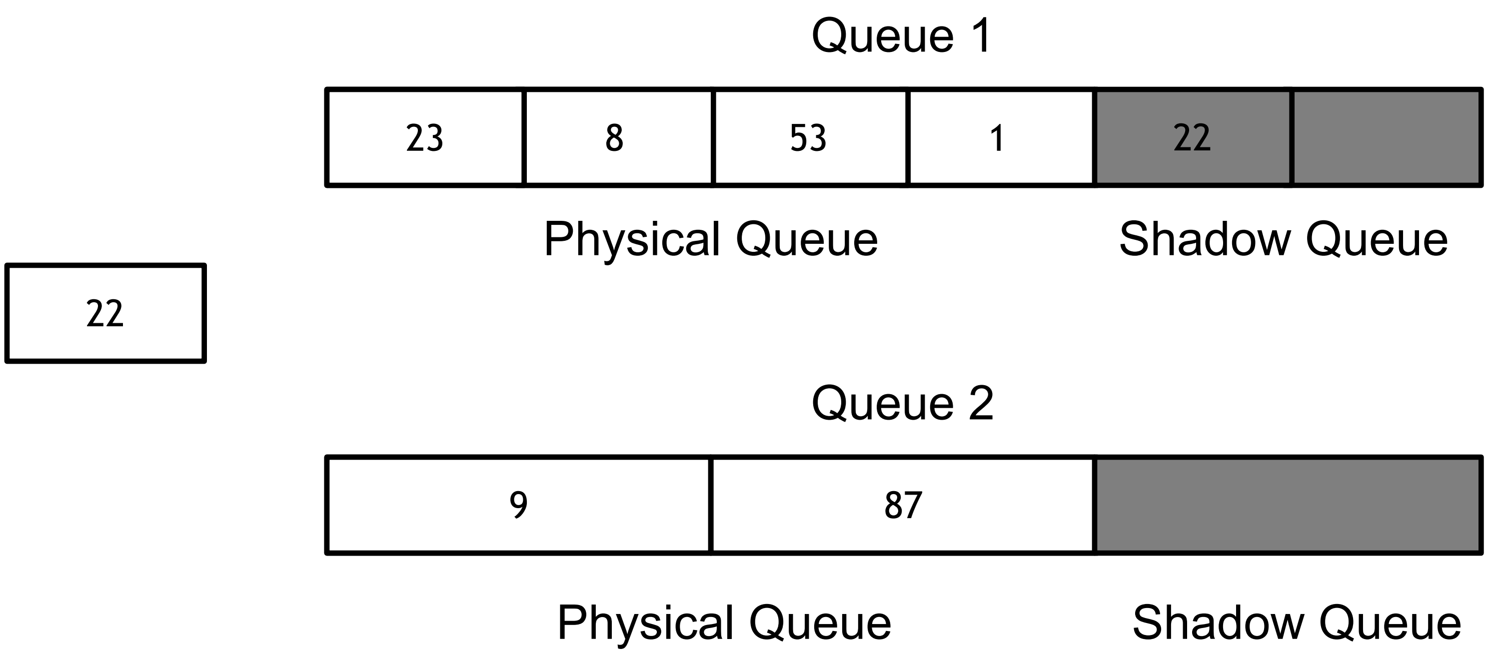
	Credits
Queue 1	0
Queue 2	0

Using Shadow Queues to Estimate Local Gradient



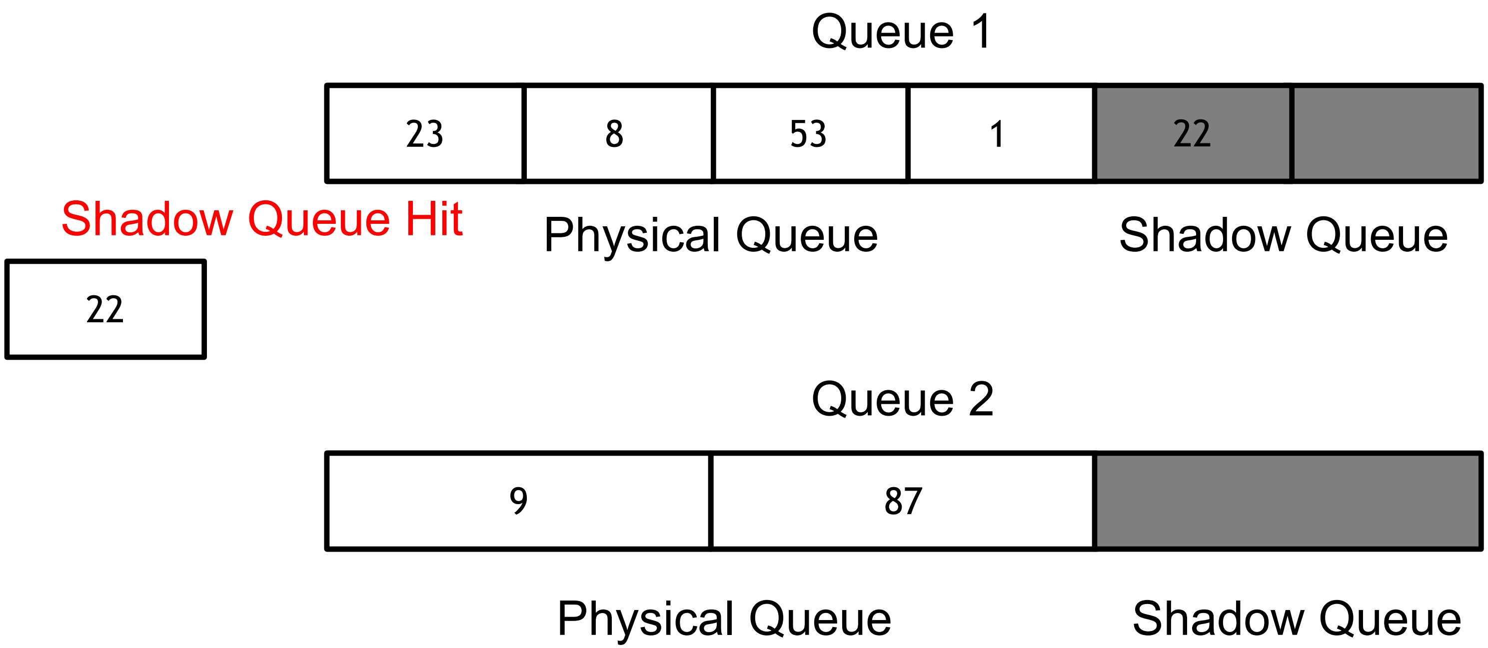
	Credits
Queue 1	0
Queue 2	0

Using Shadow Queues to Estimate Local Gradient



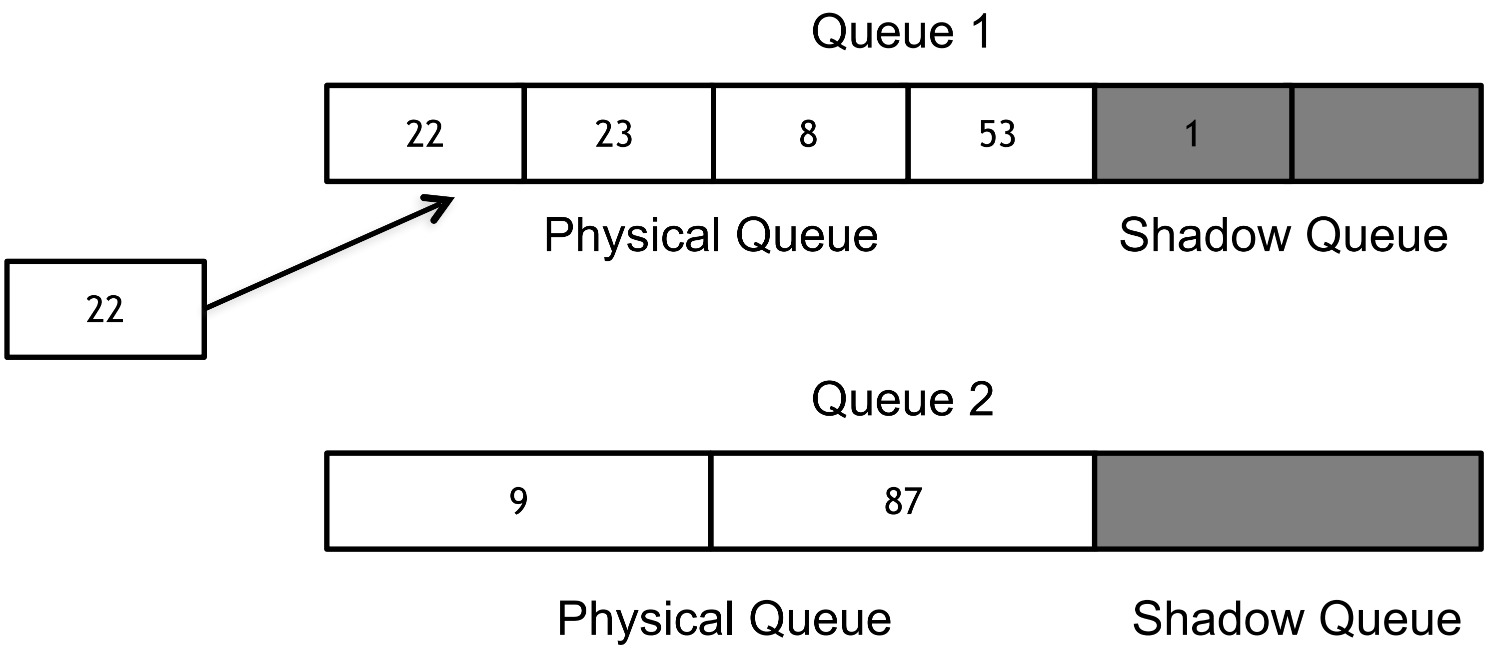
	Credits
Queue 1	0
Queue 2	0

Using Shadow Queues to Estimate Local Gradient



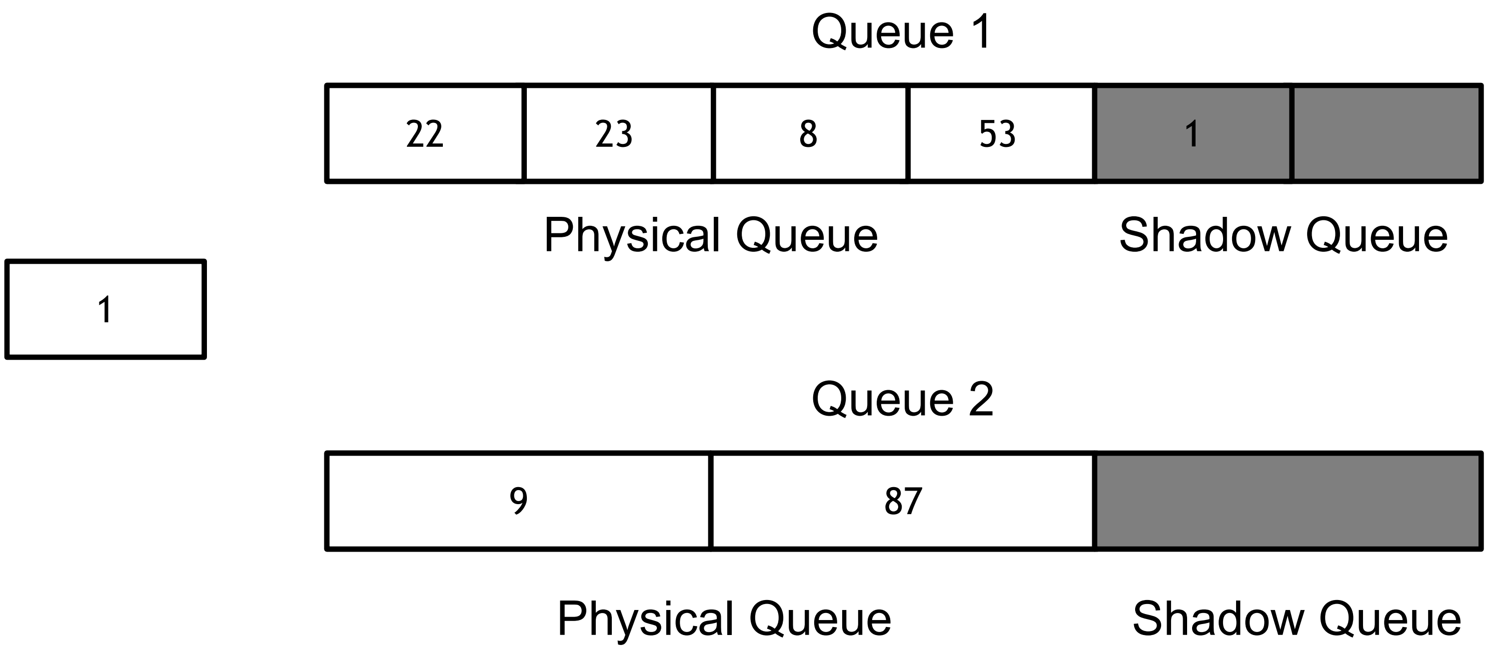
	Credits
Queue 1	1
Queue 2	-1

Using Shadow Queues to Estimate Local Gradient



