Caches in the Modern Memory Hierarchy with Persistent Memory and Flash

Latest copy: https://ymsir.com/fast19

Irfan Ahmad[1] Ymir Vigfusson[2*]

[1] CachePhysics

[2]Emory University/Reykjavik University

Monday 25th February, 2019

*Supported in part by NSF CAREER award #1553579.

Motivation

To cache is to scale

For a very long time, practical scaling of every level in the computing hierarchy has required innovation and improvement in cache management. This is as true for CPUs as it is for storage and networked, distributed systems. As such, research into cache efficiency and efficacy improvements has been highly motivated and continues with strong improvements to this day.

Recent developments:

- Memory hierarchies keep getting more tiered and complex.
- Hardware innovations are changing previous latency assumptions.
- New cache modeling techniques could spark a rebirth of the field.

An Non-computer Example of a Cache

Figure: A fixed sized table-top cache of often accessed books in a library. The contents are this cache are selected by a replacement algorithms.

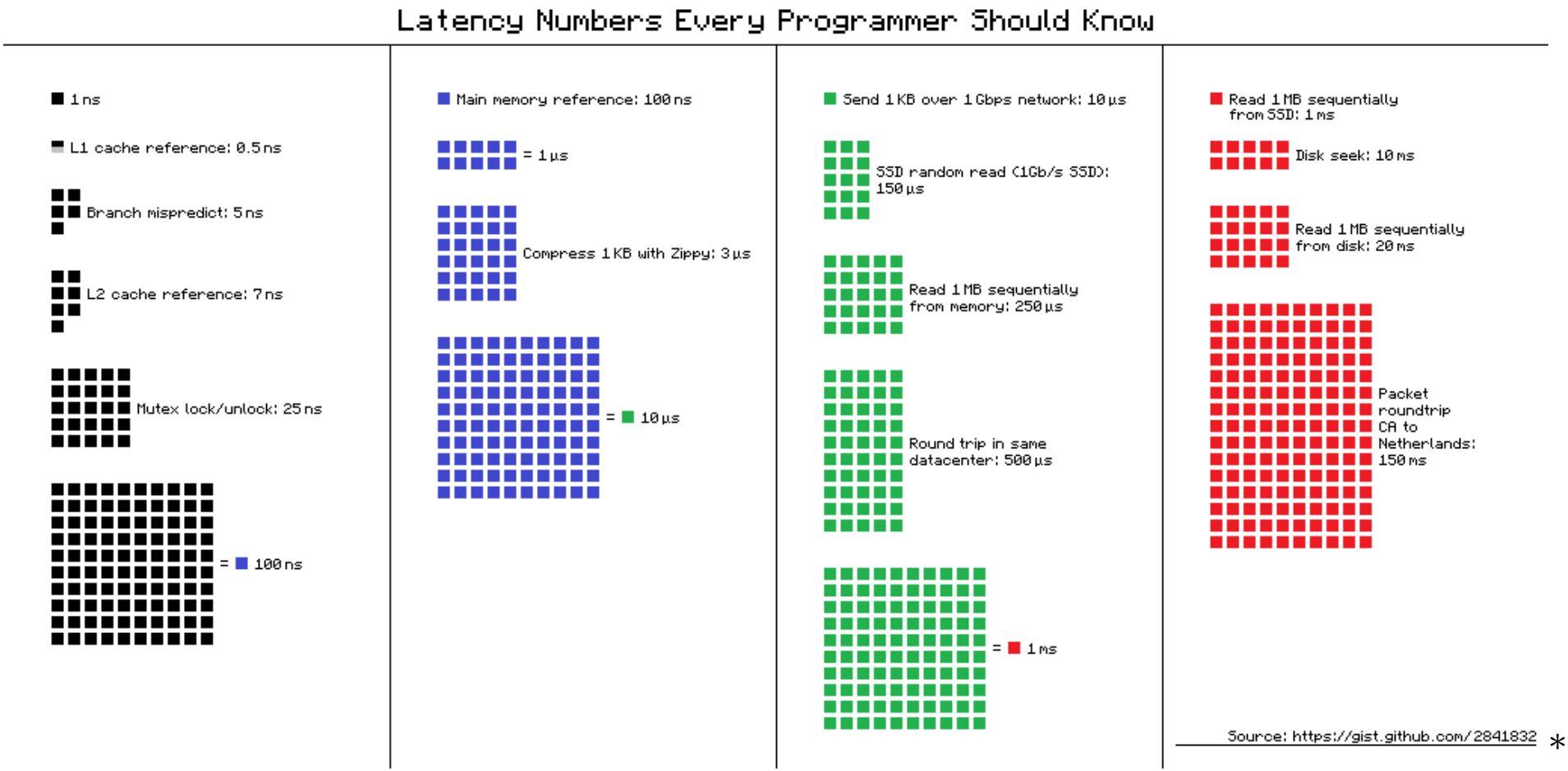
What is a Cache?

- A computer program accesses data from a bunch of "memory"
- Not all memory is created equal
- Faster memory is often more expensive and/or less dense
- Desire: reduce the time waiting for accesses to slow memory
- Ideal: store data likely to be needed in the future in faster memory

Definition

... hardware or software component that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere.
—*Wikipedia*

Scale of Latencies (circa 2016)



*https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Trade-offs

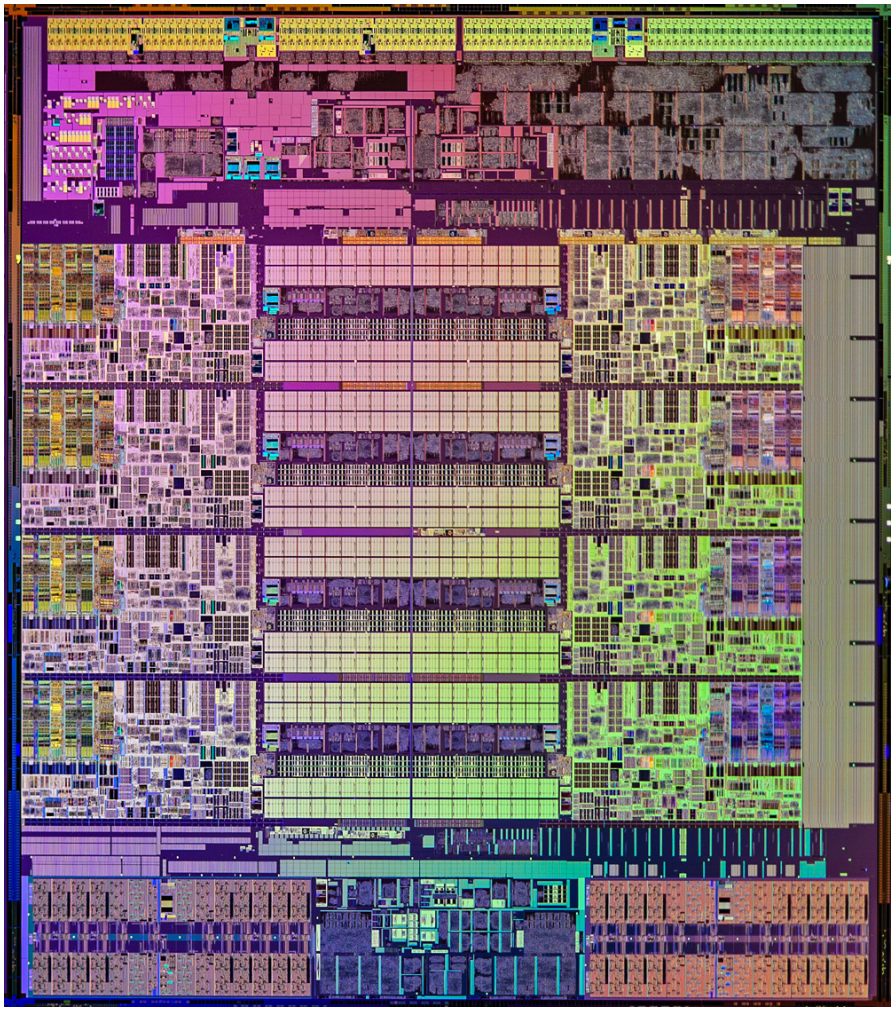
- Inherent trade-off between size and speed
- Between expensive technologies (such as SRAM, DRAM) vs cheaper commodities (such as Flash or hard disks)
- For same overall cost, can have many different configurations
  - A little more higher-level cache for a lot less lower-level cache
  - A lot more lower-level cache for a little less higher-level cache
  - Often have more than two levels of caches, configuration problem is non-trivial

	Tran. per bit	Access time	Persist?	Sensitive?	Cost
SRAM	6	1X	Yes	No	100x
DRAM	1	10X	No	Yes	1X

Figure: Ballpark comparison of an example fast, higher-level SRAM technology versus denser, lower-level DRAM.

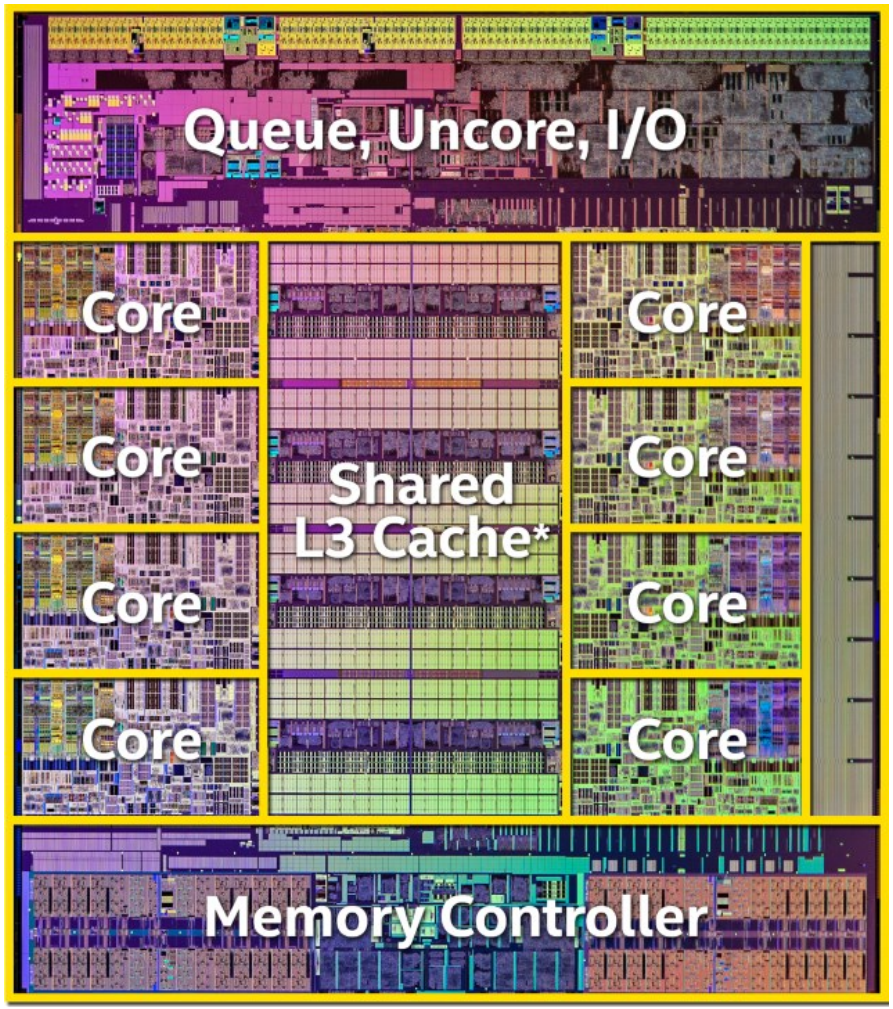
*http://www.cs.rochester.edu/users/faculty/sandhya/csc252/lectures/lecture-storage.pdf

How Essential?



- A large amount of die area on modern processors is dedicated to caches
- This example is an Intel Haswell processor (via extremetech.com)
- Not labeled here are the large Level-2 (L2) caches

How Essential?



- A large amount of die area on modern processors is dedicated to caches
- This example is an Intel Haswell processor (via extremetech.com)
- Not labeled here are the large Level-2 (L2) caches

Terminology

- Cache are modeled like other memory, responding to an online sequence of data requests called *references*

Hit

A cache *hit* occurs when the requested data can be found in the cache.

Miss

A cache *miss* occurs when requested data is not found in the cache.

- *Compulsory/Cold*: first reference to an entry (address/block, key/value pair)
- *Capacity*: cache not big enough to hold every entry
- *Conflict*: multiple entries mapped to same already full cache location

Terminology

- Cache performance ultimately boils down to the latency of memory accesses.
- The *effective access time* equals the average:

  $$\text{EAT} = \text{hit-ratio} \times \text{cache-access-time} + \text{miss-ratio} \times \text{slow-memory-access-time}$$

- Often dominated by the second term, so objective becomes to minimize the miss ratio.

Rates and Ratios

Both cache miss rates and miss ratios are used in literature, depending upon context. Rates are often measured in relative time, *e.g.* misses per 1000 instructions (*MPKI*). There doesn't appear to be a standard convention between using miss ratios (lower is better) versus hit ratios (higher is better).

Cache is Full, now what?

- On cache miss, a new entry needs to go into the cache. If cache is full, some other entry has to leave.

Cache replacement

The policy that selects which entry is thrown out (replaced) is referred to as a *cache replacement policy*, *page replacement policy* or *eviction policy* depending upon context.

- Workload performance is often very sensitive to the eviction policy.
- Eviction policies have traditionally been used in two related contexts: buffer caches and virtual memory.

Unified Buffer Caches

- In operating system (OS) buffer caches, file or block reads and writes are explicit and OS-visible making total ordering feasible.
- In the case of virtual memory managed via page tables, reads and writes are not directy OS visible. However, “access” or “use” bits maintained by hardware can be sampled to get approximate ordering.
- More recently, OSes have started to use unified buffer caches that unify the eviction schemes across both.

Eviction Policies

Examples of eviction policies and approximate versions suitable for implementation in virtual memory systems.

Original	Approximation	Notes
FIFO	FIFO	First-in First-out
LRU	CLOCK	Least recently used entry gets evicted
LIRS	Clock-Pro	Evict longest reuse distances
ARC	CAR	Adaptively emphasize recency versus frequency

We'll spend a few slides highlighting the ideas behind these and then turn our focus back on LRU.

LRU

Least-recently used (LRU)

- Evict the entry that was last accessed furthest back in the past. This is the most popular cache replacement policy.
- Exploits the high locality of reference that is often found in workloads.
- Think about the simplest way you'd implement LRU (a common interview question).

LRU

Least-recently used (LRU)

- Evict the entry that was last accessed furthest back in the past. This is the most popular cache replacement policy.
- Exploits the high locality of reference that is often found in workloads.
- Think about the simplest way you'd implement LRU (a common interview question).
- Data structures: index + linked list.

LRU Variants (1)

What if maintaining a linked list is too expensive?

- Second-chance/CLOCK
  - Maintain 1 reference bit/item denoting recency.
  - Bit set on access, removed in FIFO order, item evicted if bit=0
  - Used in low-level caches; IBM WebSphere eXtremeScale

What if maintaining an index is too expensive?

- LRU-k: Sample k items; delete the LRU of these (default in Redis. Used by Twitter)

Why place at front of linked list?

- Segmented LRU (SLRU): advance new item up by one segment
  - S4LRU 8% better than LRU for Facebook photo caching, reduced backend load by 23% (Huang et al. SOSP 2013)
- Can more generally think of most algorithms as priority queues.

LRU Variants (2)

What if cost of eviction varies?

Think of, *e.g.*, database query caches.

- Greedy-Dual and variants
  - Assume eviction cost can be known or estimated
  - Maintain a priority queue of Cost(x)/Size(x).
- Trick for cost estimation (Vigfusson)
  - Client tracks time between a miss and insert into cache.
  - Provide information to cache along with request

LRU Variants (3)

What if items have different sizes?

- Greedy-Dual-Size-Frequency (GDSF), popular in web caches
  - Maintain a priority queue of

$$\text{Pri}(x) = \text{clock} + \text{Freq}(x) \cdot \frac{\text{Cost}(x)}{\text{Size}(x)}$$

  - $\text{Freq}(x)$ starts at 1, incremented on each hit.
  - On a miss, evict lowest priority items until enough size available
  - Here, clock is the highest priority of evicted files
- Ended up being most competitive strategy for Facebook photo caching.

LRU Variants (4)

What if items have different sizes?

- Least Hit Density (2018) *, more rigorous
  - Define H_x as age when item x hit, and L as age when x hit *or* evicted.
  - Maintain a priority queue of

$$\text{Pri}(x) = \frac{\text{Hit-Prob}(x)}{\text{Lifetime}(x) \times \text{Size}(x)} = \frac{\sum_{i=\text{now}}^{\infty} \mathbb{P}[H = i]}{\text{Size}(x) \sum_{i=\text{now}}^{\infty} \mathbb{P}[L = \text{age}(x) + i]}$$

  - Track probabilities over **classes** of objects, grouped by frequency of references
  - Sample and evict lowest $\text{Pri}(x)$ value (like LRU-k) to reduce overhead.

*Beckmann et al. NSDI 2018

LRU Variants (5)

But recency metadata is lost when items are evicted!

- Ghost lists (non-resident items) extend your history
  - Maintain longer LRU queues but with a fixed number of value-less items, ala S2LRU
  - Access to these items is still a miss, but we learn the stack distance!
  - Keys should be significantly smaller than values to be worthwhile

LRU Variants (6)

What about pollution from scans?

- ARC (Mohda *et al.* IBM), Adaptive Replacement Cache
  - Maintains two LRUs of variable size, one for items seen *exactly* once, other for >1 hits
  - Uses ghost lists to extend history of both LRUs
  - Dynamically sizes the LRUs depending on scan prominence
  - Historic concerns about patents
- LIRS (Jiang and Zhang), Low Inter-Reference Recency Set
  - Two LRU queues of fixed sizes with low reuse distance (LIR) and high reuse distance (HIR)
  - Uses ghost lists for high reuse distance items
  - HIR item i switched to LIR if reuse distance of i smaller than largest within LRU
- CLOCK-variants of both: Clock-Pro (LIRS), CAR (ARC)

Birth of Virtual Memory

- Replacement algorithms became interesting first in context of virtual memory
- Atlas: first known system with virtual memory (paper: 1962).
  - Must read paper: Kilburn, T. *et al.* One-level storage system. IRE Transactions EC-11 (Apr. 1962).
  - Invented a "learning" algorithm to determine which pages should sit on drums versus in core. Had "use" bits.
  - Corner cases reportedly led to catastrophic thrashing.

Figure: Atlas console

Picture source:
http://www.computerconservationsociety.org/resurrection/res69.htm

Let's be Experimental?

- People wanted programming benefits of virtual memory but ...
  - Well known in the 1960s: virtual memory eviction algorithm performance is workload-dependent and non-linear.
  - Major efforts to understand thrashing behavior and to model cache performance goes back to 1960s.
- In the mid 1960s, the largest project of its kind to date was launched at IBM to study eviction algorithms because virtual memory systems were suffering from unexplained thrashing in the field.
- IBM, being more conservative, hadn't wanted to ship virtual memory based systems until thrashing was better understood or mitigated.

Let's be Experimental?

- Famous works of Belady *et al.* followed*.
  - Introduced Belady's MIN or (OPT) algorithm.
    - ★ Evict the element used farthest in the future.
    - ★ Intuitive, but proof subtle: show that the greedy (prescient) strategy stays ahead of the true OPT.
  - Studied relationship between block size and miss ratios.
  - Explored motivation of using access history in replacement decisions.
  - Described motivation for LRU.

*L. A. Belady, "A Study of Replacement Algorithms for a Virtual-storage Computer," IBM Syst. J., vol. 5, no. 2, pp. 78–101, Jun. 1966.