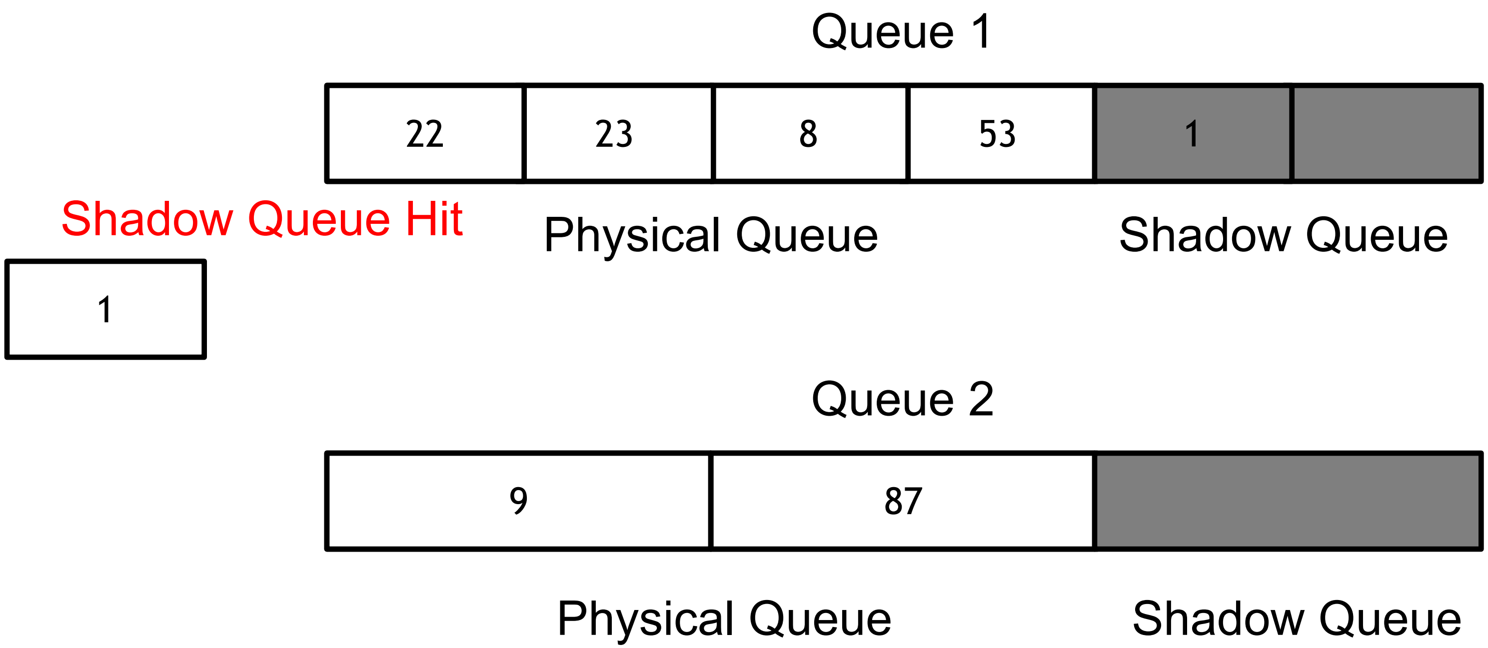
	Credits
Queue 1	1
Queue 2	-1

Using Shadow Queues to Estimate Local Gradient



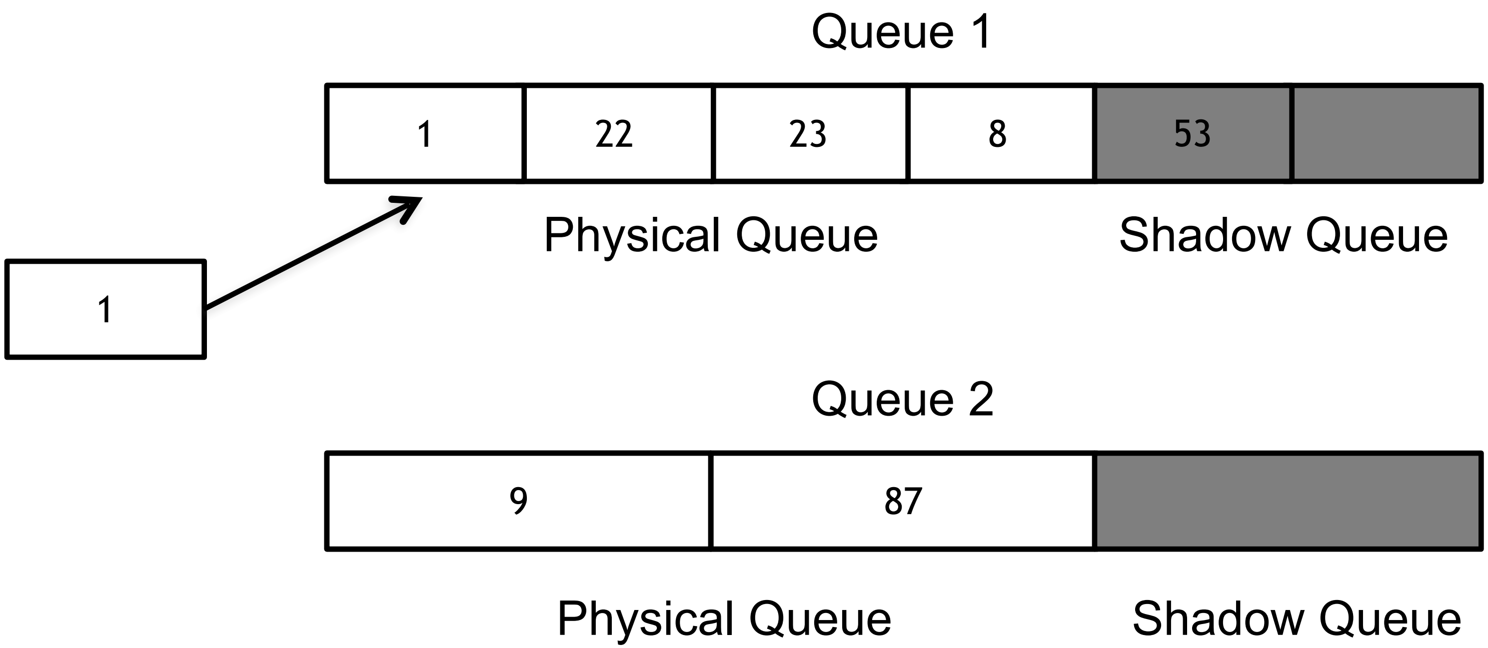
	Credits
Queue 1	1
Queue 2	-1

Using Shadow Queues to Estimate Local Gradient



	Credits
Queue 1	2
Queue 2	-2

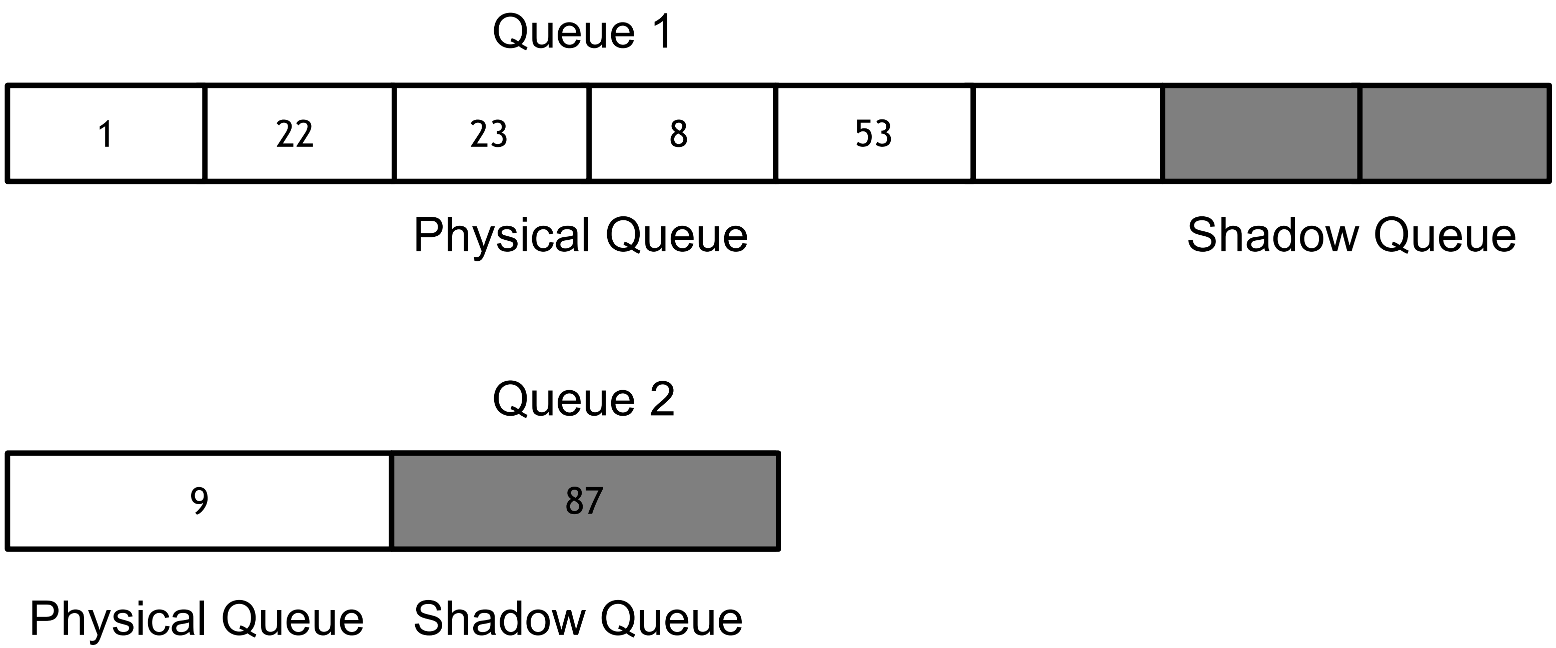
Using Shadow Queues to Estimate Local Gradient



	Credits
Queue 1	2
Queue 2	-2

Resize Queues

Using Shadow Queues to Estimate Local Gradient



	Credits
Queue 1	0
Queue 2	0