Belady's Motivation for LRU–1966

one hopes to improve replacement decisions by anticipating future references on the basis of previous references. We try to improve the techniques of FIFO, which selects the blocks according to their age in memory, but does not provide information about the distribution of references.

The idea is to dynamically order the blocks in memory according to the sequence of references to them. When a replacement becomes necessary, we replace the block to which reference has not been made for the longest time. We hope that the fact that this block has not been needed during the recent past indicates that it will not be referenced in the near future. This is a significant refinement relative to FIFO, since now frequency of use rather than stay in memory is the decisive factor.

Figure: Excerpt from Belady's 1966 seminal paper.

Exactly who invented this stuff?

Peter J. Denning on the Origin of Various Replacement Algorithms

"**I am certain there is no one person who originated LRU.** It was **an idea that was "in the air"** at the time, as many OS designers and architects were considering virtual memory and other kinds of cache. The first I remember it in a published paper was Belady's 1966 study of paging algorithms. But before that paper was out, as a newbie grad student at MIT, I was pursuing a suggestion of my advisor to become an expert on virtual memory and help them design the Multics virtual memory. The 1961 paper on Atlas was the starting point and brought the issue of replacement algorithm to the fore. Others began inquiring into paging algorithms and **when I began my investigation in 1965 I found MIT professors talking about FIFO, RAND, LRU, and others. These seemed to be the obvious ones occurring to people.**"*

*Private communication of Irfan Ahmad with Peter J. Denning, Jan. 2018

Let's be Analytical?

- Throughout the 1960s, cache modeling was done by experiments with different sizes and parameters. This was very time consuming.
- In 1970, Mattson, *et al.* provided a one-pass (online) algorithm to model cache behavior for a certain class of algorithms known as *stack algorithms* that exhibit the *subset inclusion* property.

Subset inclusion property

The subset inclusion property is satisfied iff the specific set of pages in a cache C_k of size k is always a subset of the pages in a cache C_{k+1} of size $k+1$ on the same trace.

- The property implies a unique eviction order of cache elements, forming a stack.
  - E.g. C_k would evict the unique element $\{C_k - C_{k-1}\}$.
- Breakthrough in modeling cache behavior.

Stack distance

Examples of stack algorithms include LRU, LFU and OPT. But many high-performance algorithms are non-stack: ARC, LIRS, etc.

Reuse distance

The *reuse distance* of an item x is the number of **unique** references since that item was last requested (or $+\infty$ if never, a cold miss)

- What are the reuse distances for the references *a b a c c a*?

Proof.

Exact characterization of LRU. For LRU, stack distance exactly equals reuse distance. □

Performance of LRU at *any* size k can be characterized by tracking reuse distances.

Stack distance

Examples of stack algorithms include LRU, LFU and OPT. But many high-performance algorithms are non-stack: ARC, LIRS, etc.

Reuse distance

The *reuse distance* of an item x is the number of **unique** references since that item was last requested (or $+\infty$ if never, a cold miss)

- What are the reuse distances for the references *a b a c c a*?
  $\infty \ \infty \ 1 \ \infty \ 0 \ 1$

Proof.

Exact characterization of LRU. For LRU, stack distance exactly equals reuse distance. □

Performance of LRU at *any* size k can be characterized by tracking reuse distances.

Calculating LRU stack distance

Extensive literature on this problem.

- Prominent problem in programming languages and compiler communities.
- Compilers need to decide which variables are afforded registers, whether it pays to allocate a stack frame, *etc.*[*]
- Unlike us, the compiler can take multiple passes over the data.

Modeling non-stack algorithms?

No one-pass algorithm has been discovered for precisely modeling a deterministic non-stack eviction scheme.

Theorem

Conjecture: *no such general-purpose, exact algorithm exists for the class of deterministic non-stack eviction algorithms.*

[*] Ding *et al.* PLDI 2003

Caches Critical, Yet Management Problems Loom

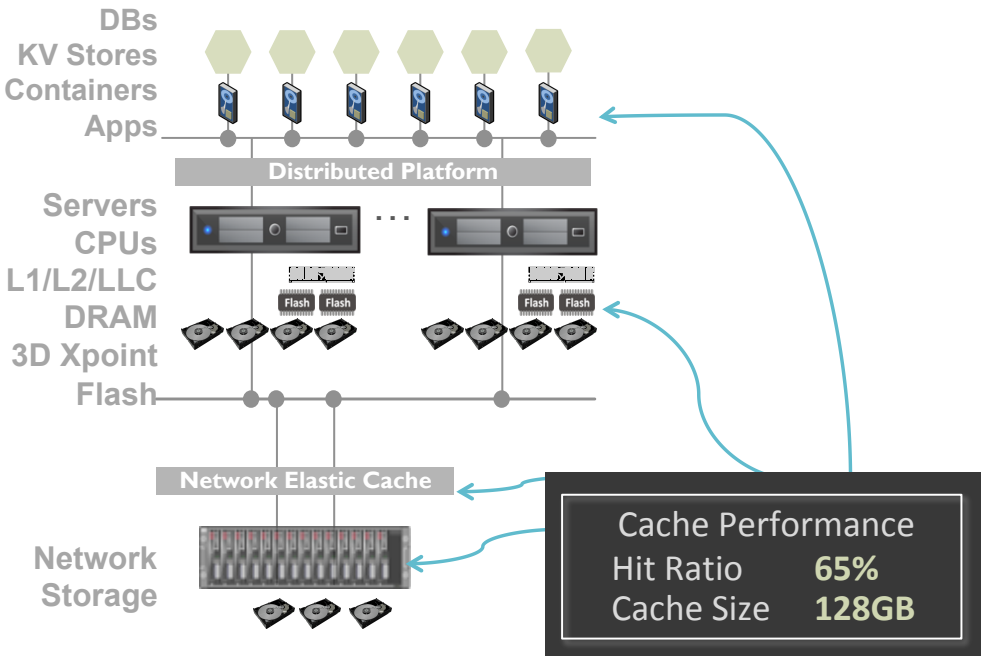


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

Caches Critical, Yet Management Problems Loom

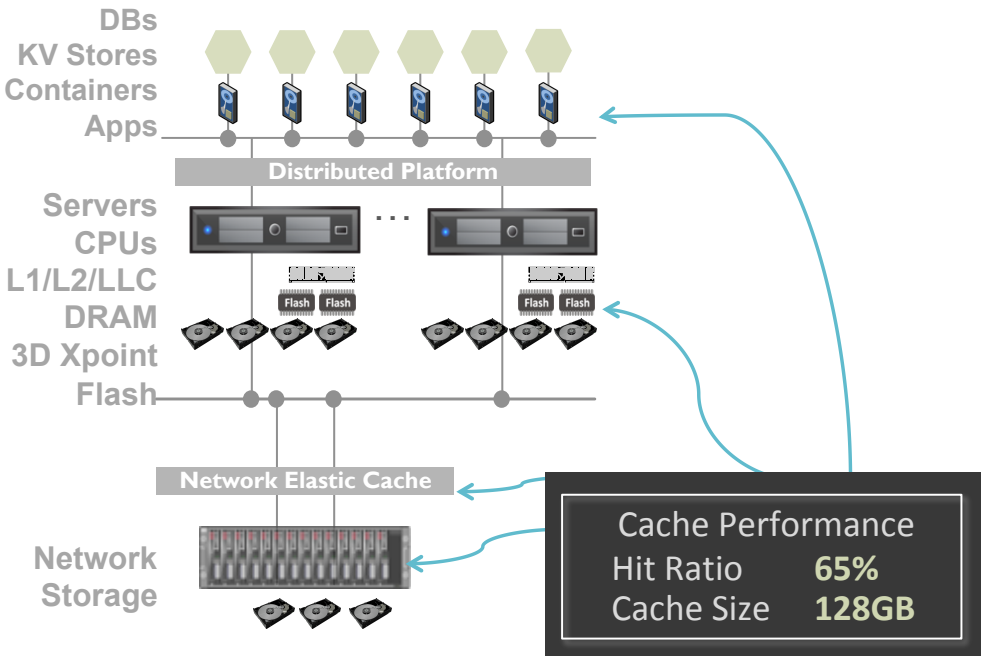


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?

Caches Critical, Yet Management Problems Loom

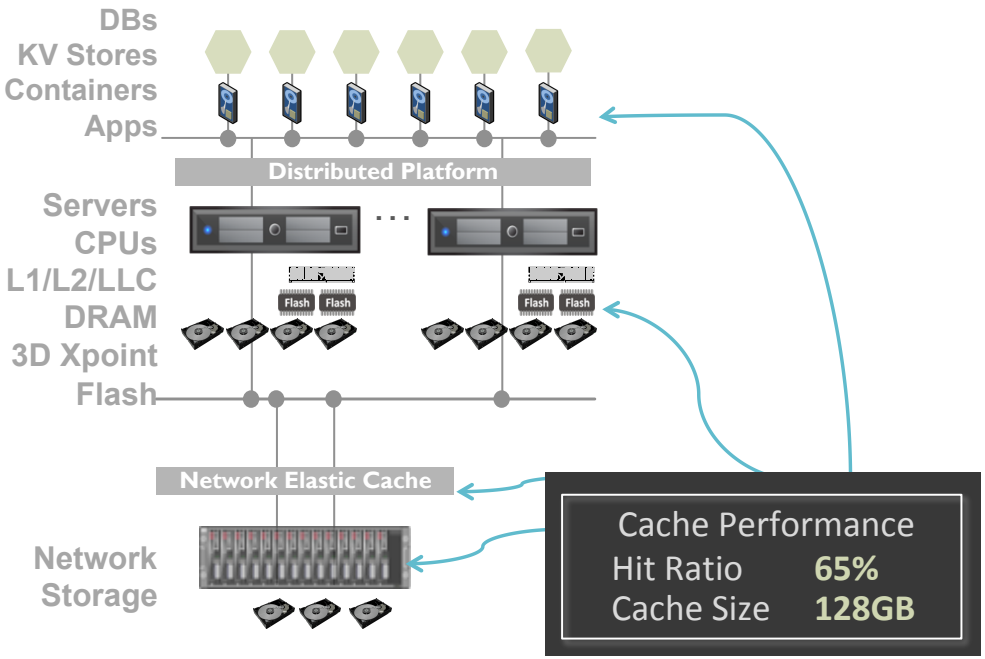


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?
- How much Cache for App A vs. B vs. ...?

Caches Critical, Yet Management Problems Loom

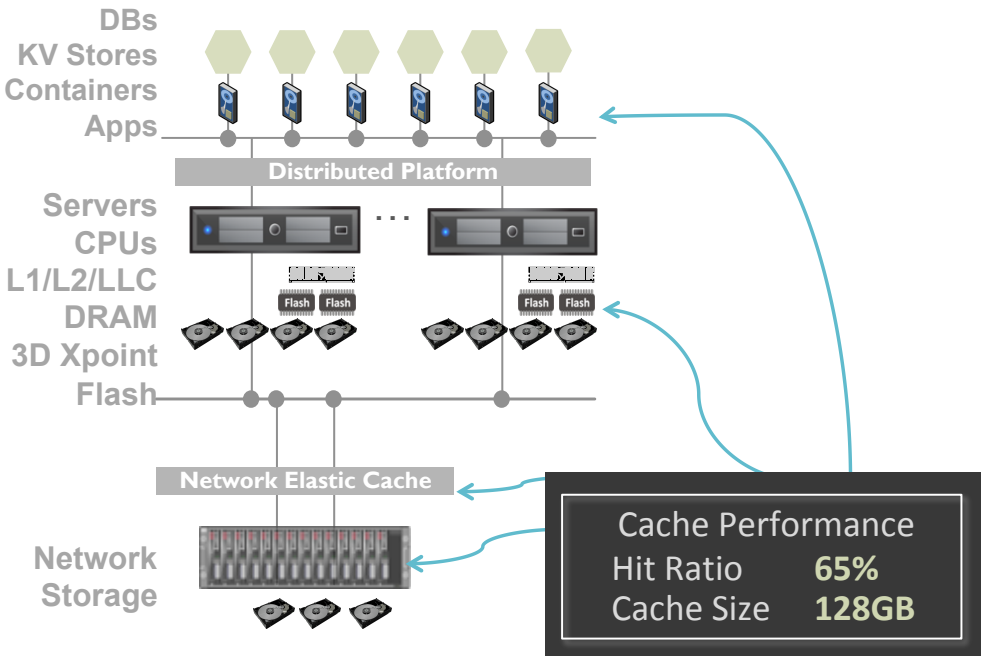


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?
- How much Cache for App A vs. B vs. ...?
- What happens if I add / remove DRAM?

Caches Critical, Yet Management Problems Loom

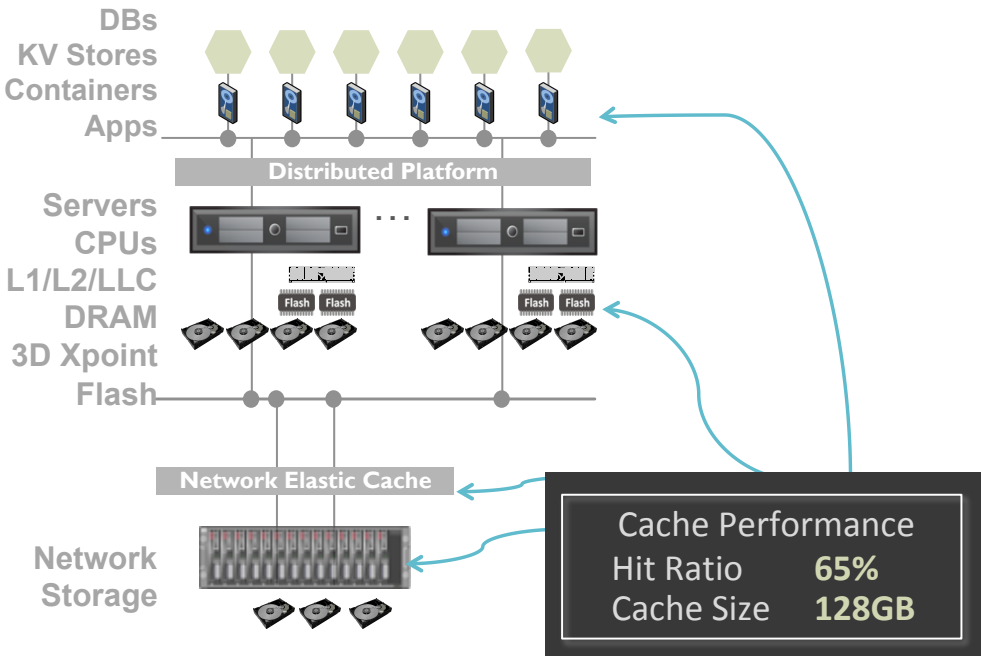


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?
- How much Cache for App *A* vs. *B* vs. ...?
- What happens if I add / remove DRAM?
- How much DRAM versus Flash?

Caches Critical, Yet Management Problems Loom

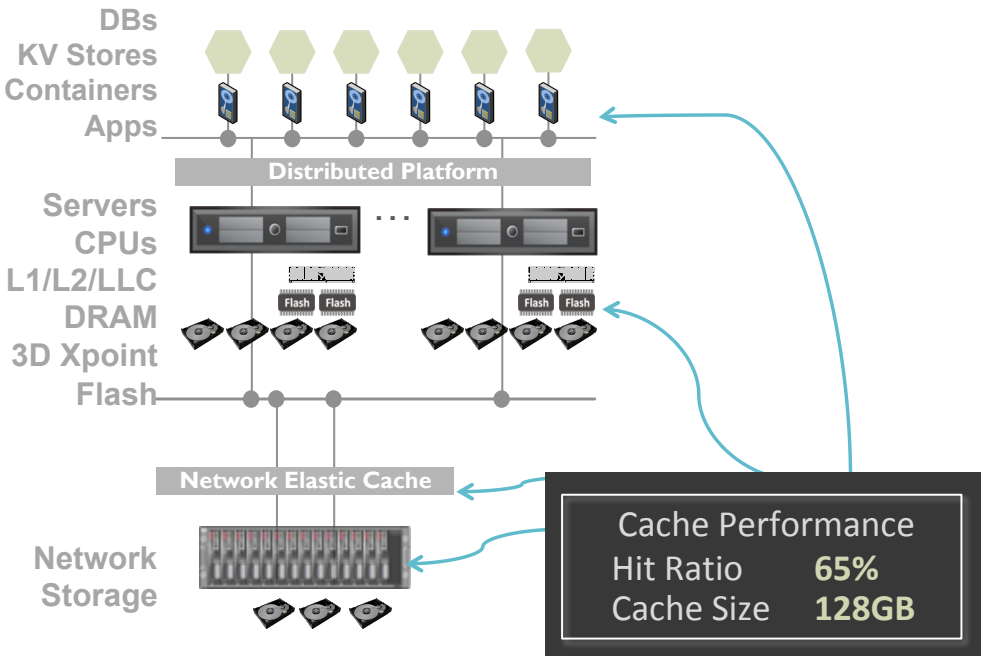


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?
- How much Cache for App A vs. B vs. ...?
- What happens if I add / remove DRAM?
- How much DRAM versus Flash?
- How to achieve 99%ile latency of X μs?

Caches Critical, Yet Management Problems Loom

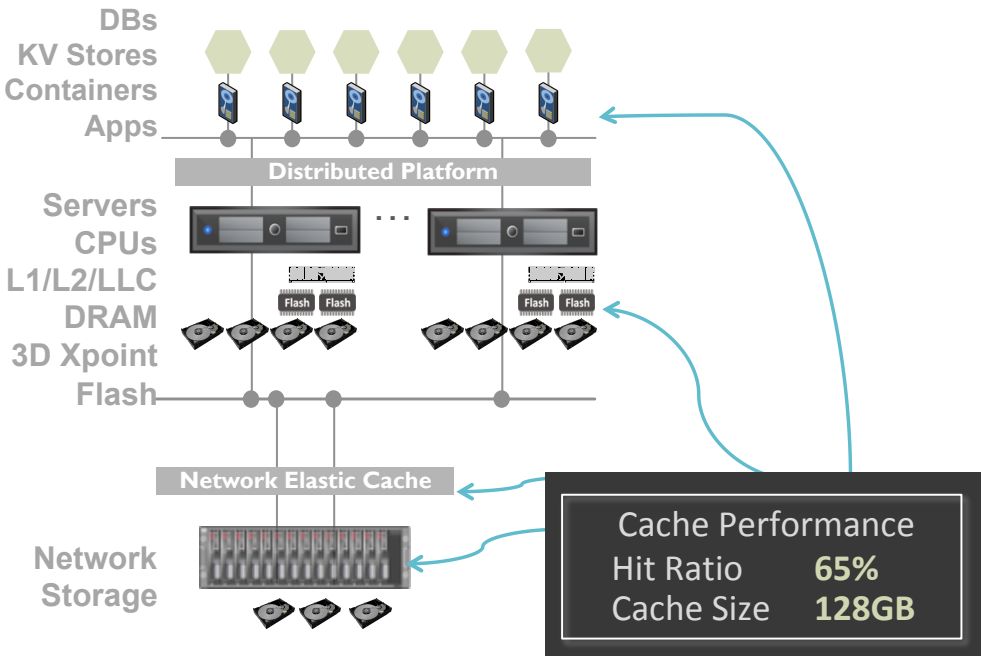


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?
- How much Cache for App A vs. B vs. ...?
- What happens if I add / remove DRAM?
- How much DRAM versus Flash?
- How to achieve 99%ile latency of X μs?
- What if I add / remove workloads?