Algorithm 1: Hill-climbing

Algorithm 1 Hill Climbing Algorithm

```
if request ∈ shadowQueue(i) then
    queue(i).size = queue(i).size + credit
    chosenQueue = pickRandom({queues} - {queue(i)})
    chosenQueue.size = chosenQueue.size - credit
end if
```

Algorithm 1: Hill-climbing

Algorithm 1 Hill Climbing Algorithm

```
if request ∈ shadowQueue(i) then
    queue(i).size = queue(i).size + credit
    chosenQueue = pickRandom({queues} - {queue(i)})
    chosenQueue.size = chosenQueue.size - credit
end if
```

Approximates optimization

- At optimal memory allocation:
  - Credit increase rate = credit decrease rate for each queue

Performance Guarantee

- Assumption: hi(mi) are increasing and concave
- Optimality condition

$$f_i h_i'(m_i) = \gamma \text{ for } 1 \le i \le s$$

$$\sum_{i=1}^{s} m_i = M$$

- Algorithm guarantee:
  - cr

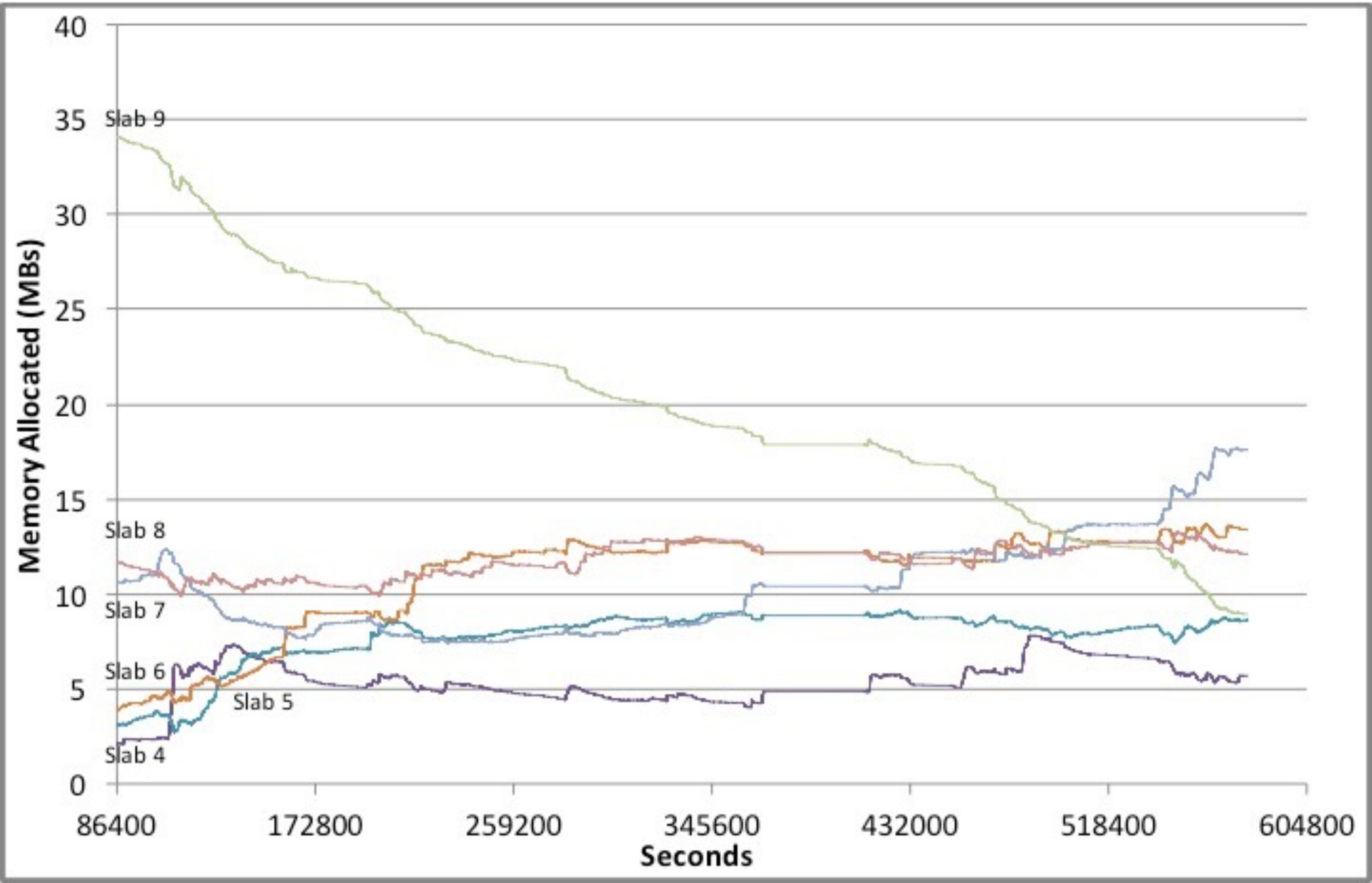
$$f_i(h_i(m_i+\delta) - h_i(m_i)) \cdot \epsilon \approx f_i h_i'(m_i) \cdot \delta \cdot \epsilon$$