Caches Critical, Yet Management Problems Loom

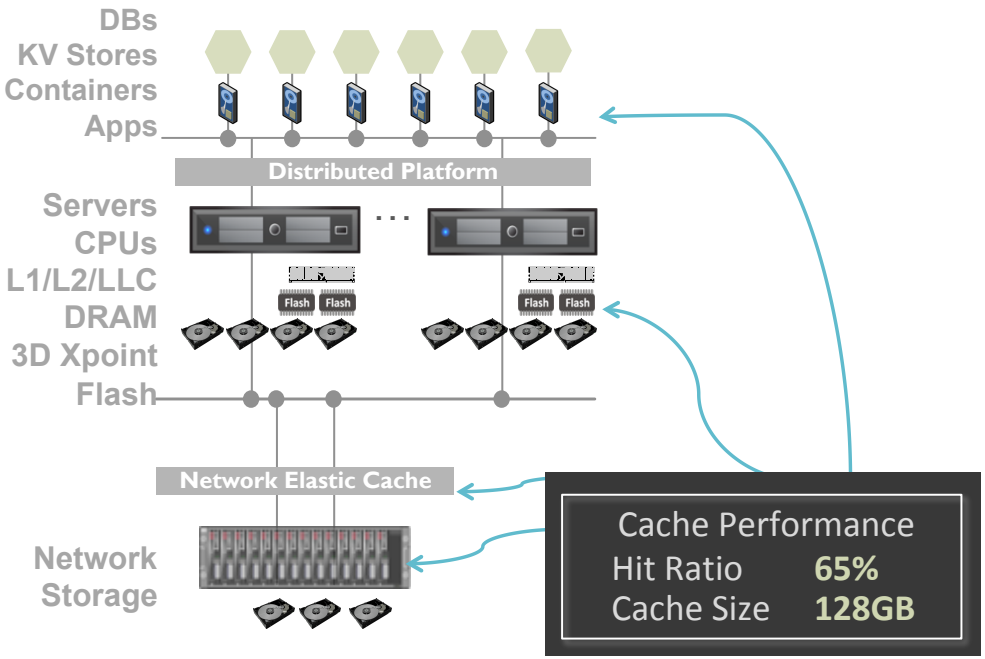


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?
- How much Cache for App A vs. B vs. ...?
- What happens if I add / remove DRAM?
- How much DRAM versus Flash?
- How to achieve 99%ile latency of X μs?
- What if I add / remove workloads?
- What if I change parameters?

Caches Critical, Yet Management Problems Loom

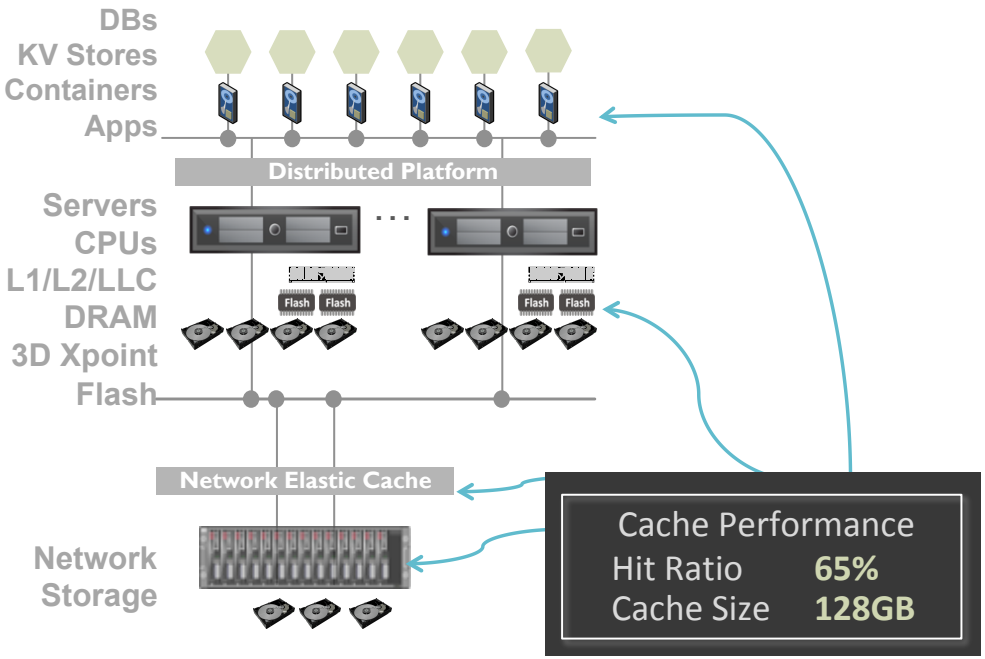


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?
- How much Cache for App A vs. B vs. ...?
- What happens if I add / remove DRAM?
- How much DRAM versus Flash?
- How to achieve 99%ile latency of X μs?
- What if I add / remove workloads?
- What if I change parameters?
- Is there cache thrashing / pollution?

Caches Critical, Yet Management Problems Loom

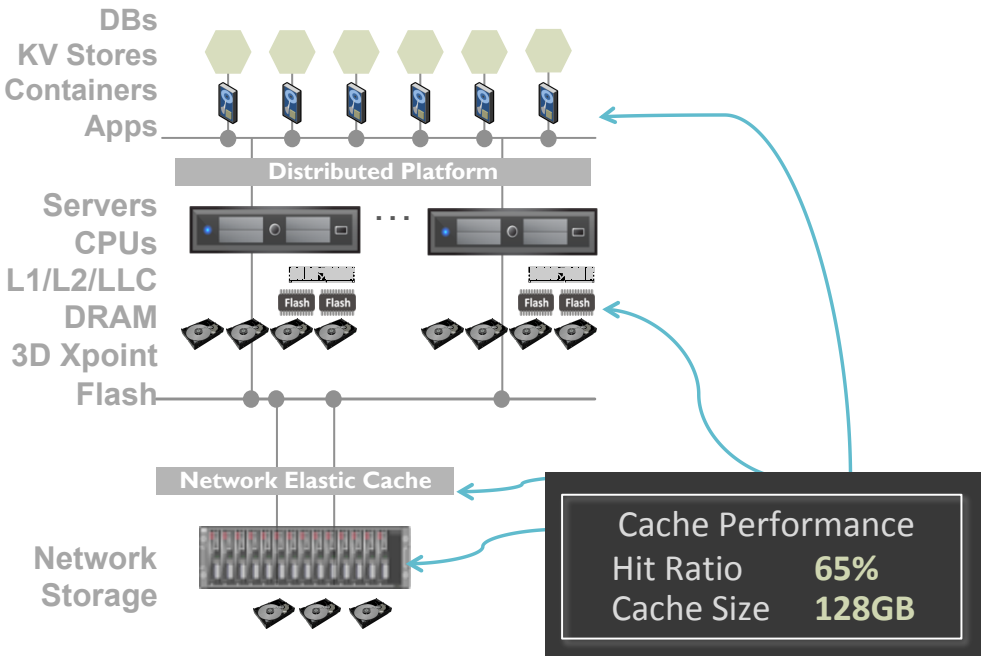


Figure: Caches are critical to the performance of everything from watches to datacenters and global distributed systems.

- Is this performance good? Can it be improved?
- How much Cache for App A vs. B vs. ...?
- What happens if I add / remove DRAM?
- How much DRAM versus Flash?
- How to achieve 99%ile latency of X μs?
- What if I add / remove workloads?
- What if I change parameters?
- Is there cache thrashing / pollution?
- How long to warm up the cache?

Cache modeling

Core strategy for realizing these applications

Cache Utility Curves

Model how performance varies as a function of provided resources (cache space). The performance of stack algorithms can modeled by tracking stack distances.

We first define the "miss rate curve" (MRC), the most common cache utility curve, more precisely.
Next up:

- How should we generate MRCs, algorithmically?
- What are the appealing properties?
- What happens on modern workloads?
- What about non-stack algorithms?

We then move on to applications.

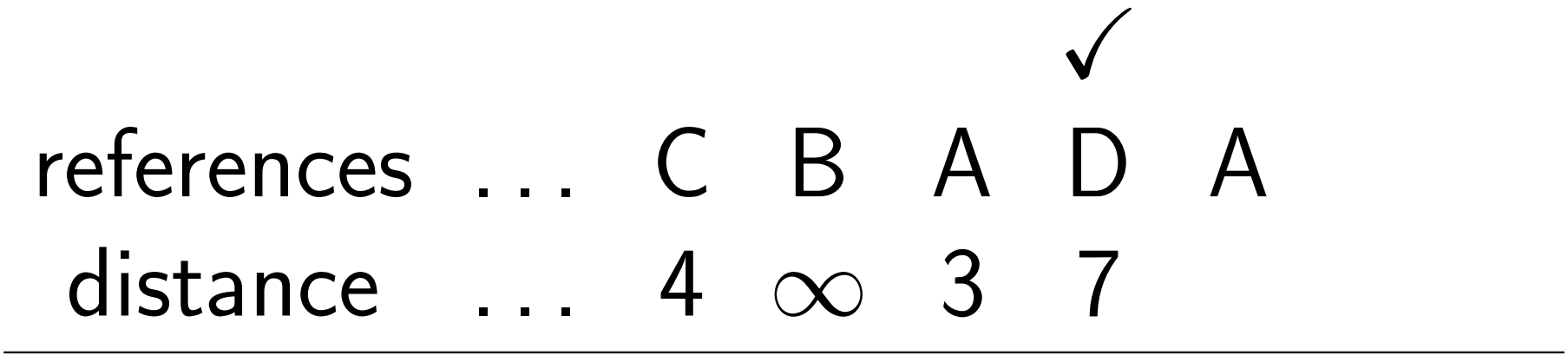
Mattson's Stack Distance Algorithm Example

references	...	C	B	A	D
distance	...	4	∞	3	7

Mattson's Stack Distance Algorithm Example

references	...	C	B	A	D	A
distance	...	4	∞	3	7	

Mattson's Stack Distance Algorithm Example



Mattson's Stack Distance Algorithm Example

references	...	C	B	A	D	A
distance	...	4	∞	3	7	1

Mattson's Stack Distance Algorithm Example

references	...	C	B	A	D	A	B
distance	...	4	∞	3	7	1	

Mattson's Stack Distance Algorithm Example

						✓	
references	...	C	B	A	D	A	B
distance	...	4	∞	3	7	1	

Mattson's Stack Distance Algorithm Example

					✓	✓	
references	...	C	B	A	D	A	B
distance	...	4	∞	3	7	1	

Mattson's Stack Distance Algorithm Example

				X	✓	✓	
references	...	C	B	A	D	A	B
distance	...	4	∞	3	7	1	

Mattson's Stack Distance Algorithm Example

references	...	C	B	A	D	A	B
distance	...	4	∞	3	7	1	2

Mattson's Stack Distance Algorithm Example

references	...	C	B	A	D	A	B	C
distance	...	4	∞	3	7	1	2	

Mattson's Stack Distance Algorithm Example

							✓	
references	...	C	B	A	D	A	B	C
distance	...	4	∞	3	7	1	2	

Mattson's Stack Distance Algorithm Example

						✓	✓	
references	...	C	B	A	D	A	B	C
distance	...	4	∞	3	7	1	2	

Mattson's Stack Distance Algorithm Example

					✓	✓	✓	
references	...	C	B	A	D	A	B	C
distance	...	4	∞	3	7	1	2	

Mattson's Stack Distance Algorithm Example

				X	✓	✓	✓	
references	...	C	B	A	D	A	B	C
distance	...	4	∞	3	7	1	2	

Mattson's Stack Distance Algorithm Example

			X	X	✓	✓	✓	
references	...	C	B	A	D	A	B	C
distance	...	4	∞	3	7	1	2	

Mattson's Stack Distance Algorithm Example

references	...	C	B	A	D	A	B	C
distance	...	4	∞	3	7	1	2	3

Mattson's Stack Distance Algorithm

- $M \leftarrow |\text{unique references}|$, $N \leftarrow |\text{all references}|$
- If you track references on a list
  - Calculate unique refs since last access
  - Distance from top of priority-function-ordered stack (e.g. LRU)
  - Hit if distance $<$ cache size, else miss
  - Cost: $O(M)$ space, $O(NM)$ compute
- If you store and retrieve stack distances using a tree.
  - Reduces cost to $O(N)$ space, $O(N \log N)$ compute*.
- If your tree stores unique references
  - Reduces cost to $O(M)$ space, $O(N \log M)$ compute†.
- $O(N \log M)$ still too expensive to run online, real-time

*B. T. Bennett and V. J. Kruskal. LRU stack processing. IBM Journal for Research and Development, July 1975.

†F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Perform. Eval. 3, 2 (1983)

Modeling with Miss Ratio Curves (MRCs)

- Learn the performance model of applications and cache.
- Predict the performance of workload as $f(\text{cache size}, \text{params})$

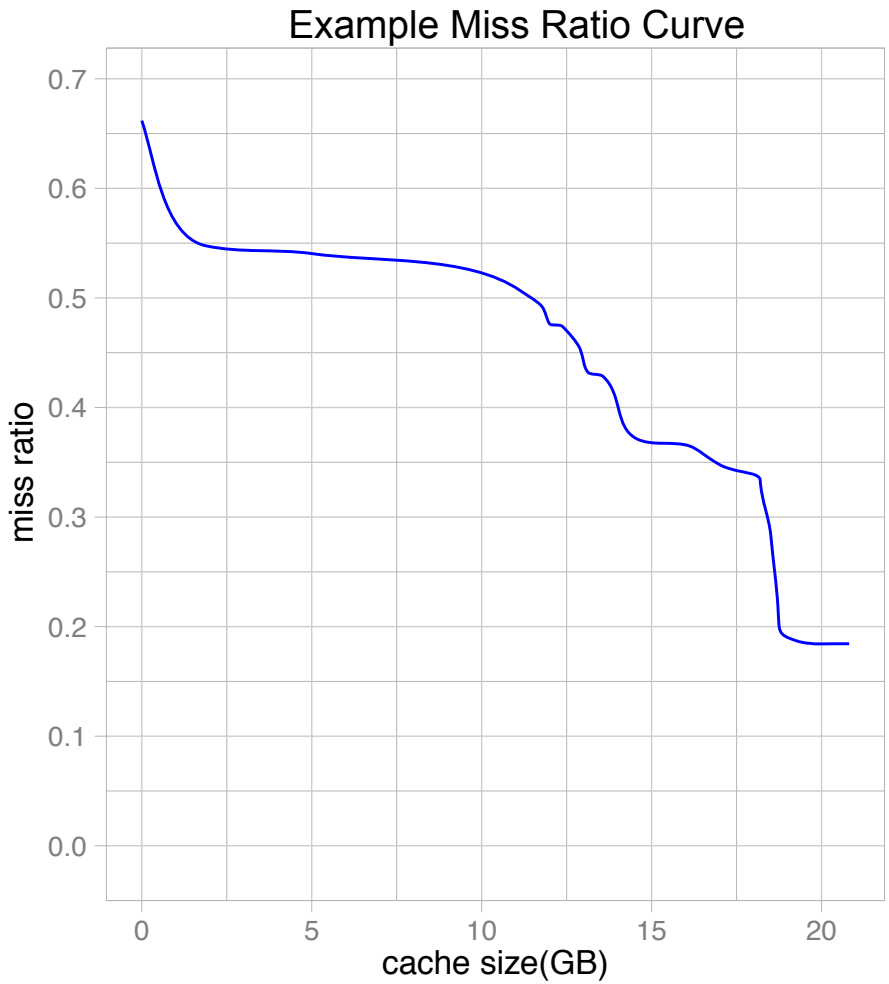


Figure: Example MRC*.

*Waldspurger *et al*, Efficient MRC Construction with SHARDS, USENIX FAST '15

Understanding Cache Models

- Models help decide useful increments of change.
- In this example, no significant benefit despite an 8× increase in budget.
- Many real-life workloads naturally have MRCs with staircase patterns. Why?

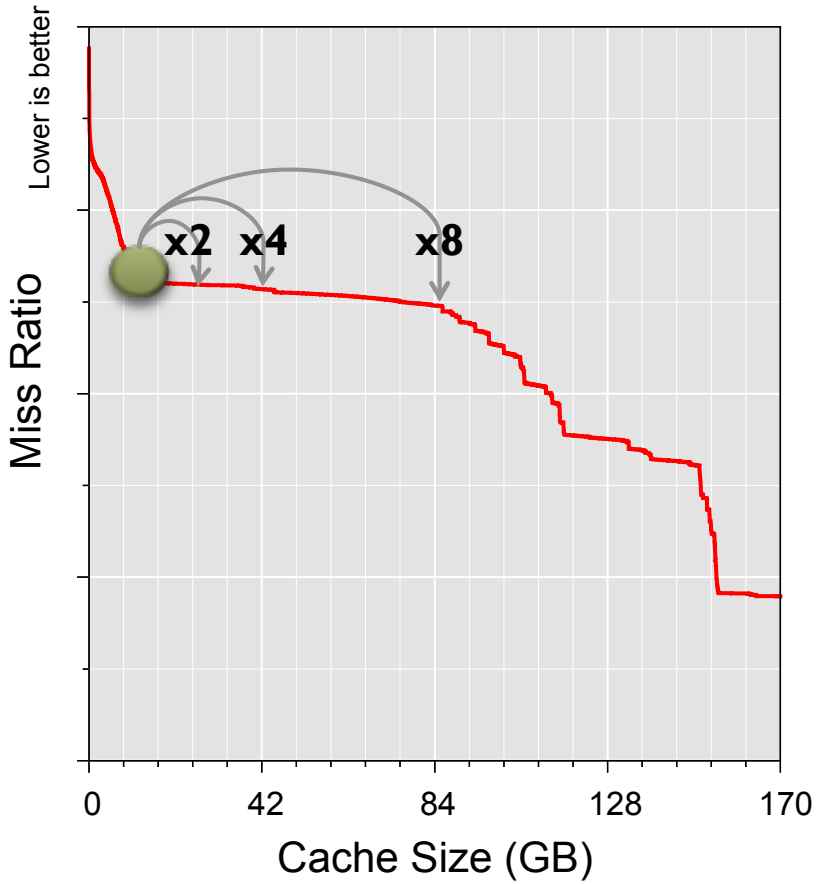


Figure: A real workload MRC illustrating capacity allocations that yield negligible benefits despite 3 orders of magnitude increase in cost.

Understanding Cache Models

- Models help decide useful increments of change.
- In this example, no significant benefit despite an 8× increase in budget.
- Many real-life workloads naturally have MRCs with staircase patterns. Why?
  - Working sets of different sizes.
  - Might also represent different time scales.
  - *e.g.* a diurnal vs weekly vs monthly working set sizes.

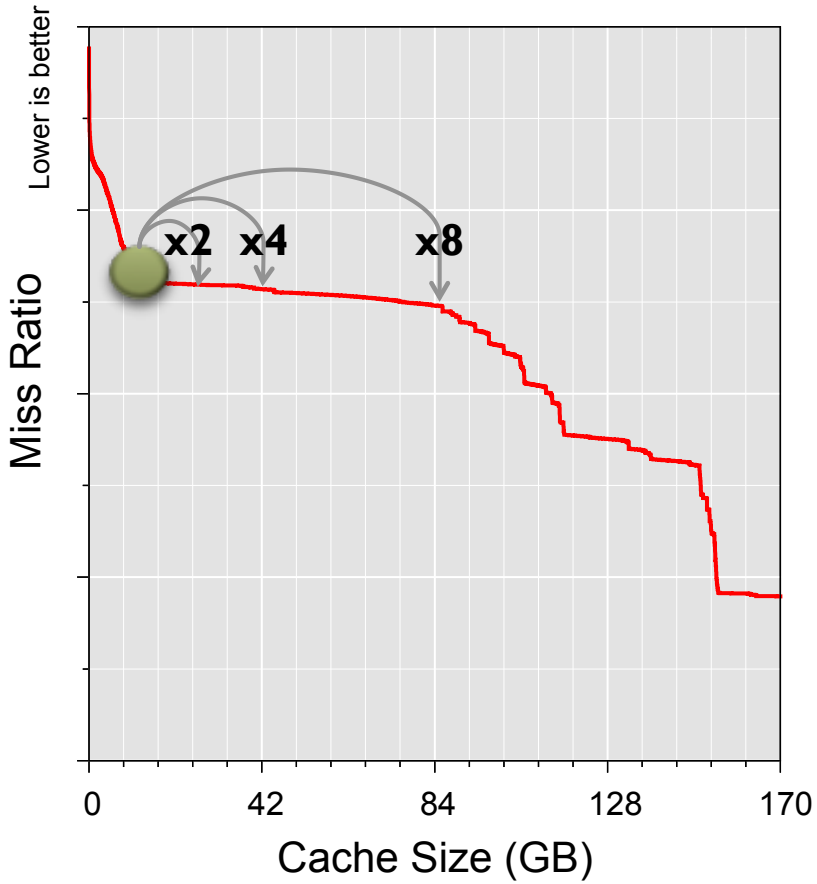


Figure: A real workload MRC illustrating capacity allocations that yield negligible benefits despite 3 orders of magnitude increase in cost.

Understanding Cache Models (2)

- Often, most operating points are highly inefficient.
- Look for cliff bottoms to improve efficiency.
- This cache is operating at the lowest ROI point; equivalent performance to 1/8 the budget.
- Arrows represent the efficient operating points which can be enumerated programmatically.

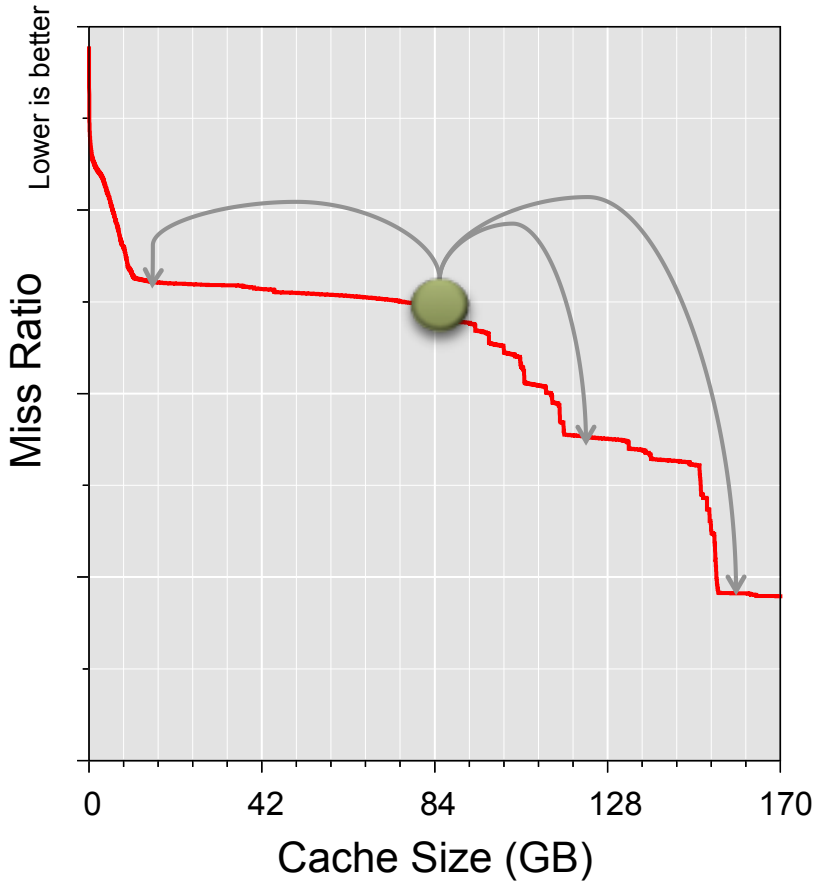


Figure: An MRC annotated with the efficient operating points for this workload's LRU cache.

MRCs from 1966 (Belady)

- Belady used MRCs to compare the impact of different memory cache block sizes (note the log scale).
  - Block size made a huge difference.
  - We see same for modern storage workloads.
  - Alas, few caches systematically exploit this.
- Cache block size is just one of many parameters.

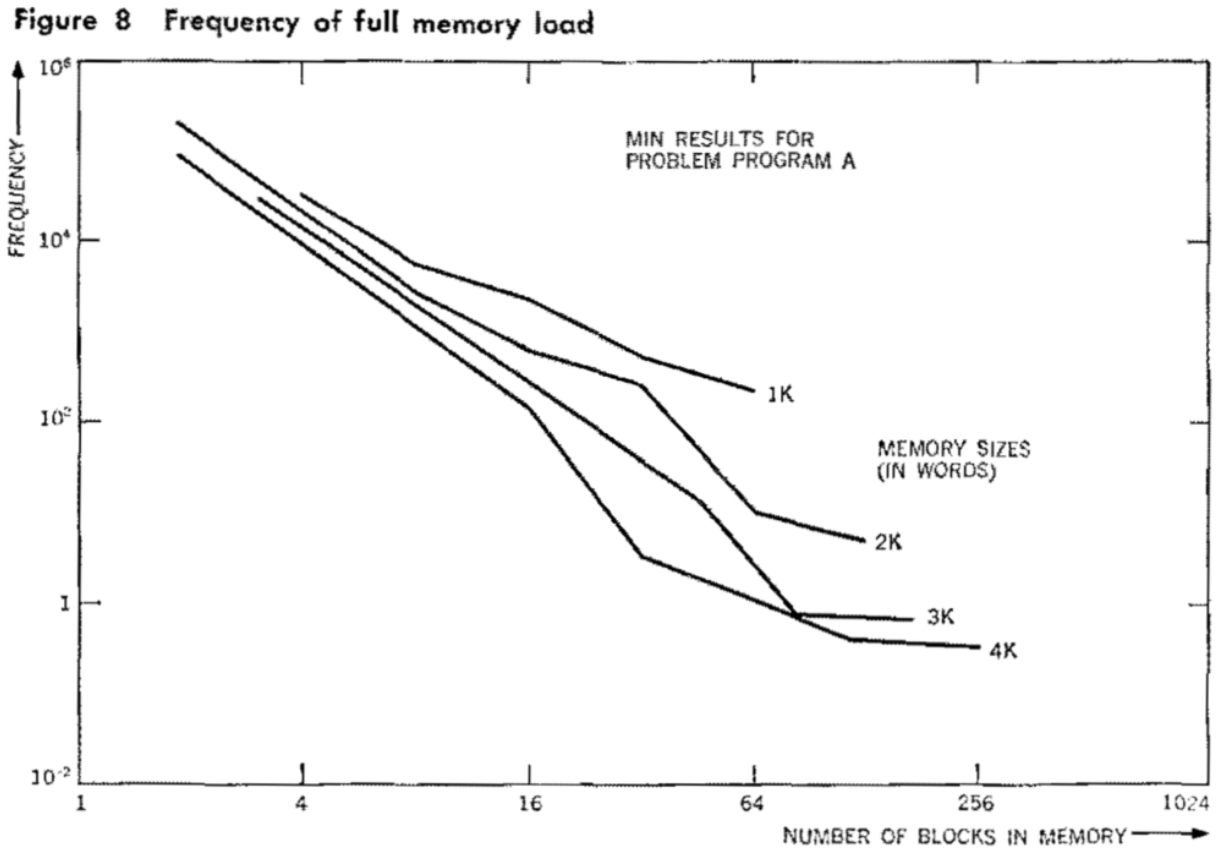


Figure: Miss Rate Curves of Different Block Sizes from Belady 1966

Modern-Day Storage Workload Block Size MRCs

- Single workload; LRU with different block sizes.
- Example of how MRCs can help predict performance under different policies.
- A self-adapting data infrastructure could measure and dynamically adjust to workload.

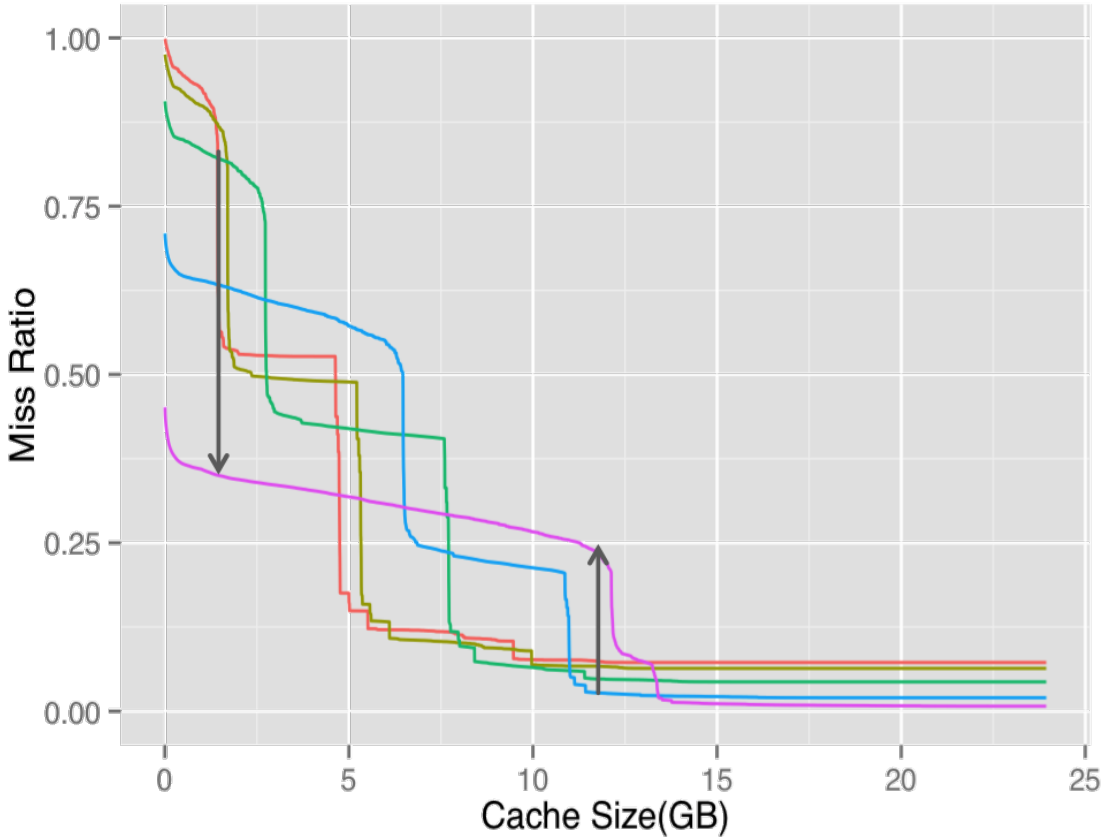


Figure: A real storage workload with LRU MRCs for different cache block sizes.

Modern-Day Storage Workload MRCs (LRU)

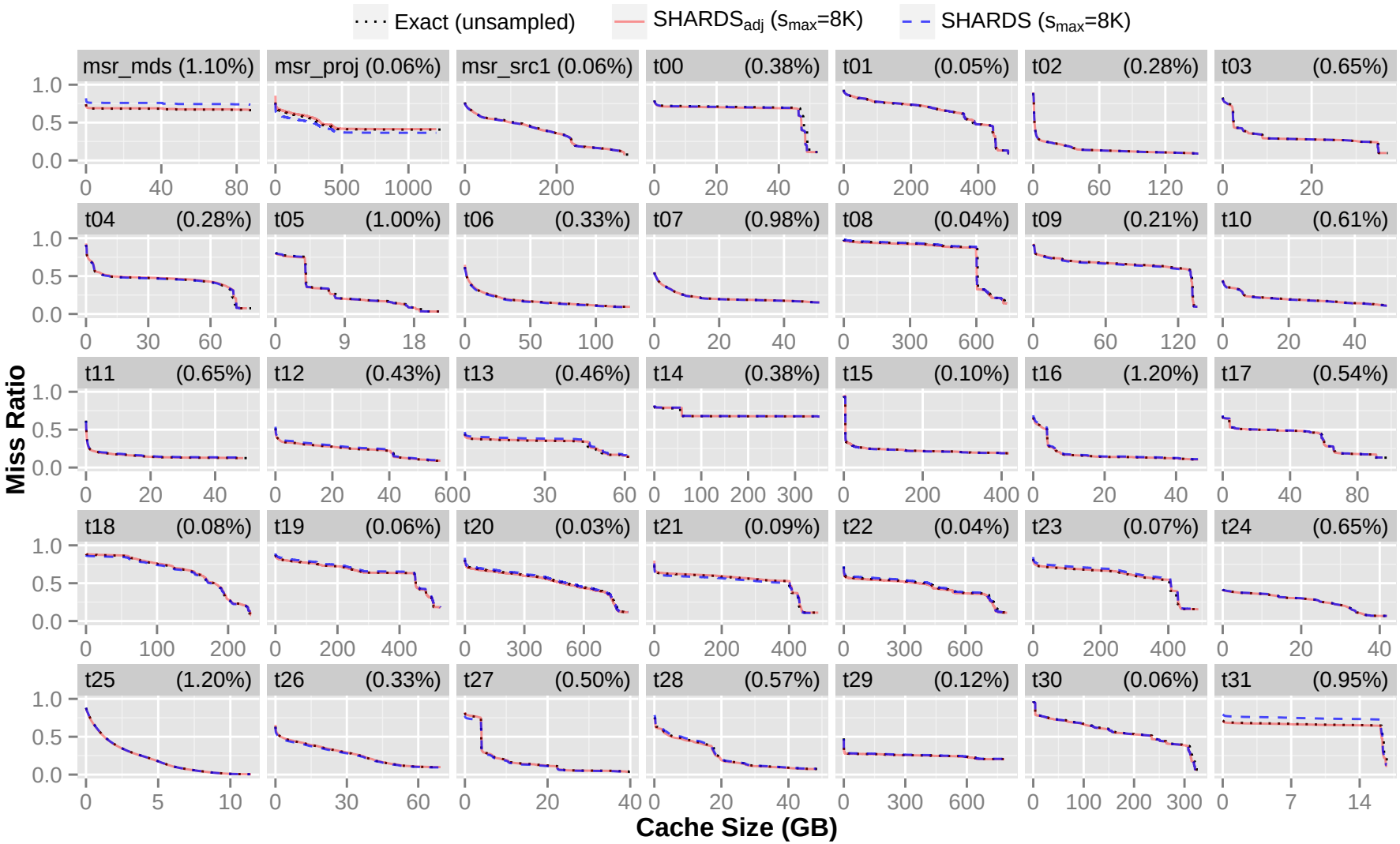


Figure: Real storage workload MRCs for LRU*.

*Waldspurger *et al*, Efficient MRC Construction with SHARDS, USENIX FAST '15

Modern-Day Storage Workload MRCs (ARC, LIRS, OPT)

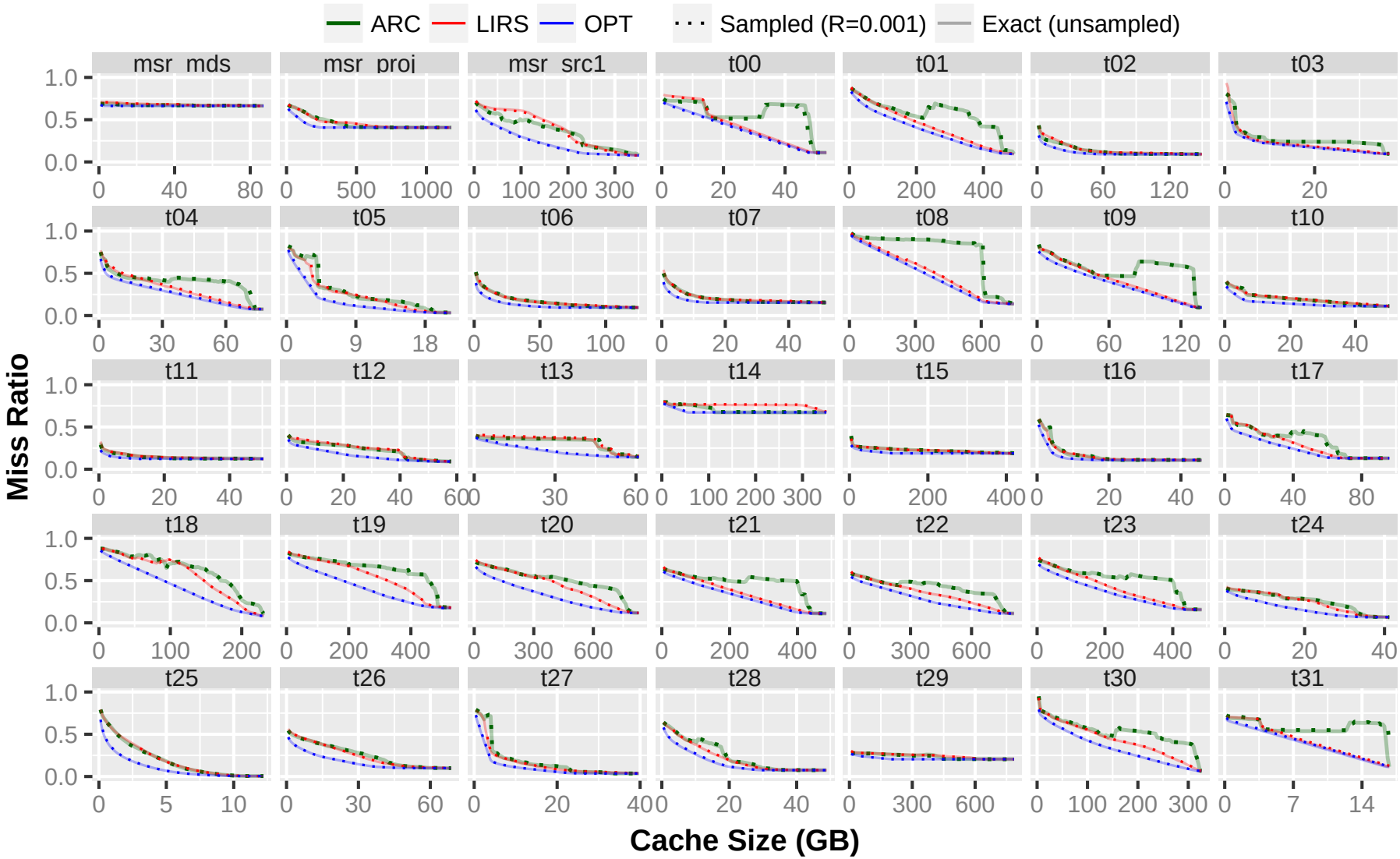


Figure: Real storage workload MRCs for different eviction algorithms*.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Closer Look at ARC and LIRS Examples