  - cr

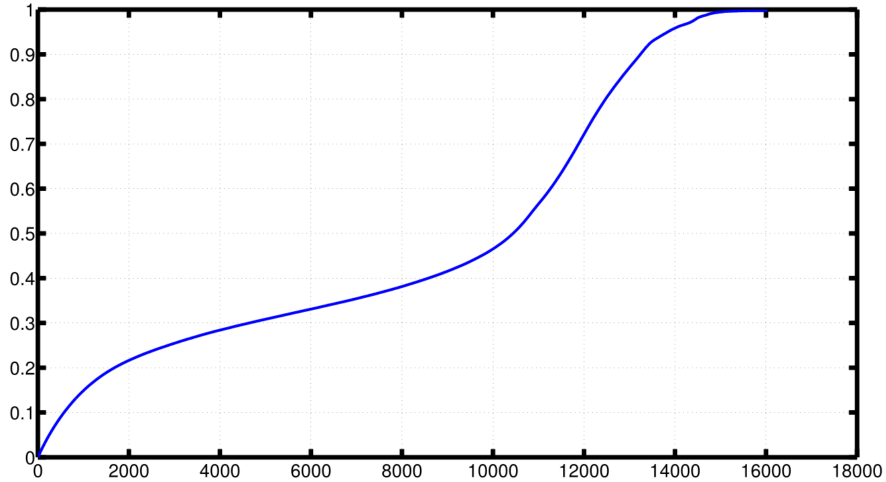
$$\frac{\sum_{j=1}^{s} f_j(h_j(m_j+\delta) - h_j(m_j)) \cdot \epsilon}{s} \approx \frac{\sum_{j=1}^{s} f_j h_j'(m_j) \cdot \delta \cdot \epsilon}{s}$$

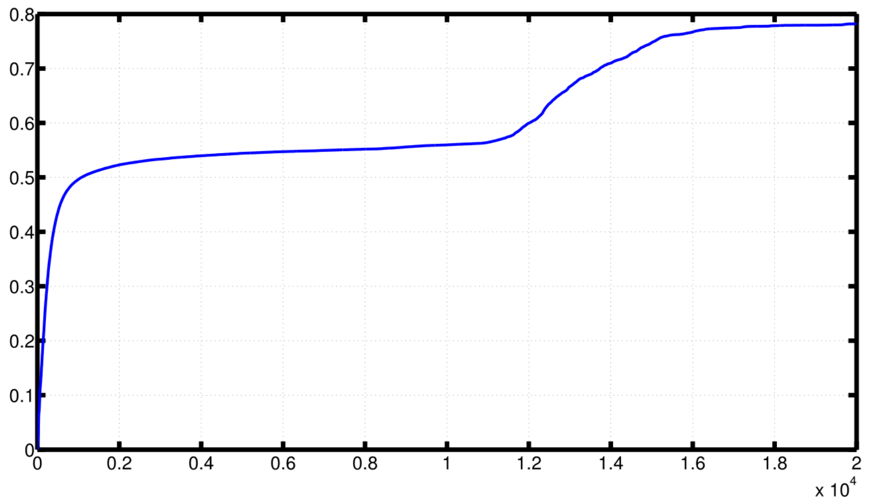
$$f_i h_i'(m_i) = \frac{\sum_{j=1}^{s} f_j h_j'(m_j)}{s} = \gamma$$

Hill Climbing Algorithm Over Time (Application 5)



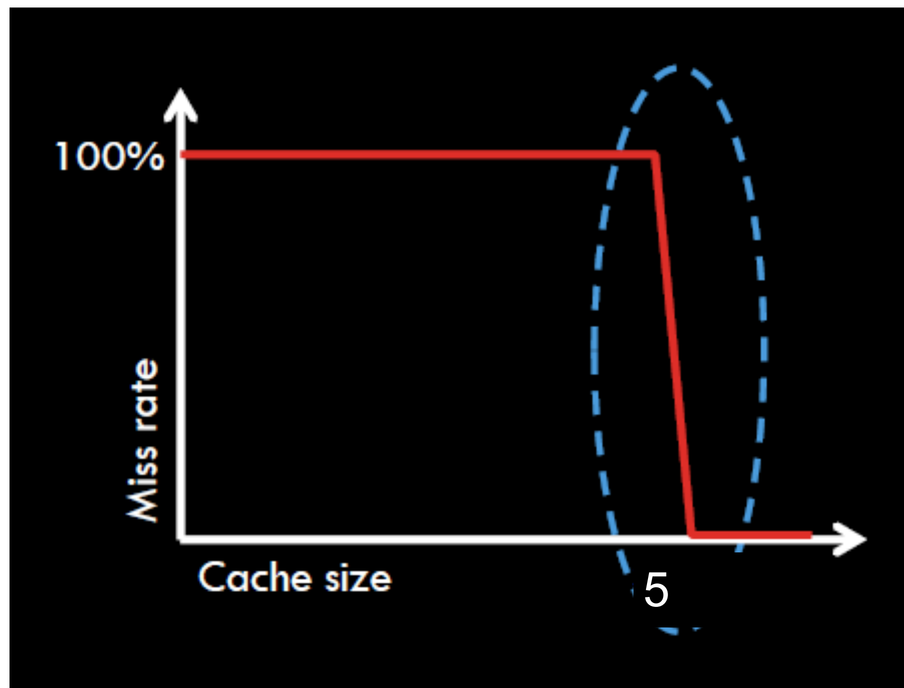
Performance Cliffs Hurt Local Optimization





Why Do Performance Cliffs Occur?

- Applications issues requests 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, …
- Queue size = 4
  - 0% hitrate
- Queue size = 5
  - 100% hitrate



Talus: Goal

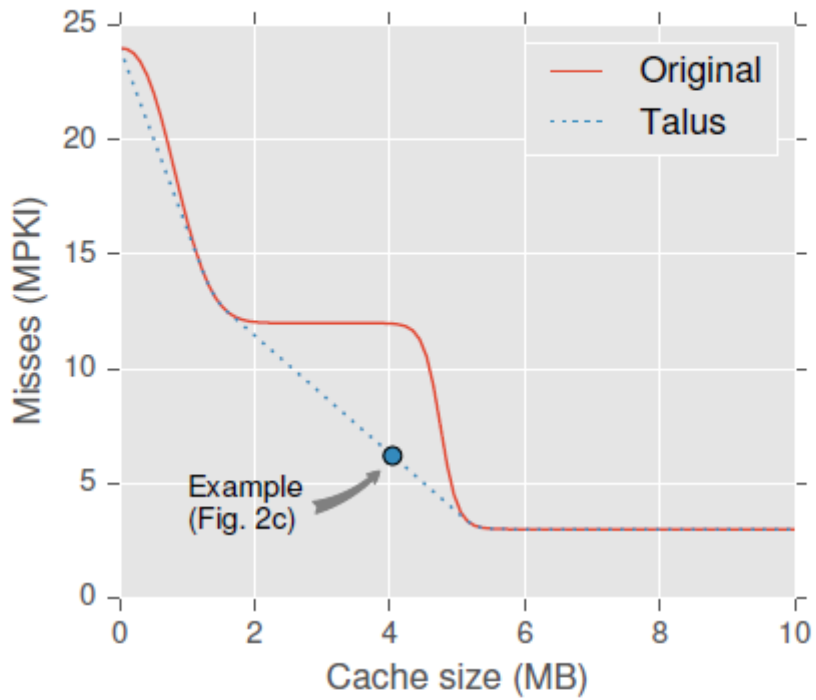


Fig. 3: Example miss curve from an application with a cliff at 5 MB. Sec. III shows how Talus smooths this cliff at 4 MB.