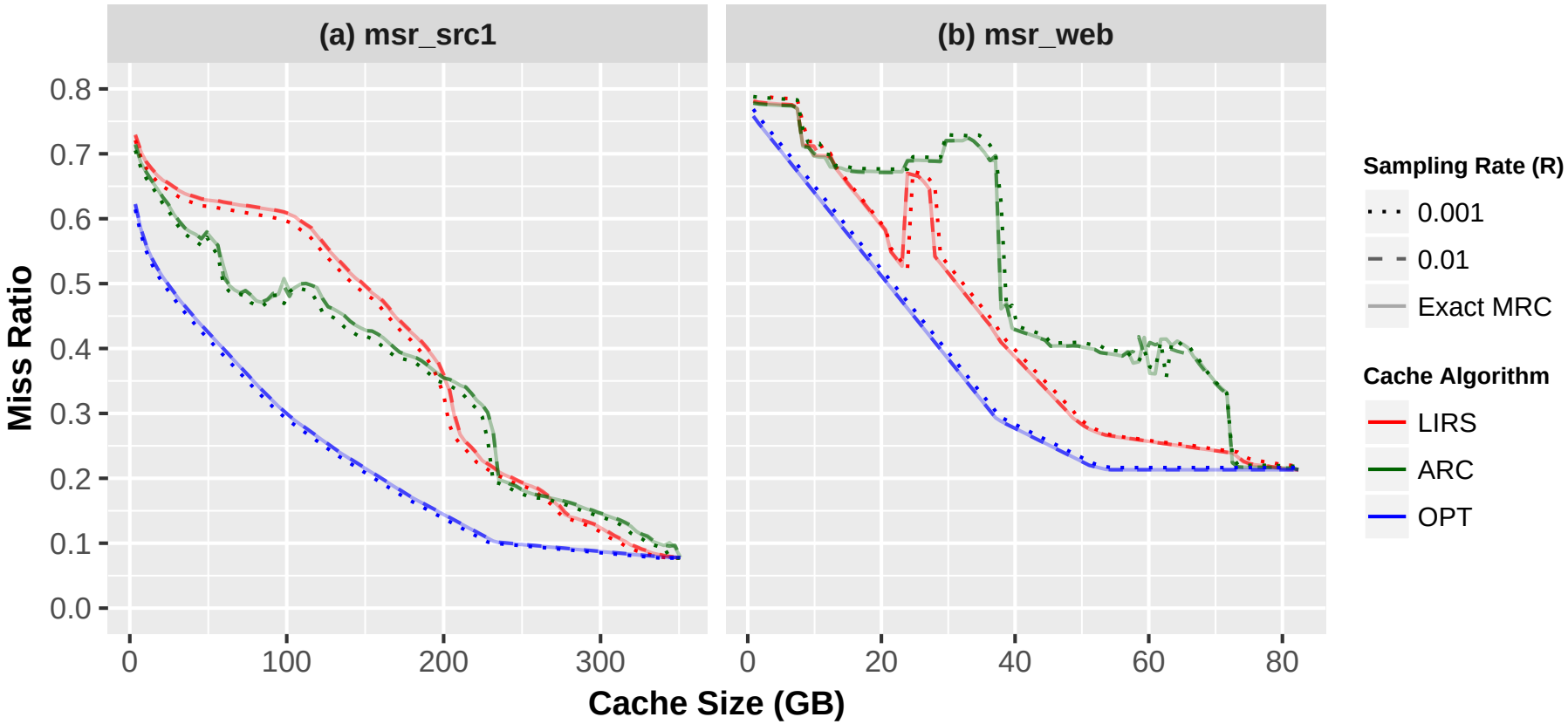


Figure: Real storage workload MRCs for different eviction algorithms*.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Cache Performance Observations

- Cache performance highly variable in several parameters, for example:
  - Cache size, cache block size
  - Cache eviction and prefetch policies
  - Write-through versus write-back
- Benefit varies widely by workload.
- **Opportunity:** dynamic cache management
  - Efficient sizing, allocation, and scheduling
  - Improve performance, isolation, QoS

Cache Modeling Research

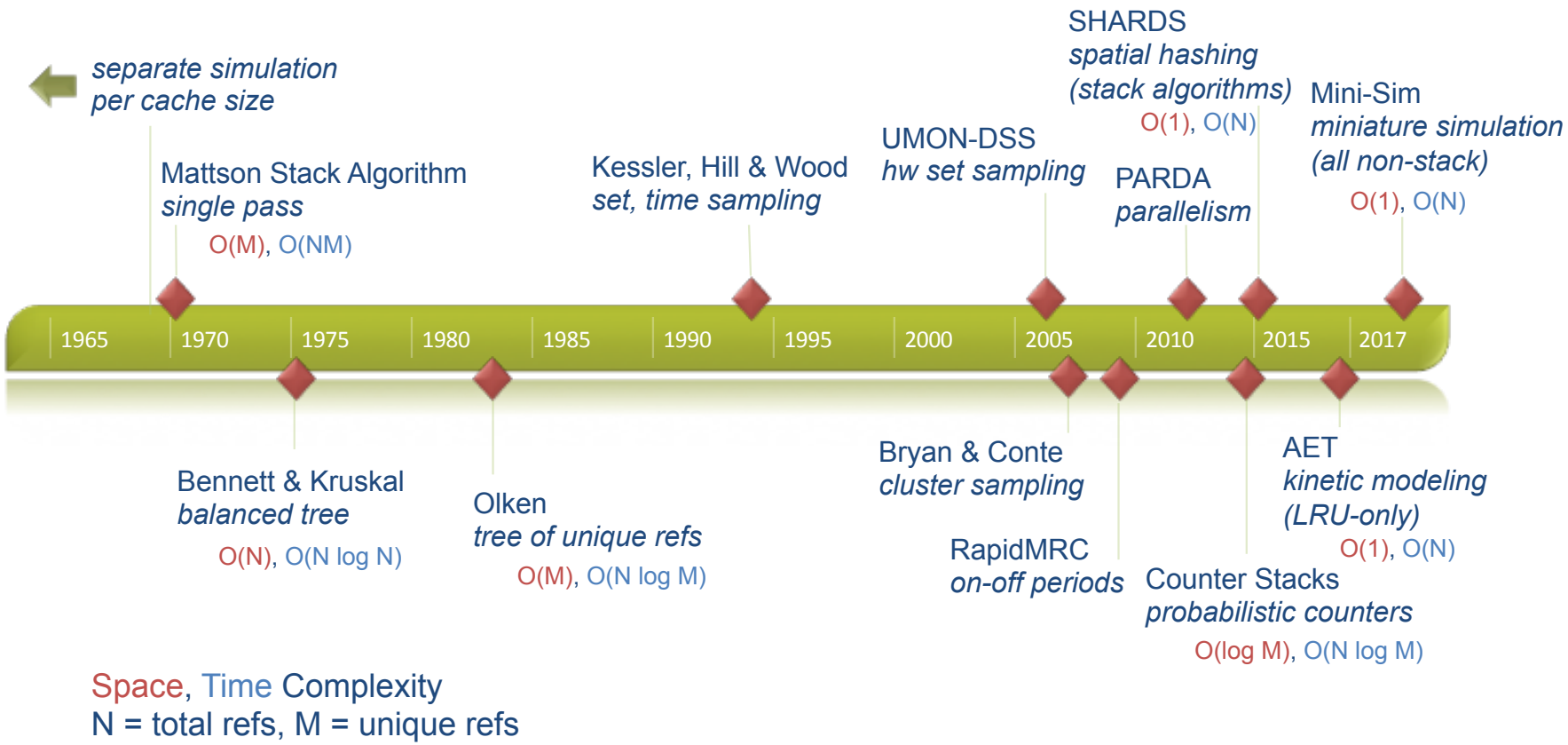


Figure: Historical time line for cache modeling literature*.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

MRC Construction Methods

	Exact	Approximate
Stack Algorithms	Mattson algorithm all sizes at once	MIMIR [SOCC '14], Counter Stacks [OSDI '14], SHARDS [FAST '15], AET [ATC '16], Victim Footprint [TACO '17]
Any Algorithm	separate simulation for each size	Miniature Simulation [ATC '17]

The appealing properties of MRCs

Property 1: Scaled down reference stream is self-similar

- Here, assume non-normalized miss *rate* curves.
- Assume *spatial sampling*, for instance sample based on `hash(key)`
- Temporal sampling generally performs worse.

Theorem (Sampling Theorem (LRU))

(Vigfusson) Let $m_\alpha(s)$ denote the MRC on a reference stream where each item is spatially sampled with probability α. The original MRC is $m_1(s) = m(s)$. Then

$$\mathbb{E}[m_\alpha(s)] \approx \alpha m\left(\frac{s}{\alpha}\right).$$

The fit depends exponentially on how "cliffy" the MRC is at point s.

- Sampling theorem argued generally by Kessler *et al.* (IEEE ToC 1994).

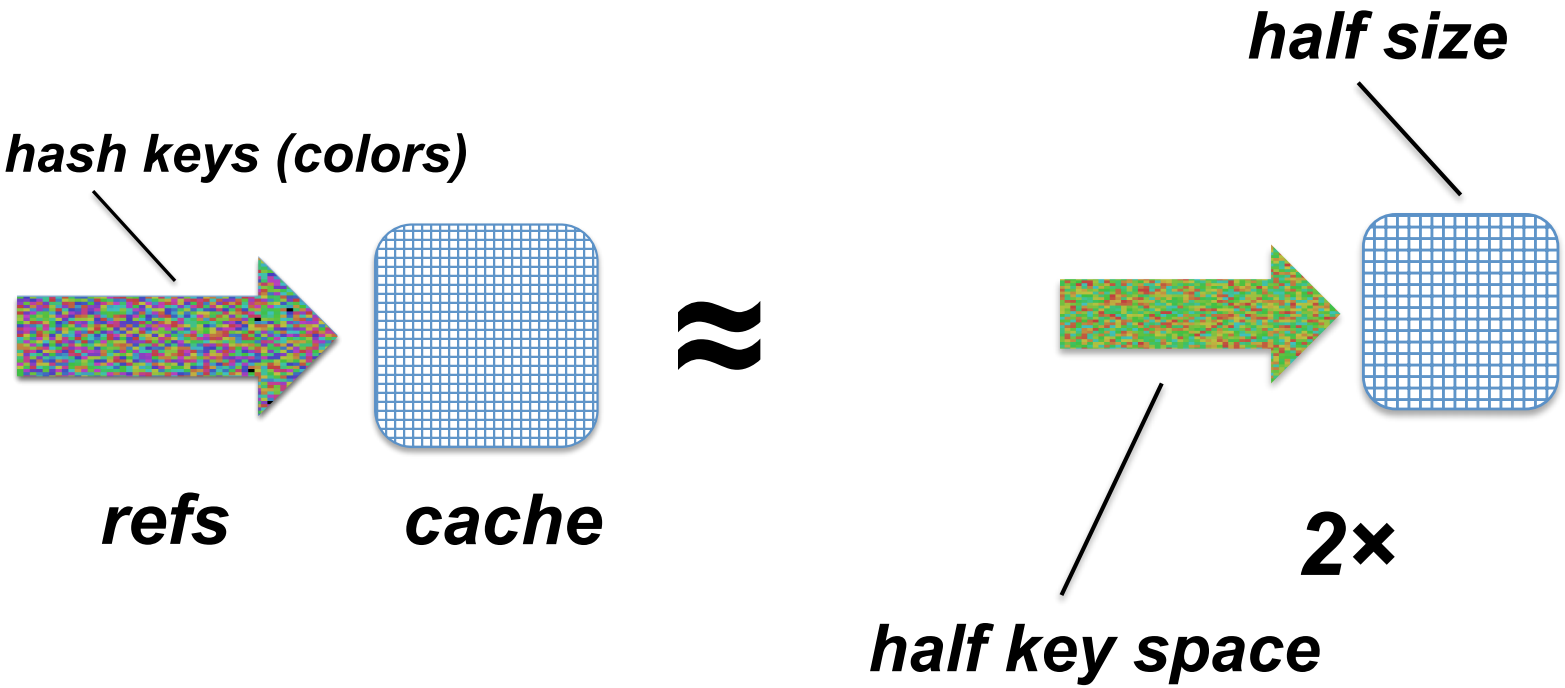
Takeaway: Spatial sampling scales the MRC down, both in terms of hit rate and cache size.

If Too Hard to Simulate, Mini-Simulate*

- Use the sampling theorem to simulate large cache using a tiny one
- Scale down reference stream, cache size
  - Random sampling based on `hash(key)`
  - Assumes statistical self-similarity (broad argument by Kessler *et al.* IEEE ToC 1994)
- Run unmodified algorithm
  - LRU, LIRS, ARC, 2Q, FIFO, OPT, ...
  - Track usual stats

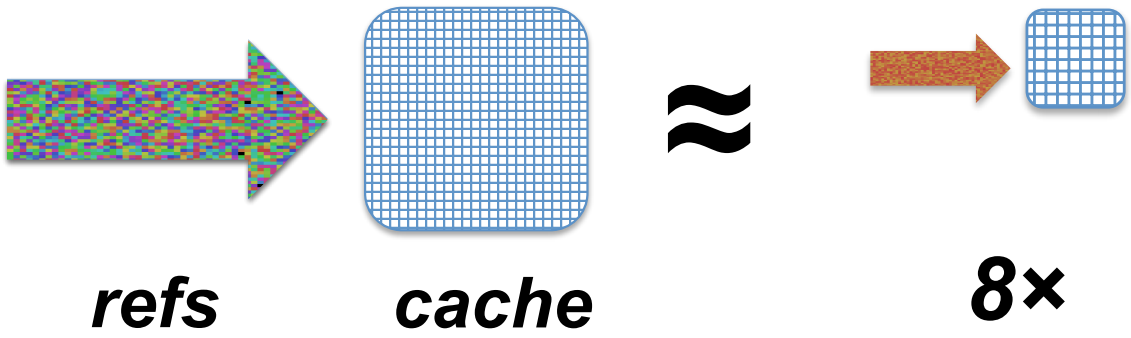
Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

If Too Hard to Simulate, Mini-Simulate*



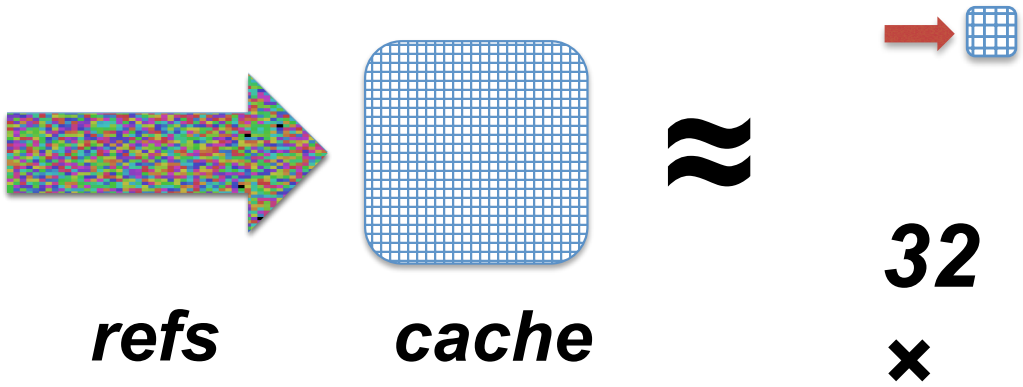
*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

If Too Hard to Simulate, Mini-Simulate*



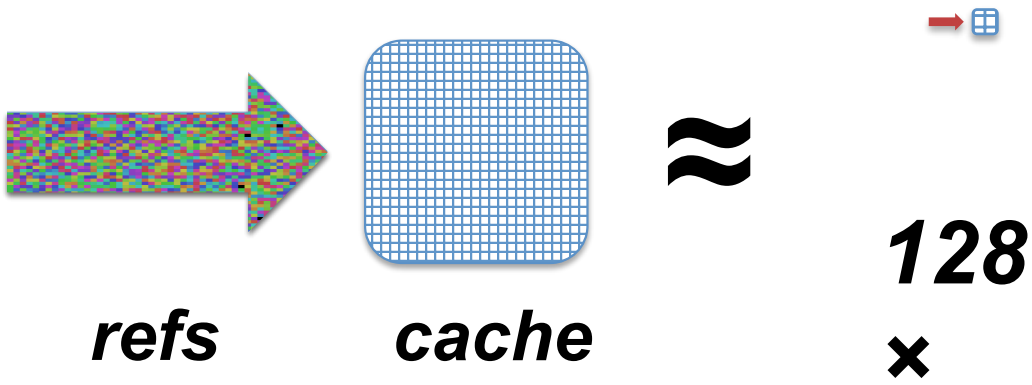
*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

If Too Hard to Simulate, Mini-Simulate*



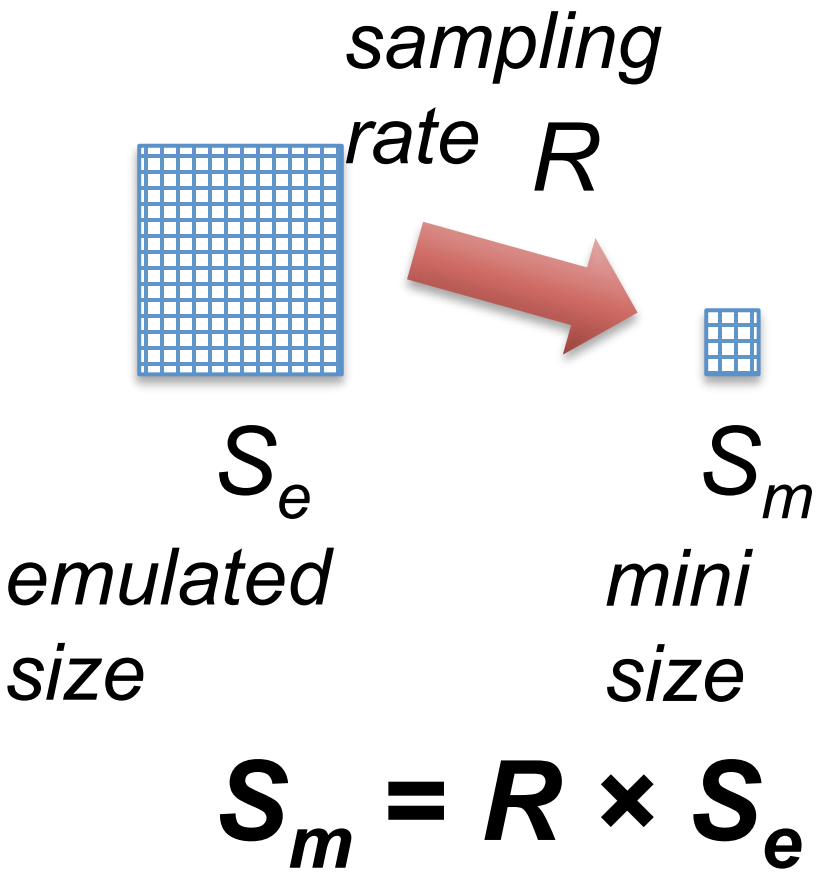
*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

If Too Hard to Simulate, Mini-Simulate*



*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Flexible Scaling with Mini-Sim



- Time/space tradeoff
  - Fixed sampling rate R
  - Fixed mini size S_m
- Example: $S_e = 1M$
  - $R = 0.005 \Rightarrow S_m = 5000$
  - $S_m = 1000 \Rightarrow R = 0.001$

Figure: Flexible time-space tradeoff in Mini-Sim configuration.

Using Mini-Sim to Adapt Online: Select Best Parameters

- Sometime, algorithm parameters tuning can lead to significant performance improvements
- *e.g.* LIRS has S stack size factor, f. Here we show mini-sims with $f = 1.1 - 3$.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Using Mini-Sim to Adapt Online: Select Best Parameters

- Sometime, algorithm parameters tuning can lead to significant performance improvements
- *e.g.* LIRS has S stack size factor, f. Here we show mini-sims with $f = 1.1 - 3$.