Talus allows us to achieve a hit rate that is a linear interpolation between any two points in the hit rate curve

Talus: Idea

Example: a-z

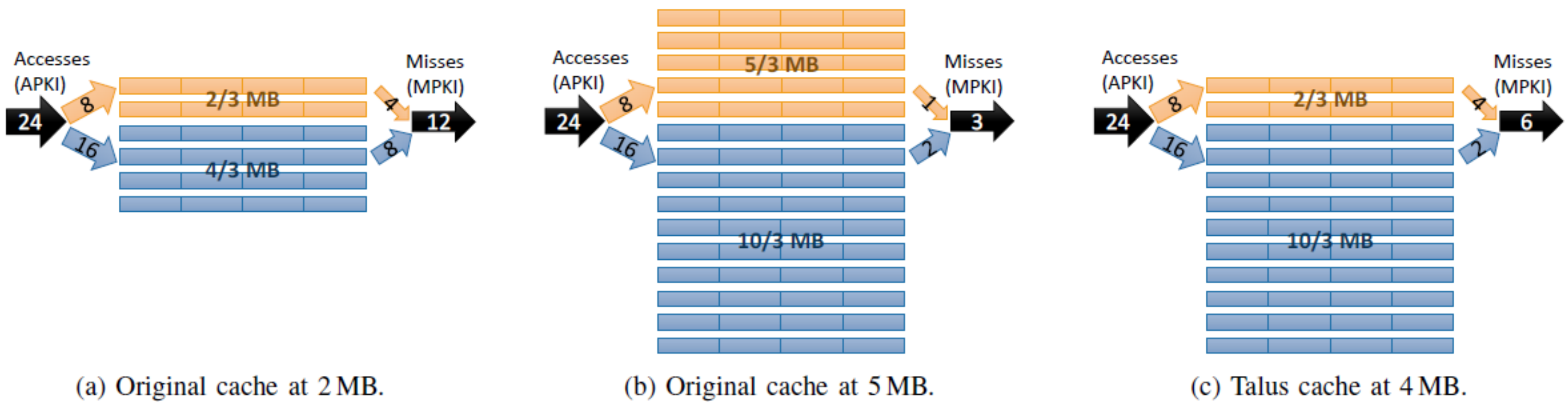


Fig. 2: Performance of various caches for the miss curve in Fig. 3. Fig. 2a and Fig. 2b show the original cache (i.e., without Talus), conceptually dividing each cache by sets, and dividing accesses evenly across sets. Fig. 2c shows how Talus eliminates the performance cliff with a 4 MB cache by dividing the cache into partitions that *behave like the original* 2 MB (top) and 5 MB (bottom) caches. Talus achieves this by dividing accesses in *dis*-proportion to partition size.

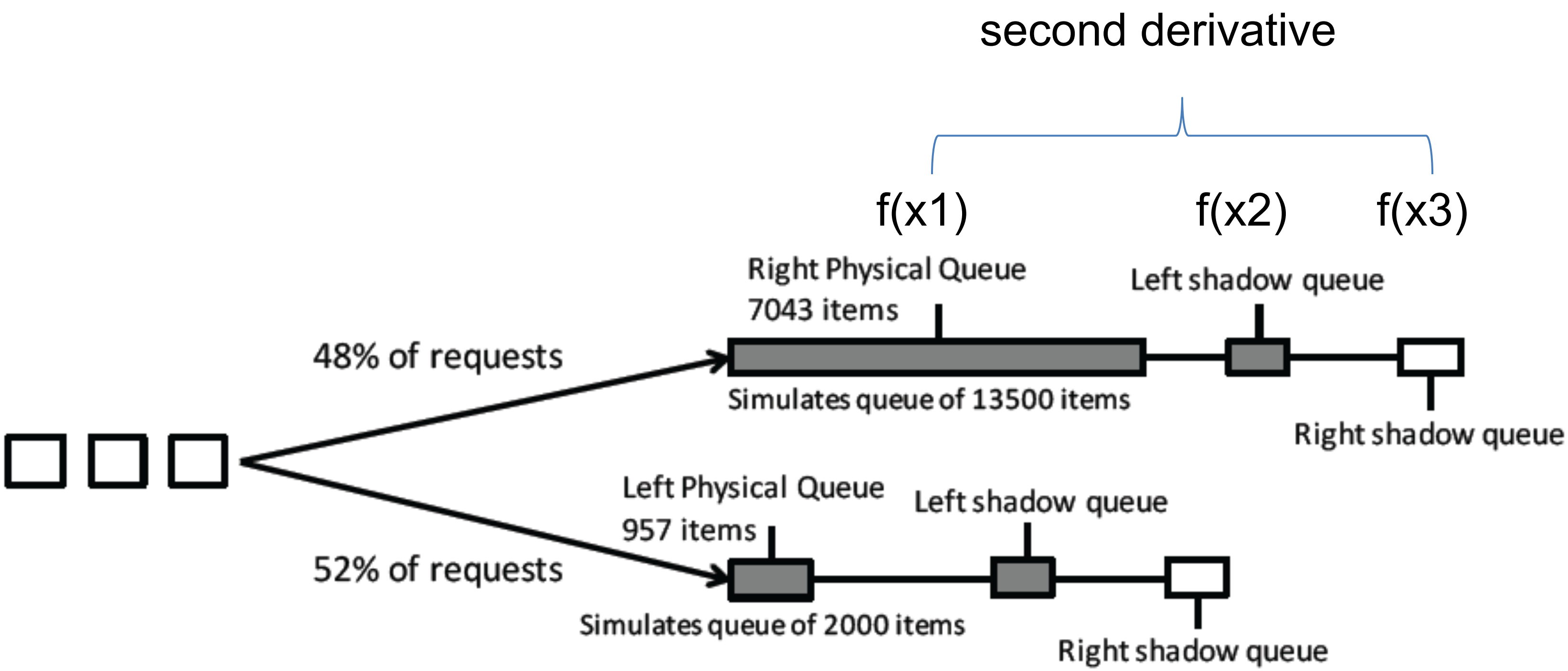
$2x + 5(1-x) = 4 \quad \Rightarrow \quad x = 1/3$

control the size of the two partitions as well as
how accesses are distributed between them