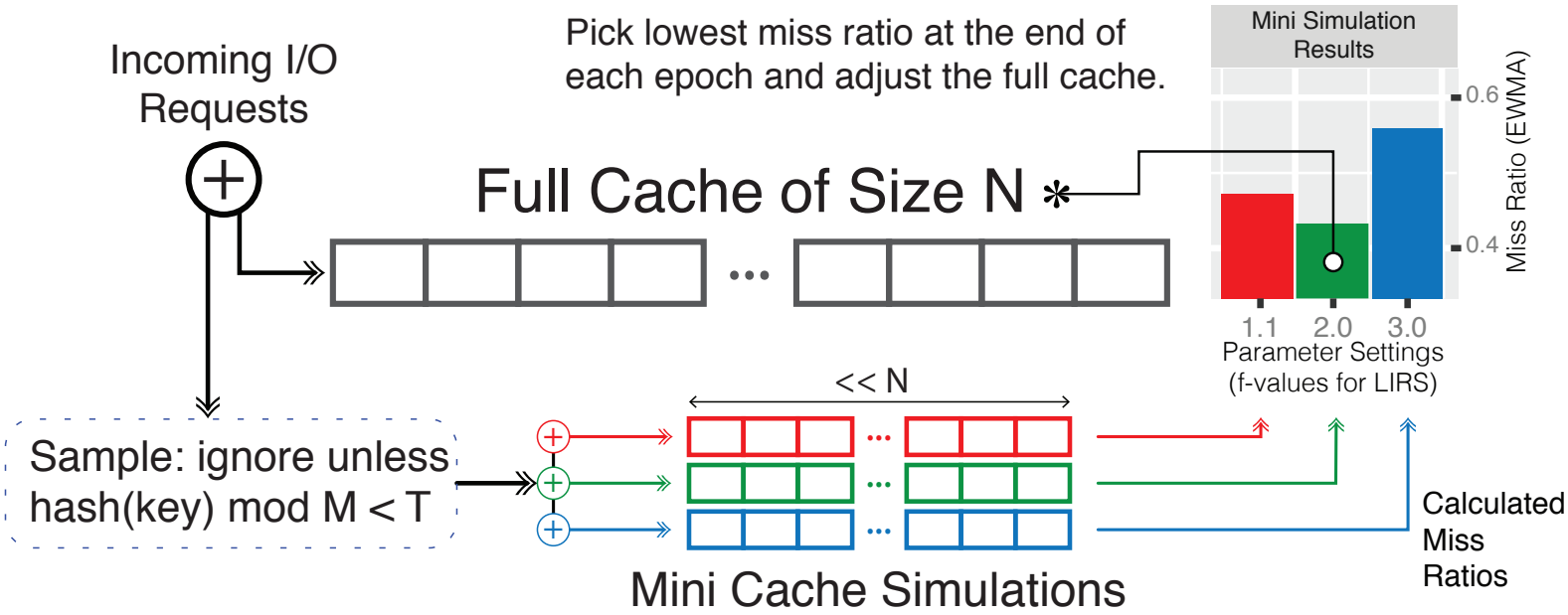


Figure: Overview of Mini-Sim*

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Mini-Sim Accuracy

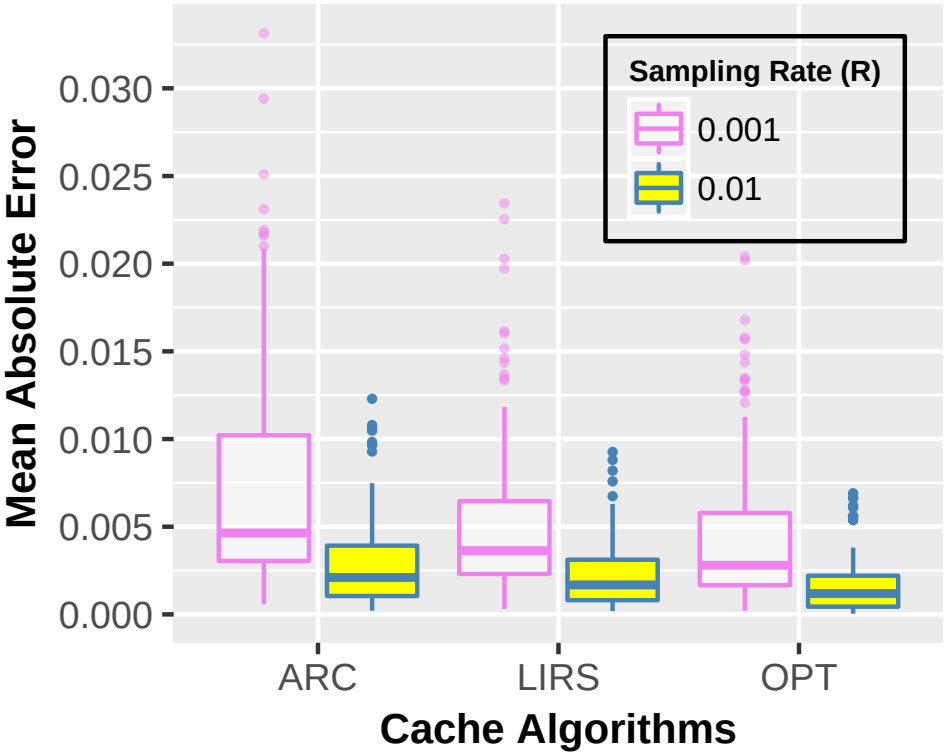


Figure: Mini-Sim Error Analysis. Distribution of mean absolute error for all 137 traces with three algorithms (ARC, LIRS, OPT) at two different sampling rates.

- 137 real-world traces
  - Storage block traces
  - CloudPhysics, MSR, FIU
  - 100 cache sizes per trace
- Mean Absolute Error
  - $\|\text{exact} - \text{approx}\|_1$
  - Average over all sizes

Problem: MRCs not always convex

Many applications of MRCs depend on convexity, but they aren't always.

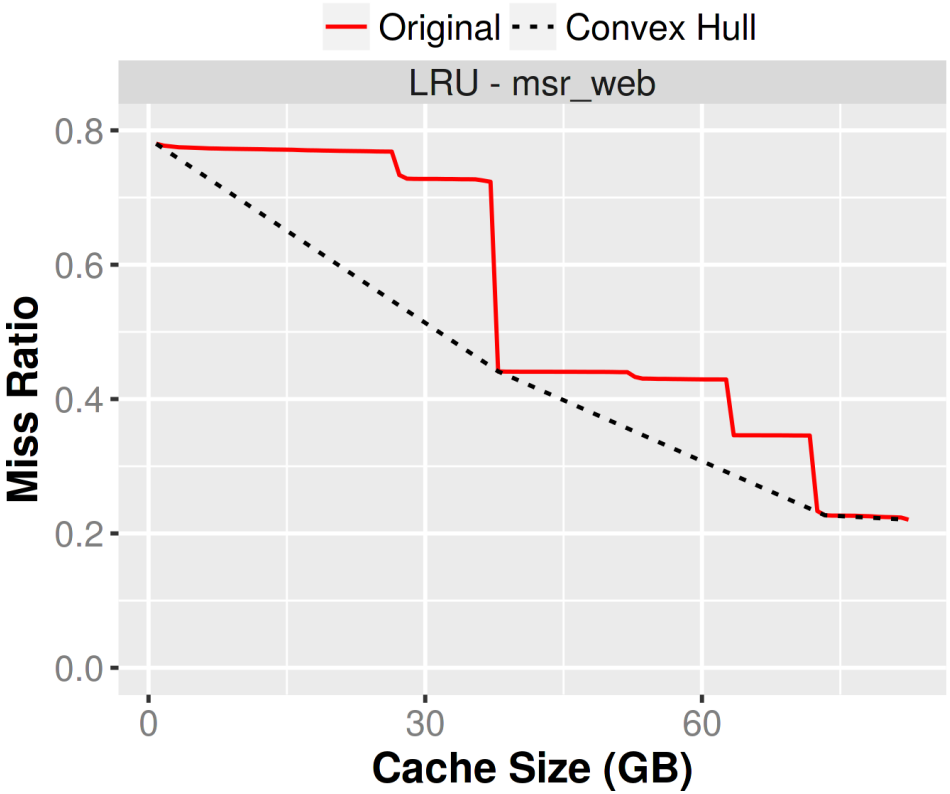


Figure: Convex hull of an MRC*

- Convex MRCs make optimal partition sizing trivial (no need for expensive search algorithms, greedy algorithms are optimal).
- If we can convert concave MRCs into convex ones, can sometime improve performance dramatically (this MRC is a good example).

*Beckmann et al, HPCA '15

Problem: MRCs not always convex

Many applications of MRCs depend on convexity, but they aren't always.

Talus [HPCA'15]

Make them convex!

- *Intuition:* Suppose you have 100GB, but the MRC is in a valley relative to 50GB and 150GB.
- Could just run at 50GB. But that gives more misses ... and is wasteful.
- What if we sent *half* the requests randomly to a cache of size 50GB?
- Sampling theorem says it should behave like the 100GB cache (just with half of the requests).
- What if we sampled more, or less? What if we changed the size from 50GB?
- What if we had *two* caches?

Problem: MRCs not always convex

Many applications of MRCs depend on convexity, but they aren't always.

Talus [HPCA'15]

Make them convex!

- Suppose MRC m of original policy is known. We want our miss rate m' to be convex.
- Partition cache into two parts α and β.
- Sample ρ fraction of requests to α by hashing the key.
  - Sampling Theorem applies to each! $m_\rho(s) = \rho m\left(\frac{s}{\rho}\right)$
- For a given s, we can pick two caches sizes r, t with $r < s < t$ on the *convex hull* of m.
- Sampling $\rho = \frac{t-s}{t-r}$-fraction to α gives

$$m_\rho(s) = \rho m(r) + \frac{s-r}{t-r} m(t)$$

- This is a linear interpolation between $m(r)$ and $m(t)$ and thus would make m' convex even if m wasn't in the $[r, t]$ range.

Convexification of MRCs via Talus*

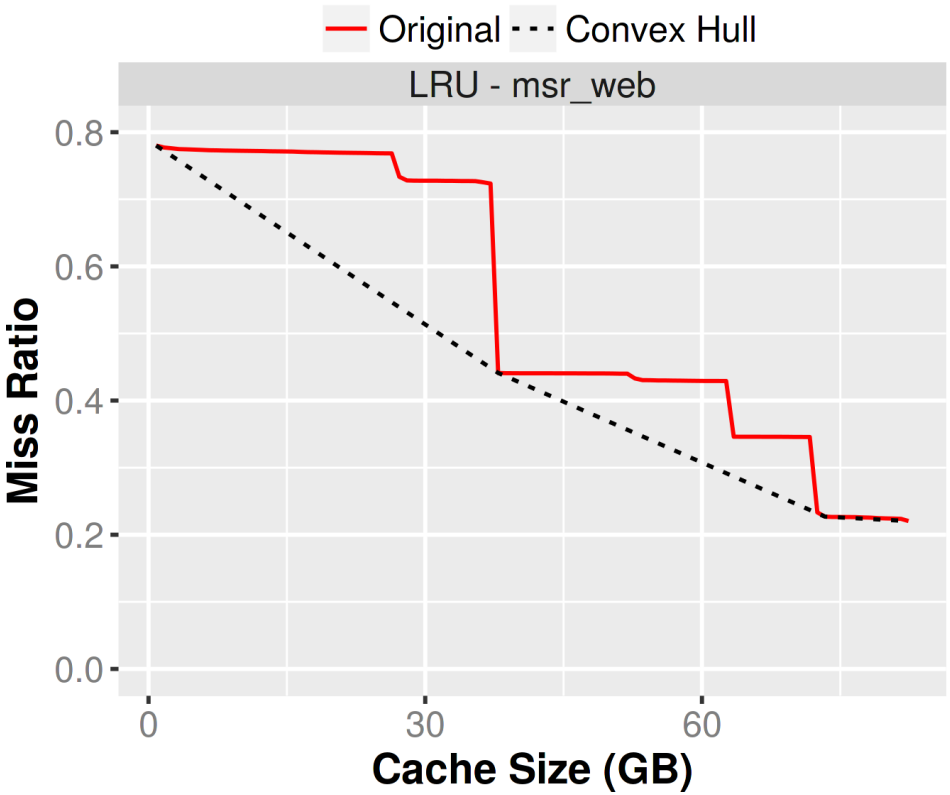


Figure: Convex hull of an MRC*.

- Assume original MRC (red) available.
- Emulate the cache on the convex hull via hashing (spatial sampling).
- Steer different fractions of references to α and β.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Beckmann et al, HPCA '15

Convexification of MRCs via Talus*

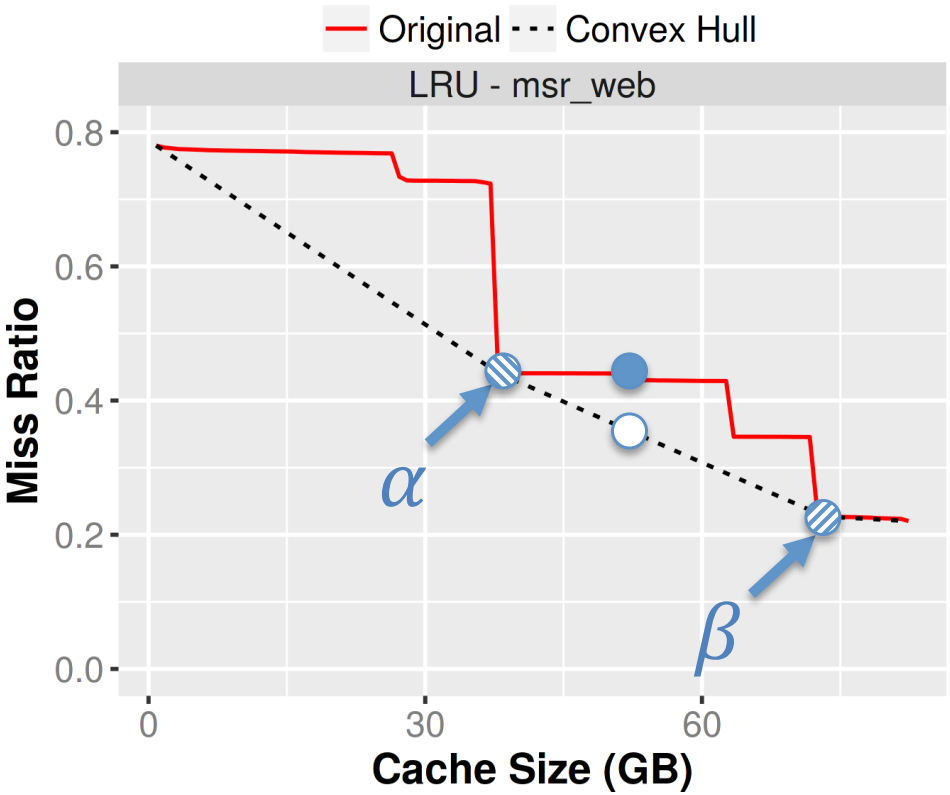


Figure: Convex hull of an MRC*.

- Assume original MRC (red) available.
- Emulate the cache on the convex hull via hashing (spatial sampling).
- Steer different fractions of references to α and β.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Beckmann et al, HPCA '15

Can we make Non-LRU curves convex?

- Need efficient online MRCs
- Support dynamic changes?
  - Workload and MRC evolve over time
  - Resize partitions, lazy vs. eager?
  - Migrate cache entries in "wrong" partition? Not clear how to merge/migrate state

SLIDE: Transparent Cliff Removal

- **S**harded **L**ist with **I**nternal **D**ifferential **E**viction
  - Single unified cache, no hard partitions
  - Defer partitioning decisions until eviction
  - Avoids resizing, migration, complexity issues
- New SLIDE list abstraction
  - No changes to ARC, LIRS, 2Q, LRU code
  - Replaces internal LRU/FIFO building blocks

Implementing Convexification using SLIDE lists

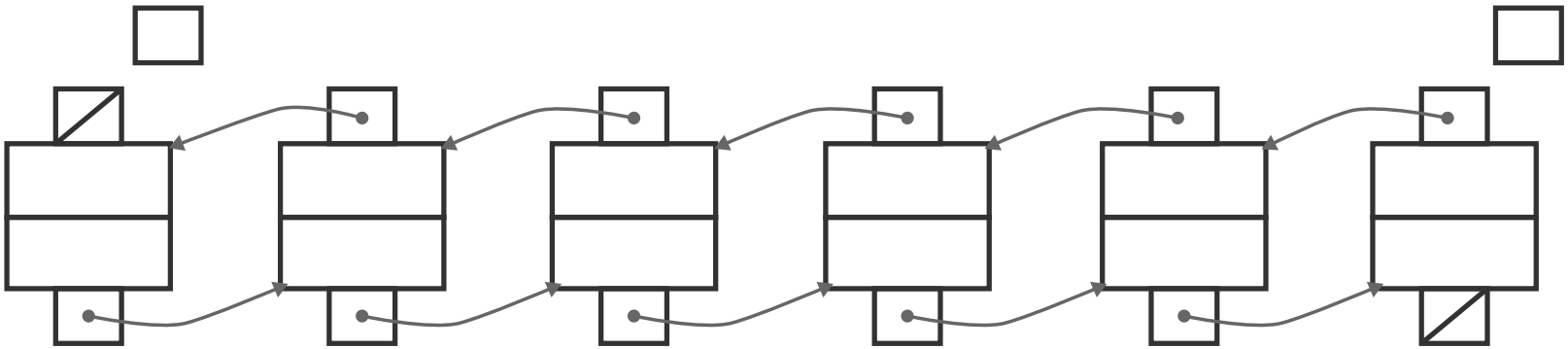


Figure: Overview of SLIDE lists*.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Implementing Convexification using SLIDE lists

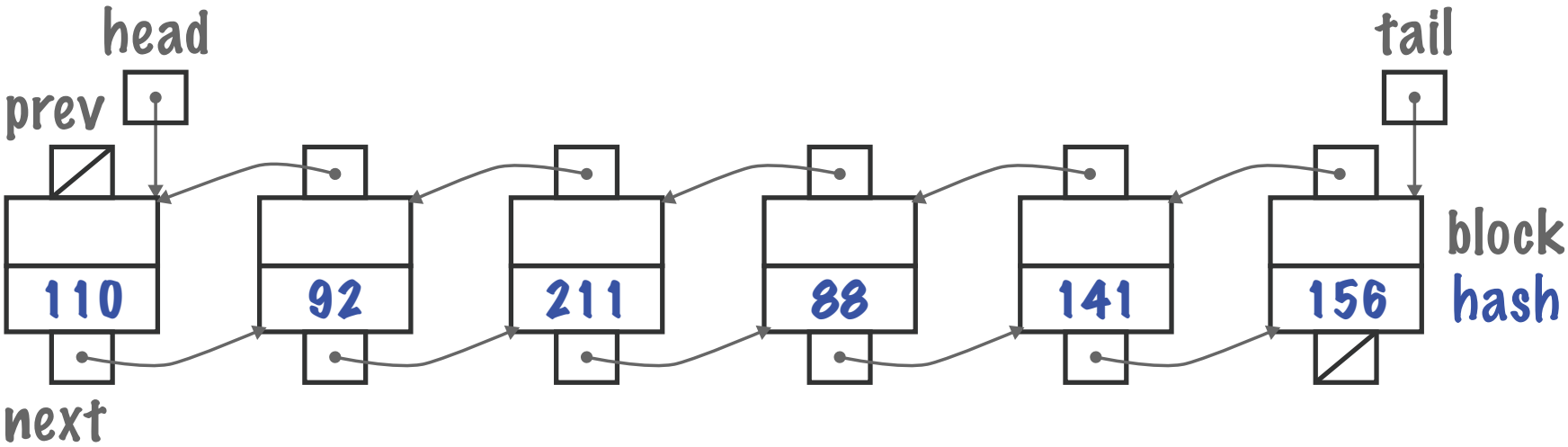


Figure: Overview of SLIDE lists*.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Implementing Convexification using SLIDE lists

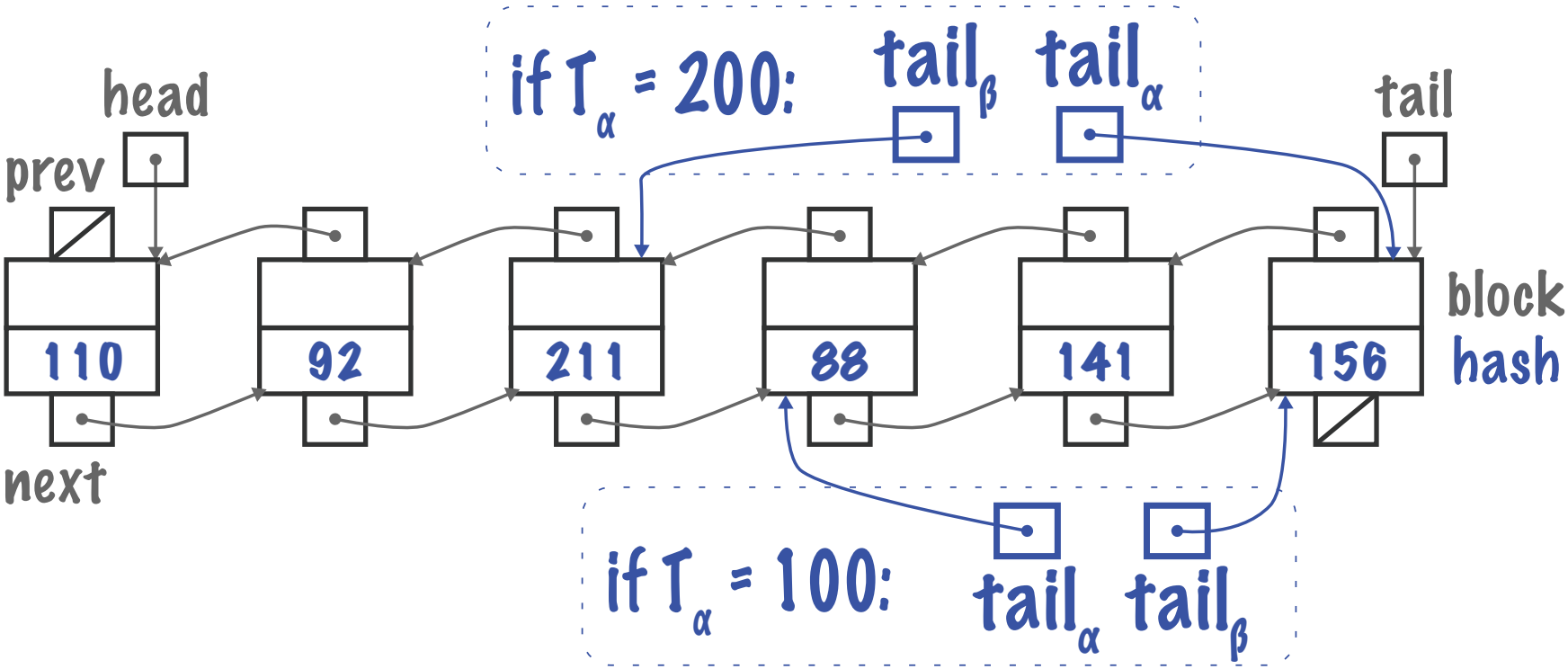


Figure: Overview of SLIDE lists*.