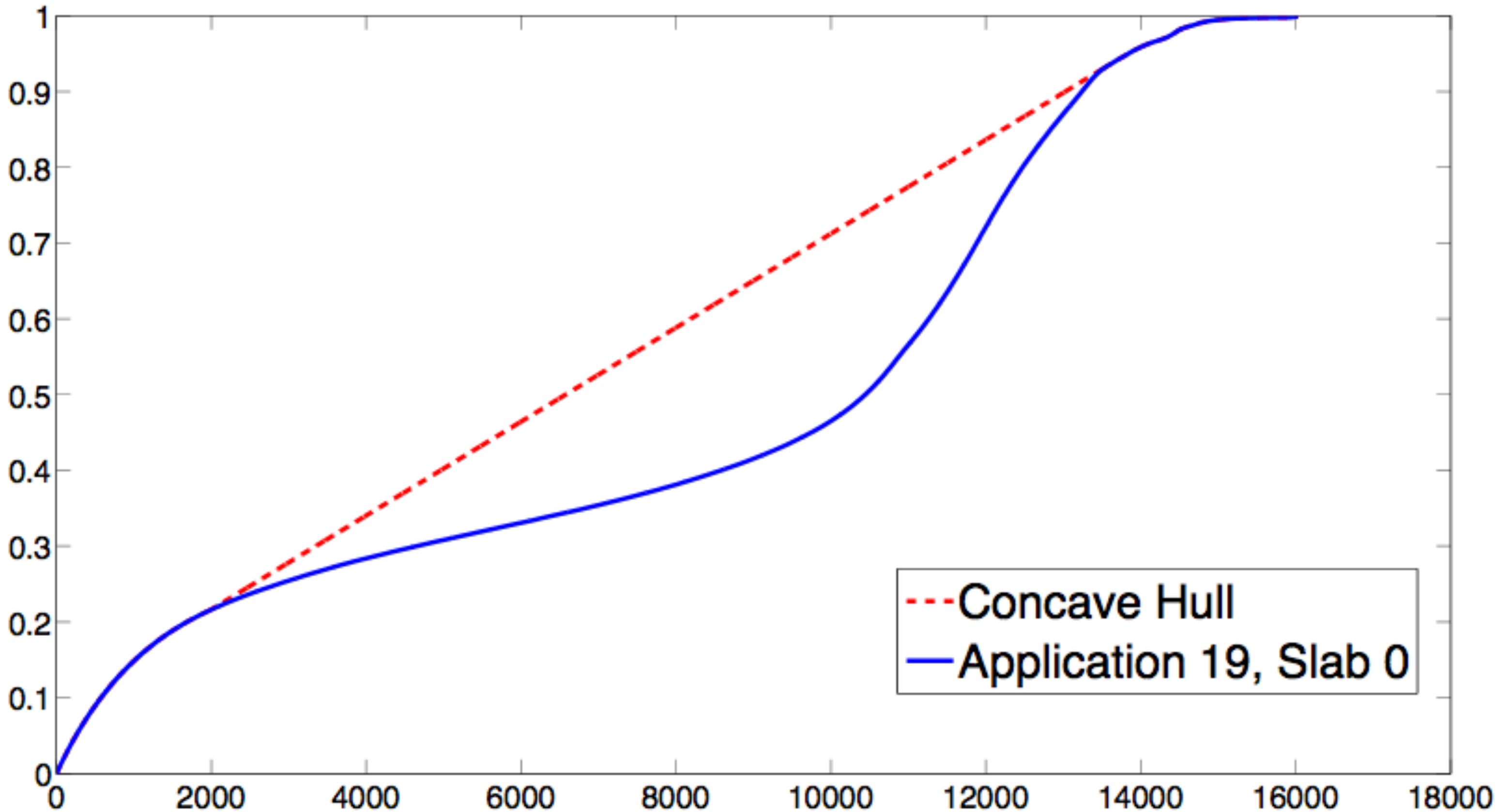
Algorithm 2: Cliff Scaling

- Talus requires knowledge of hitrate curve
  - Where the performance cliff starts and ends
- Algorithm 2 locally estimates where the performance cliff starts and ends
  - Estimate the second derivative with shadow queues

Visualization of Shadow Queues



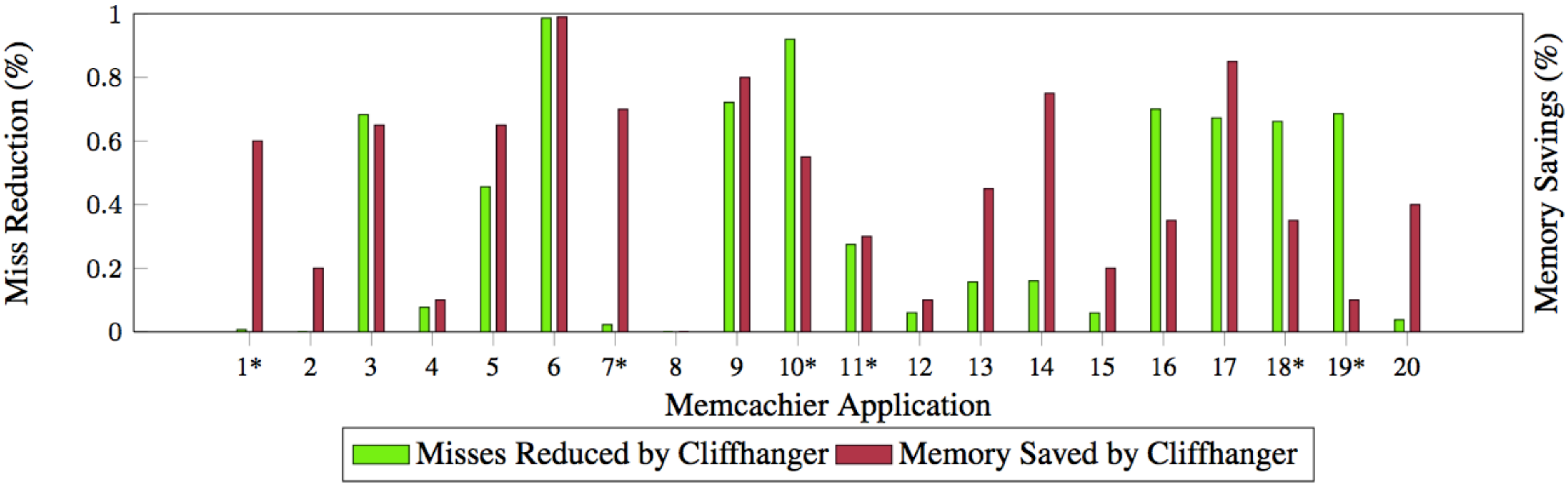
Estimating Second Derivative with Shadow Queues



Cliffhanger Runs Both Algorithms in Parallel

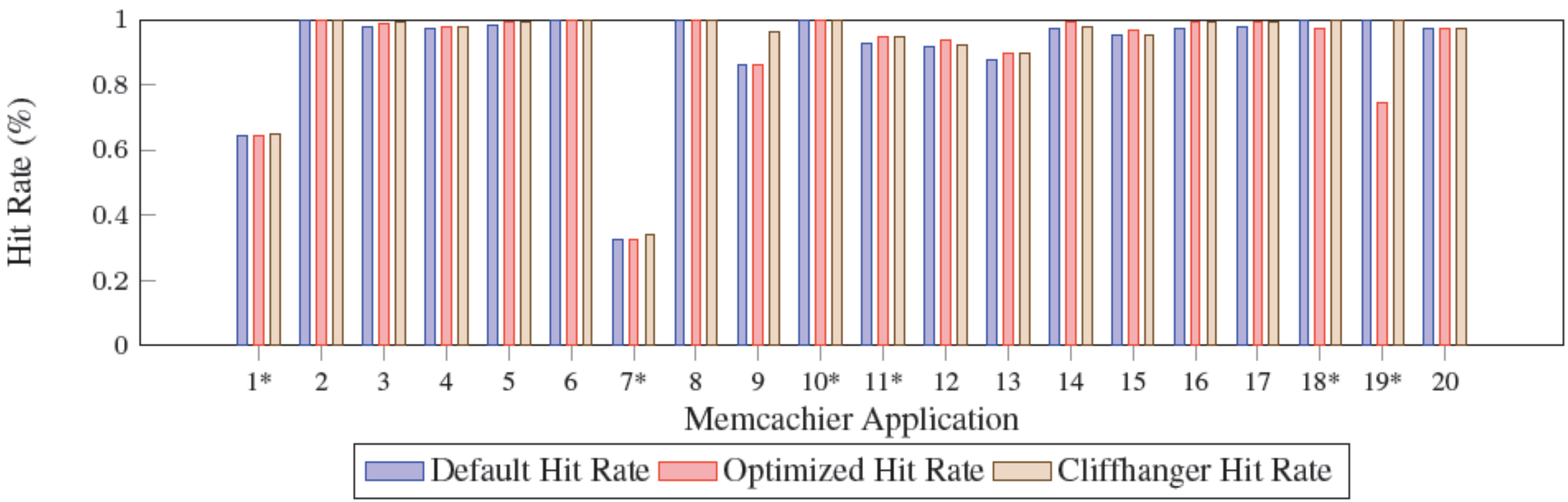
- Algorithm 1: incrementally optimize memory across queues
  - Across slab classes
  - Across applications
- Algorithm 2: scales performance cliffs

Cliffhanger Reduces Misses and Can Save Memory



- Average misses reduced: 36.7%
- Average potential memory savings: 45%

Cliffhanger Outperforms Default and Optimized Schemes



- Average Cliffhanger hit rate increase: 1.2%

Low Overheads

- Latency overhead:

Algorithm	Operation	Cache Hit	Cache Miss
Hill Climbing	GET	0%	1.4%
Hill Climbing	SET	0%	4.7%
Cliffhanger	GET	0.8%	1.4%
Cliffhanger	SET	0.8%	4.8%

- Throughput overhead:

% GETs	% SETs	Throughput Slowdown
96.7%	3.3%	1.5%
50%	50%	3%
10%	90%	3.7%

- Memory overhead: 500KB for each application

Summary

- Web-scale applications heavily reliant on memory cache hit rate
- But, existing cache allocation is not optimized for max hit rate
- Cliffhanger's incremental dynamic cache allocation using shadow queues maximizes hit rates and addresses performance cliffs

Appendix

Related Work

- Cache partitioning for performance cliffs
  - Talus: Beckmann et al [HPCA ‘15]
- Optimizing memory allocation across applications based on hitrate curves
  - Mimir: Saemundsson et al [SOCC ‘14]
- Rebalancing slabs to reduce slab calcification
  - Twitter: Rajashekhar et al [Twitter blog ’12]
  - Facebook: Nishtala et al [NSDI ’13]
- Optimizing Memcached multi-threaded performance
  - MICA: Lim et al [NSDI ’14]

Comparison with “Facebook LRU”

Application	Original Hitrate	Facebook Hitrate	Cliffhanger + LRU Hitrate	Cliffhanger + Facebook Hitrate
3	97.7%	97.8%	99.3%	99.3%
4	97.4%	97.6%	97.6%	97.6%
5	98.4%	98.5%	99.1%	99.1%

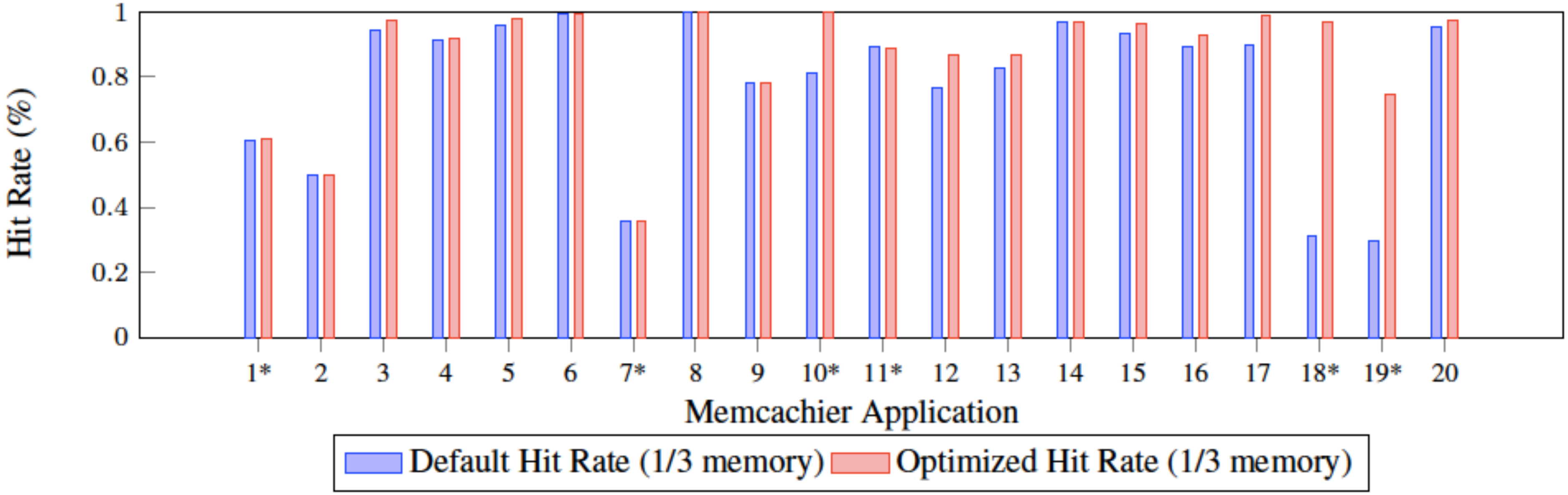
Log Structured Memory is Still Greedy

Application	Original Hitrate	Log-structured Hitrate	Dynacache Solver Hitrate
3	97.7%	99.5%	98.8%
4	97.4%	97.8%	97.6%
5	98.4%	98.6%	99.4%

Algorithms are Complementary (Memcachier's Application 19)

Slab Class	Original Hitrate	Cliff Scaling Hitrate	Hill Climbing Hitrate	Combined Algorithm Hitrate
0	38.1%	44.8%	95.3%	98.3%
1	37.3%	45.6%	67.4%	69.1%
Total Hitrate	37.3%	45.5%	70.3%	72.1%

Solver's Potential for Improvement



Solver's Potential for Improvement