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

SLIDE Cliff Removal for ARC and LIRS

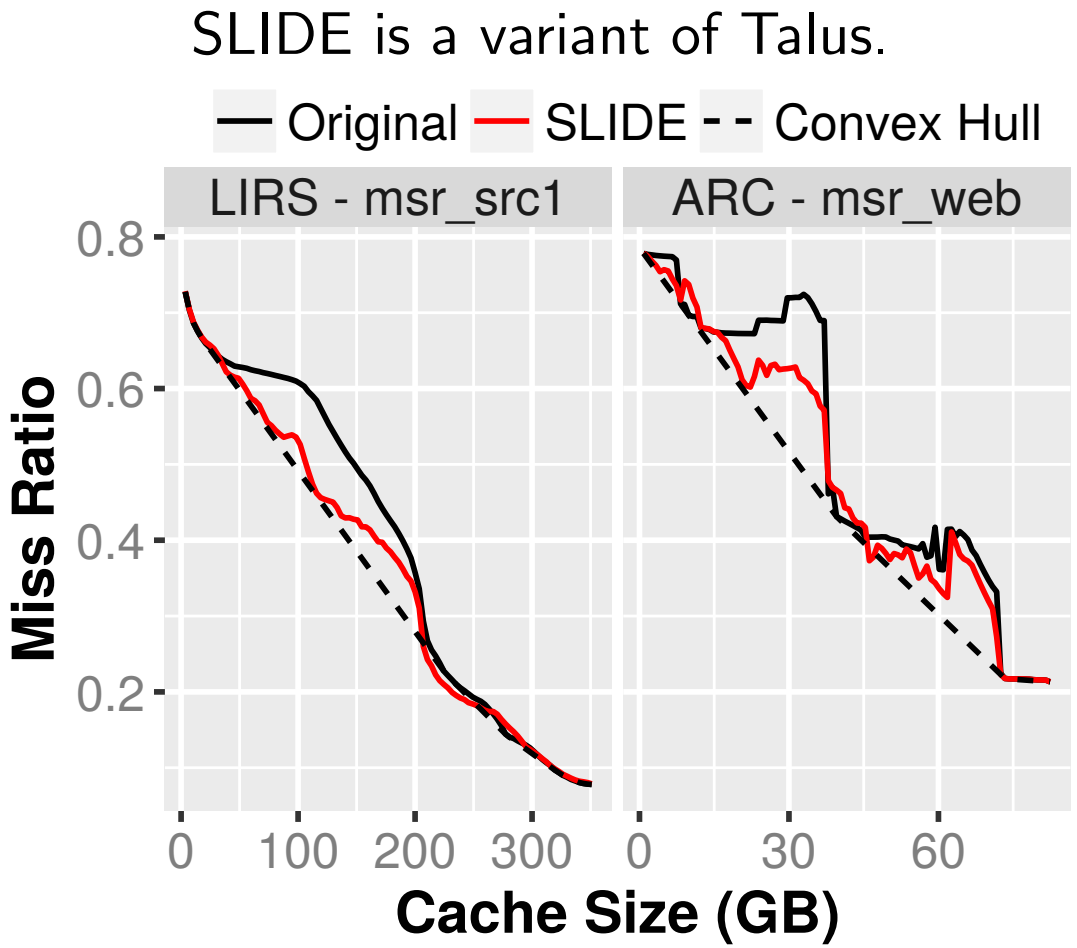


Figure: Bridging the gap from convex hull using SLIDE*

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

SLIDE Cliff Removal Results

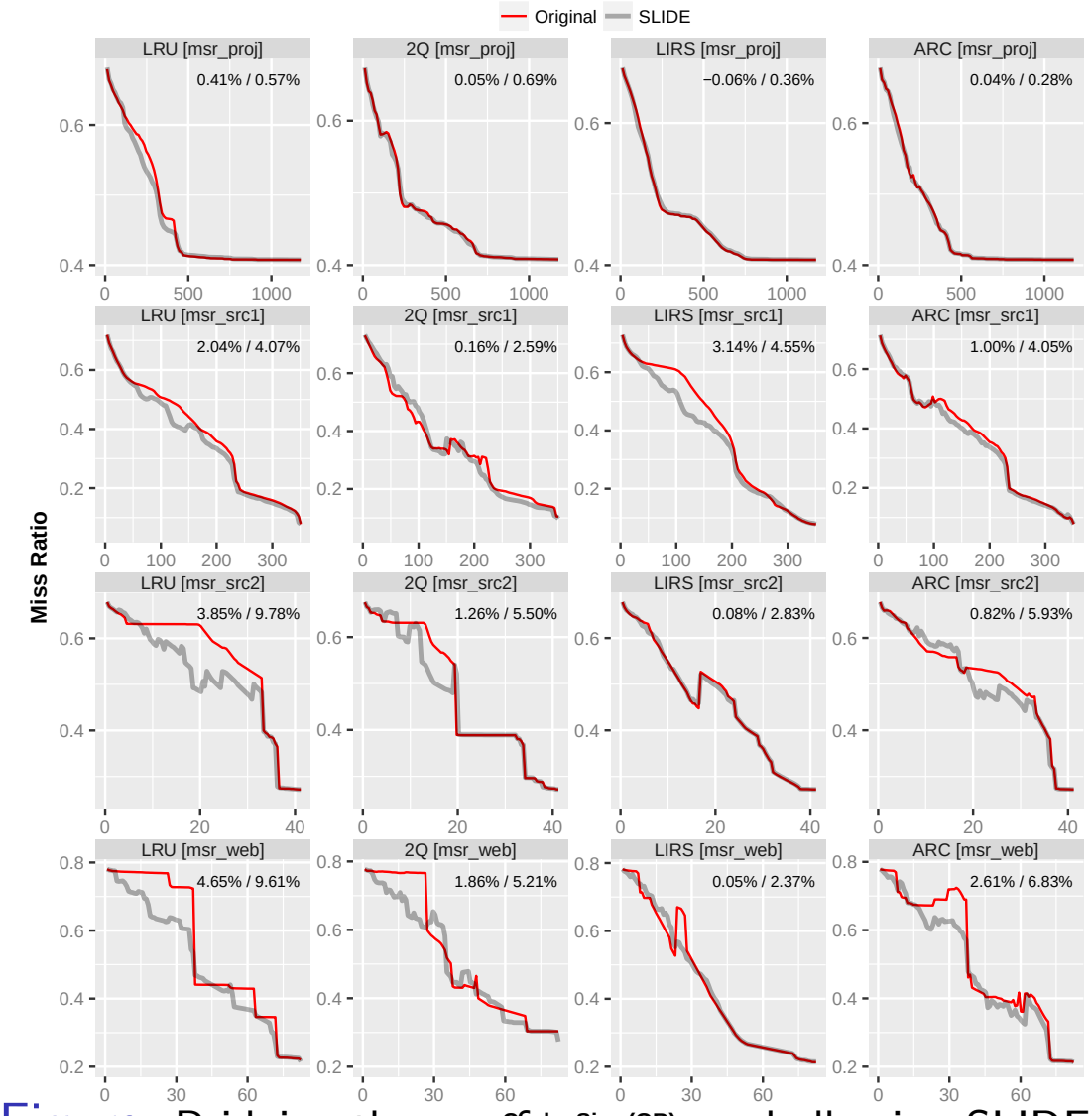


Figure: Bridging the gap from convex hull using SLIDE*

*Waldspurger *et al*, Cache Modeling and Optimization using Miniature Simulations, USENIX ATC '17

Sharing caches

Cache space often shared between multiple cache tenants.

- CPU world: Multiple cores concurrently accessing data in cache
- ... or part of cache could be shut off to save energy
- O/S: Multiple VMs or processes contending for main memory
- Cloud: Shared memcache instances of customers (e.g. Memcachier)

Sharing caches

Cache space often shared between multiple cache tenants.

- CPU world: Multiple cores concurrently accessing data in cache
- ... or part of cache could be shut off to save energy
- O/S: Multiple VMs or processes contending for main memory
- Cloud: Shared memcache instances of customers (e.g. Memcachier)

Would be great to have system-wide management of caches.
But competition between cache tenants, often with different workloads, can have adverse effects due to the lack of **isolation**.

Cache partitioning

Common approach is to partition the cache space between tenants.

- Without partitioning, some workloads cause cache to be vulnerable to pollution
  - one tenant might rapidly scan over data and cause all other contents to be evicted
  - one tenant might have a relatively slow request stream, and never see cache benefit
- Cache could run different replacement policies for different users
- ... or a single policy, with tenant ownership information used to improve over regular replacement decisions

Cache partitioning

Common approach is to partition the cache space between tenants.

- Without partitioning, some workloads cause cache to be vulnerable to pollution
  - one tenant might rapidly scan over data and cause all other contents to be evicted
  - one tenant might have a relatively slow request stream, and never see cache benefit
- Cache could run different replacement policies for different users
- ... or a single policy, with tenant ownership information used to improve over regular replacement decisions

Tension between minimizing overall miss rate and being "fair" to each tenant.

Cache partitioning

Wait, what do you mean by fairness?

Common notions:

- Each tenant given reserved cache space (and thus some performance isolation)
- Reward the tenants who most benefit from cache space in terms of miss rate
- QoS / SLAs, such minimum target hit rate

Each can be translated into to a minimum cache size constraint for each tenant.

Cache partitioning formalized

General cache partitioning problem

Let N denote cache size. Let c_i denote cache blocks allocated to tenant $i \in I$ and a_i minimum cache size for tenant $i \in I$.
Let $m_i(x)$ denote miss rate of tenant $i \in I$ for cache size x during the period.

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in I} m_i(c_i) \\ \text{s.t.} \quad & \sum_{i \in I} c_i = N \\ & c_i \geq a_i \qquad i \in I \end{aligned}$$

Problem is *NP*-complete for general functions (reduction to Knapsack or Partition).

Solving cache partitioning (1)

How do we compute this $(c_i)_{i \in I}$ partition?

Property 2: Monotonicity

The MRCs of all stack algorithms are monotone (non-increasing).

More space – fewer misses.
Empirically true for many but not *all* other policies (think RANDOM for counterexample)

Solving cache partitioning (1)

How do we compute this $(c_i)_{i \in I}$ partition?

Property 2: Monotonicity

The MRCs of all stack algorithms are monotone (non-increasing).

More space – fewer misses.
Empirically true for many but not *all* other policies (think RANDOM for counterexample)

Property 3: Convexity

MRCs of many algorithms are convex (or can be made convex with Talus). Specifically,

$$m(tx + (1-t)x) \leq tm(x) + (1-t)m(x) \text{ for all } t \in [0,1]$$

.

(Note that we're considering a natural extension of $m(x)$ to the reals.)

Solving cache partitioning (2)

How do we compute this $(c_i)_{i \in I}$ partition?
Bennett Fox in 1966* defines an optimal algorithm for the cache partitioning problem for these properties.

Theorem (Optimal algorithm)

Start with $c_i = a_i$ for all $i \in I$.
If m is convex and monotone, repeatedly allocate a unit of space to the tenant i with the largest **marginal benefit**

$$m_i(c_i + 1) - m_i(c_i) \geq m_j(c_j + 1) - m_j(c_j)$$

for all $j \in I$ by setting $c_i \leftarrow c_i + 1$ until out of space ($\sum_{i \in I} c_i = N$).
This algorithm is **optimal.**

Mnemonic: Greedily climb the steepest slopes.

*See discussion in Stone et al. IEEE ToC 41(9) 1992

Proof Sketch.

Use Lagrangian multipliers adapted to discrete optimization to show that all derivatives $m_i'(c_i) = m_j'(c_j)$ at the optimal point $(c_i)_{i \in I}$ for $i, j \in I$. □

The running time is $O(K + N \log K)$ for $K = |I|$.*

*Galil and Meggido (J.ACM 1979) provide a $O(\max K, K \log \frac{N}{K})$ algorithm

Combining MRCs

Property 4

MRCs can be combined.

- Generally difficult and dependent on workload.
- If we assume average interleaving of the workloads, we can approximate.
- From Whirlpool (Backmann *et al.*, ASPLOS 2016).

```
function combineMRCs(m1,m2):
    m = array(N)
    s1,s2 = 0
    for s=0 to N:
        m[s] = m1[s1]+m2[s2]
        s1 += m1[s1] / m[s]
        s2 += m2[s2] / m[s]
    return m
```

- Think of a cache as having a rate of outgoing *flow* based on evictions.
- The flow is exactly the miss rate at that size.

Exclusive Caches Increasingly Studied for CPU Caches

Benefits of Exclusivity in CPU L2/LL3

"We observe that server workloads benefit tremendously from an exclusive hierarchy with large private caches ... For a 16-core CMP, an exclusive cache hierarchy improves server workload performance by 5-12% as compared to an equal capacity inclusive cache hierarchy." *

*High Performing Cache Hierarchies for Server Workloads – Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches, HPCA 2015

Exclusive versus Inclusive: Storage

Recent developments:

- Low-latency RDMA networks are becoming cheaper, commodity
- How low latency? Less than the cost of a system call?
- Mesh cache hierarchies now feasible?
  - The ratio $\frac{\text{local cache size}}{\text{remote cache size}} \rightarrow 1$
  - Happening already with hyper-converged distributed storage systems?
  - Inclusive caching is very inefficient in such cases
- Persistent memory technologies introduce size differences between layers in caches hierarchy that are smaller less than before
- But exclusive caching has not seen much use in practice
- **Conjecture:** exclusive caching could become more prevalent in the next decade

Cache hierarchies

Some large systems, like web caches, have had to consider the placement of data in a *cache hierarchy*.

Underlying assumption in much analytic work:

Independent Reference Model (IRM)

Each item i in cache is accessed with i.i.d. probability q_i.

Open question: How accurate is this assumption?

Cache hierarchies

Some large systems, like web caches, have had to consider the placement of data in a *cache hierarchy*.
Underlying assumption in much analytic work:

Independent Reference Model (IRM)

Each item i in cache is accessed with i.i.d. probability q_i.

Open question: How accurate is this assumption?

Theorem (Che/Fagin approximation)

The miss rate of item $i \in I$ in cache of size N is $m_i = e^{-q_i \tau}$ where τ is the unique root of

$$\sum_{i \in I} \left(1 - e^{-q_i \tau}\right) = N$$

called the characteristic time for the trace. [a]

[a] Che *et al.* IEEE J. Selected Areas in Communications 2002

Relies on q_i being small, so $\ln q_i \approx -q_i$.

Cache hierarchies

Cache as a low-pass filter

Observe miss rate has phase transition when q_i approaches τ^{-1}. Items with access frequency below say $e^{-1}\tau^{-1}$ are effectively one-time items. Can thus view cache as a low-pass filter with cut-off frequency from above of τ^{-1}.

Theorem (Cache dimensions)

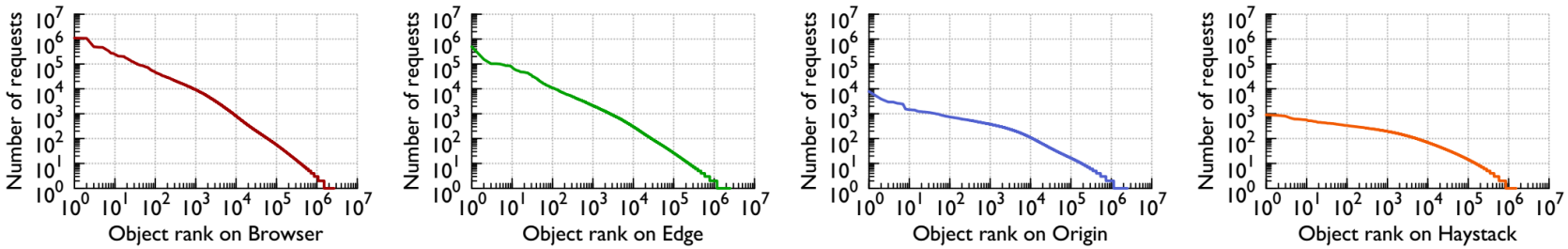
Define characteristic time τ_ℓ for each cache layer ℓ. Then a cache hierarchy should have $\tau_\ell > \tau_{\ell'}$ for $\ell > \ell'$.

The equation for the characteristic ($m_{\ell i} = e^{-q_{\ell i}\tau_\ell}$) can be inverted to calculate space for each layer.

- Each layer can track τ_ℓ by tracking timestamps of last hit to an item.
- Access frequencies varies between layers. Assumption that worthwhile tracking frequent items (e.g. documents).

Cache hierarchies

Empirical view from Facebook's photo caching hierarchy*



Hierarchy impact

Lower level caches absorb recency and remove the popularity skew.

*Huang *et al.* SOSP 2013

Modern Multi-ter Hierarchies

Figure: Increasing hardware complexity with more layers*.

*Courtesy CachePhysics, Inc.

Multi-Tier Cache Modeling (LRU)

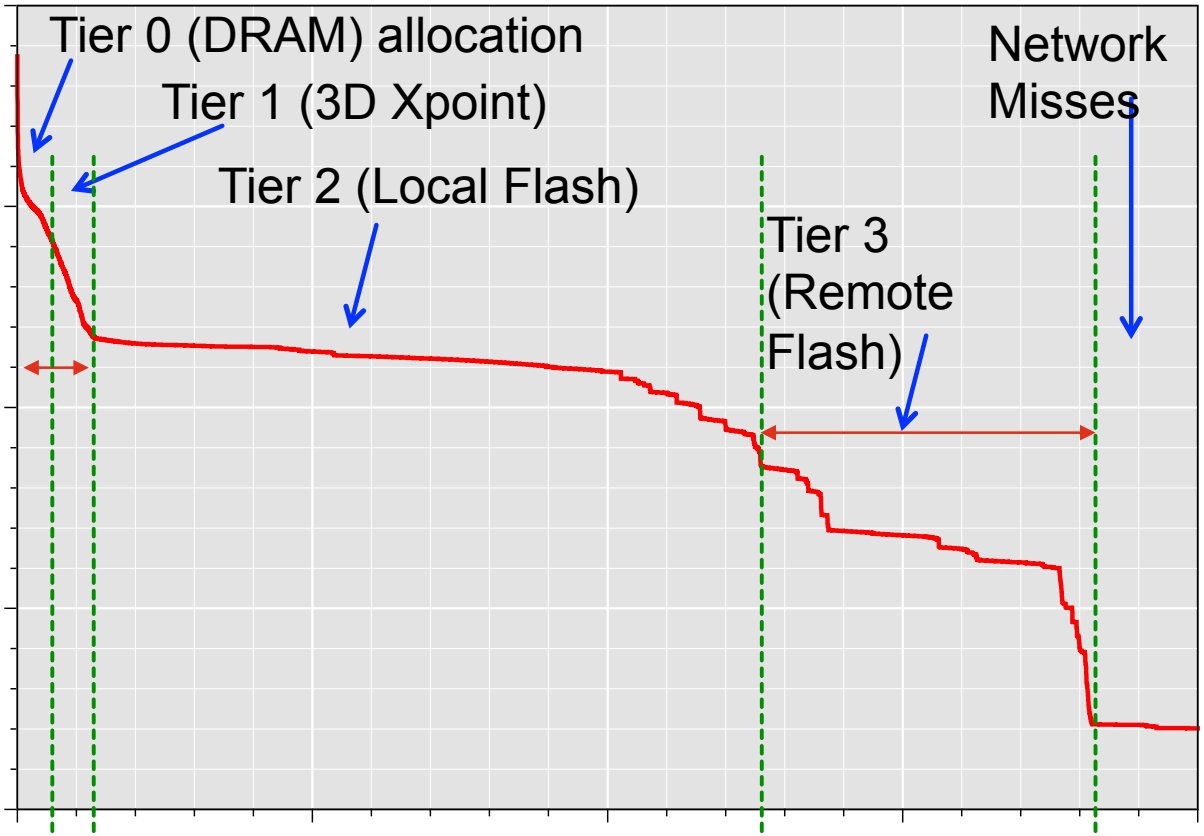


Figure: Model multi-tier cache system performance*.

*Courtesy CachePhysics, Inc.

Using Cache Utility Curves for Latency Prediction

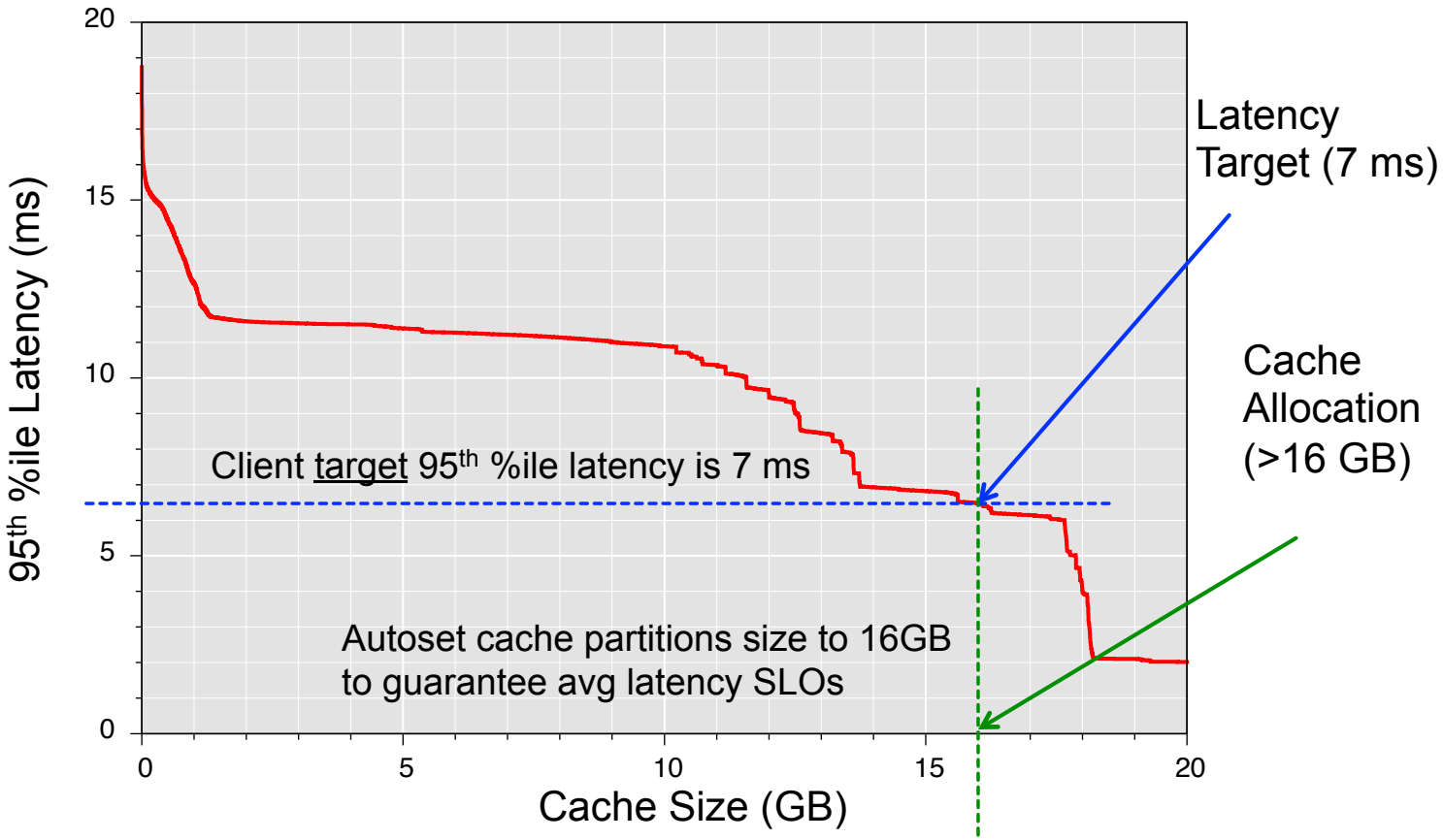
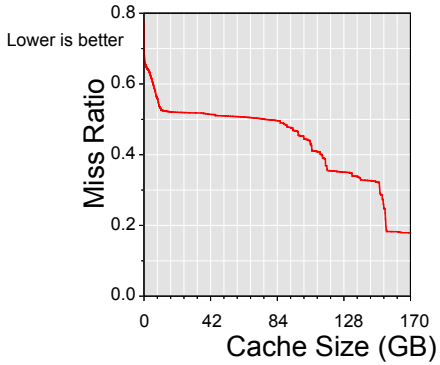


Figure: Model multi-tier cache system performance*.

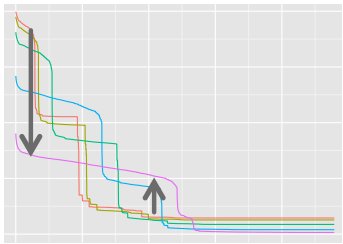
*Courtesy CachePhysics, Inc.

Review of Some Cache Modeling Use Cases

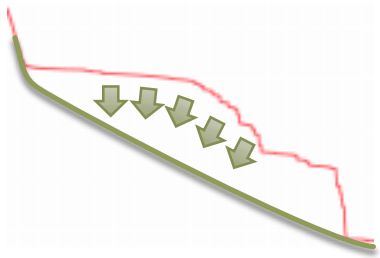
Monitoring



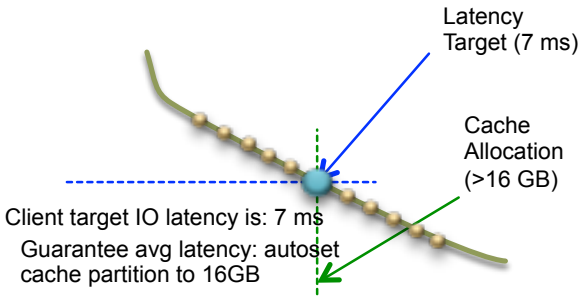
Auto-Select Policies
(dynamic parameters)



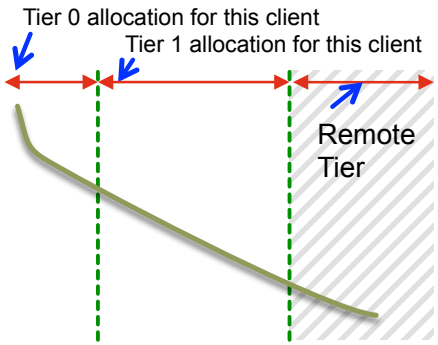
Latency Reduction
(Thrashing Remediation)



Latency Guarantees



Accurate Tiering



Multi-Tenant Isolation

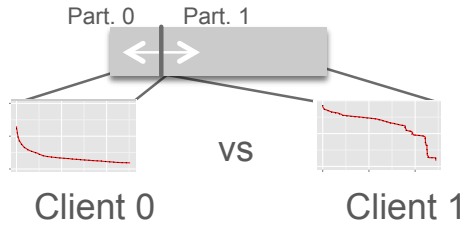


Figure: Some example use cases of cache utility curves*.

*Courtesy CachePhysics, Inc.

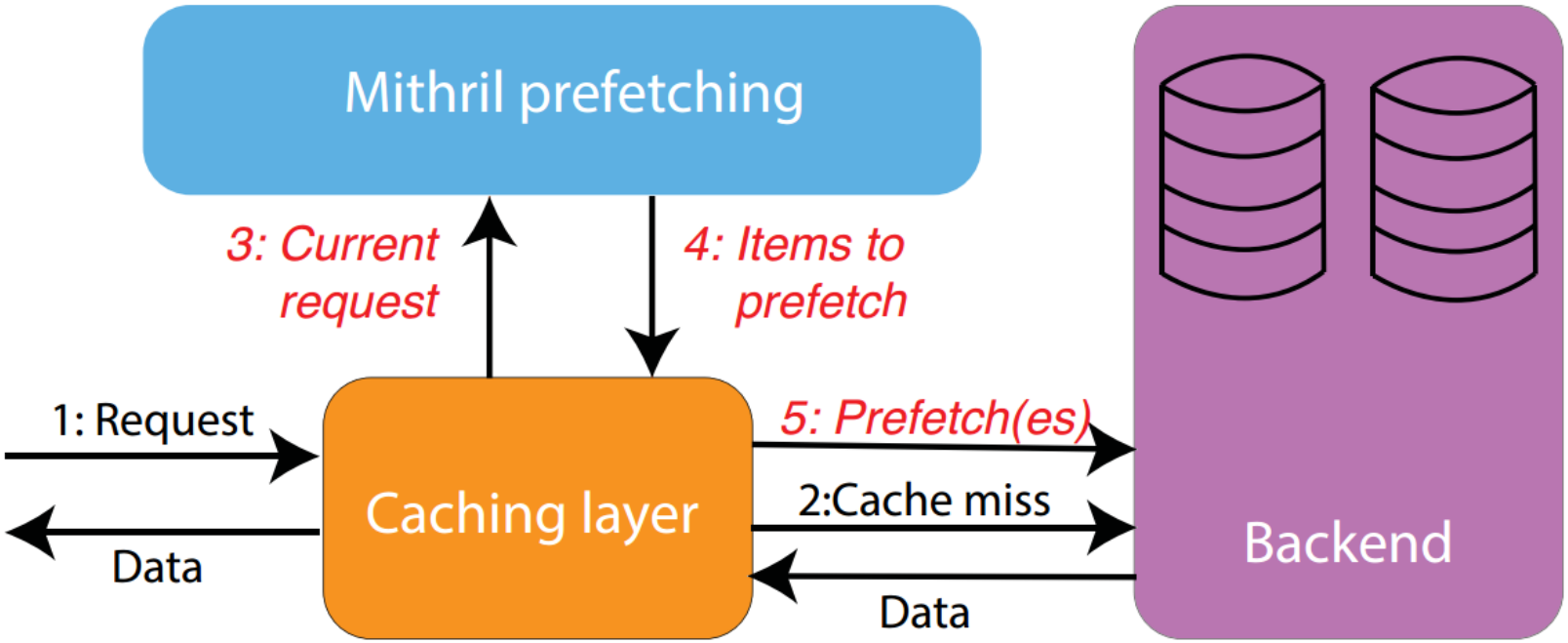
Cache prefetching

Low-level caches often absorb repetitive work – think of scans or loops.

What if we could predict what will be requested, and have it ready in time?

Cache prefetching

When a cache requests items ahead of time in anticipation of their access.



Cache prefetching approaches (1)

How should we determine which items to request ahead of time? Key approaches are two-fold.

- **Sequence-based**: anticipate access to consecutive block identifiers
  - AMP* (Adaptive Multi-Stream Prefetching) dynamically adjust the prefetch length and timing to avoid cache pollution.
  - AMP tries to be careful not to request data that is likely already in flight (prefetch wastage)
  - TaP† uses a table to track longer sequentiality, thus detecting interleaved workloads

*AMP: Gill & Bathen, FAST 2007

†Li *et al.* FAST 2008

Cache prefetching approaches (2)

How should we determine which items to request ahead of time? Key approaches are two-fold.

- **History-based**: tracking deep correlation among past accesses, normally expensive
  - Probability Graph* takes a Bayesian approach, creating dependency graph of items being accessed (expensive)
  - QuickMine† uses data mining techniques, but requires hint from the application
  - Mithril‡ also uses data mining but for teasing out mid-popularity items.

*Griffionen & Appel, USENIX ATC 1994

†Soundarajan *et al.* , USENIX ATC 2008

‡Yang *et al.* SOCC 2017

Mithril cache prefetching

Requests	...	a	a	b	c	b	a	c	d	c	e	f	d	g	g	e	g	d	e	g	a	g	g	g	...
Timestamps	...	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	...

1: Record/ Convert

Minimum support ***R***=2
Maximum support ***S***=6
Look ahead range Δ=5

Blocks	ts1	ts2	ts3	ts4	ts5	ts6
a	20	21	25	39		
b	22	24				
c	23	26	28			
d	27	31	36			
e	29	34	37			
f	30		not enough info			
g	~~32~~	~~33~~	~~35~~	~~38~~	~~40~~	~~41~~

~~42~~ too frequent

Timestamp matrix ***T***

2: Mining

compare each two rows

f	# of ts < ***R***
g	deleted: # of ts > ***S***

Ignored for mining

a	20	21	25	39		
b	22	24	X	X		

Different length

c	23	26	28			
d	27	31	36			

$|28-36| > \Delta$

Not associated

a	20	21	25			
c	23	26	28			

Weak association
$23-20 \le \Delta$
$|21-26| \le \Delta$, $|25-28| \le \Delta$

d	27	31	**36**			
e	29	34	**37**			

Strong association
requirements of weak association
+
difference of one ts pair = 1

Associated

Challenges and Open Questions

Time-varying behavior of caches are not well understood.

- Many workloads appear to have time periods of very low cache utilization.
- In context of Cloud and Edge computing, should we give up memory to save money and rent it back in the nick of time to warm for next busy period?

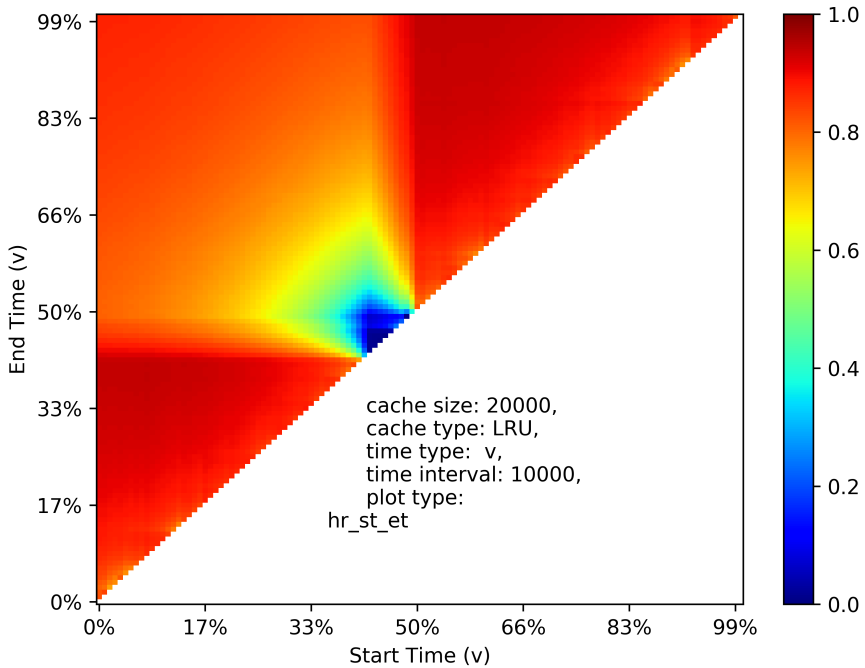


Figure: Each pixel denotes hit rate of in MSR MDS trace assuming cache started from scratch at time x and ran until time y.

Challenges and Open Questions

- Is the persistence property of non-volatile memory useful for caching?
  - Can we know what data is still useful when the cache comes back up?
  - What expectations can we have for data consistency with possibly very stale data?
  - Facebook struggled to maintain stronger levels of consistency in their long-lived caches.
- Challenges in mesh cache hierarchies.

Takeaways

- Cache replacement has gotten a lot of mileage out of LRU, but workloads and questions shifting
- Plethora of new replacement algorithms, and modifications to problem/interfaces.
  - Priority queues, application-level hints, cache prefetching
- MRCs allow you to reason about parameters you can use to change and optimize your cache.
  - Including how to partition space between tenants, resize a cache, *etc.*
- Miniature simulation allows you to quickly reason about different cache replacement algorithms
- Cache hierarchies abound, but difficult to reason about.

Caching: an old field of paramount importance, ripe with low-hanging fruit

MIMIRCACHE: Cache Analysis in Python

Tool for efficient and easy cache analysis.

- Two versions: PyMimircache (accessible) and CMimircache (fast)
- Allows **researchers** to study and design cache policies.
- Allows **system administrators** to analyze and visualize cache performance.
- Main goals:
  - performance
  - extensibility and flexibility
  - ease of use

MIMIRCACHE: Cache Analysis in Python

What can Mimircache do?

- Help visualize and analyze your cache policy and workload
  - Heatmaps for understanding cache dynamics
  - Various visualizations of workload similarity
  - Scan visualizer
- Help design and refine your caches
  - Compare different cache replacement algorithms
  - Implement and test your own caching strategies

MIMIRCACHE: Cache Analysis in Python

Using Mimircache.

- Instructions: `http://mimircache.info/`
- Installation: `$ pip3 install pymimircache`

```
from PyMimircache import Cachecow
c = Cachecow()
c.csv(dat, init_params={"real_time": 1, "label": 5})

# iterate through the workload requests
for req in c:
    print(req)
```

MIMIRCACHE: Cache Analysis in Python

Another example

```
# print some info about the workload
print(c.stat())

# plot hit ratio curve with comparisons
c.plotHRCs(["LRU", "LFU", "Optimal", "ARC", "SLRU"])

# interval hit ratio
c.twoDPlot(plot_type="interval_hit_ratio",
           cache_size=20000)

# draw heatmap of hit ratio between start (x) and end (y) time
# using real time of requests (r) within a particular interval
c.heatmap(plot_type="hr_st_et", cache_size=20000,
           time_mode="r", time_interval=200000000)
```

MIMIRCACHE: Cache Analysis in Python

Example output #1.

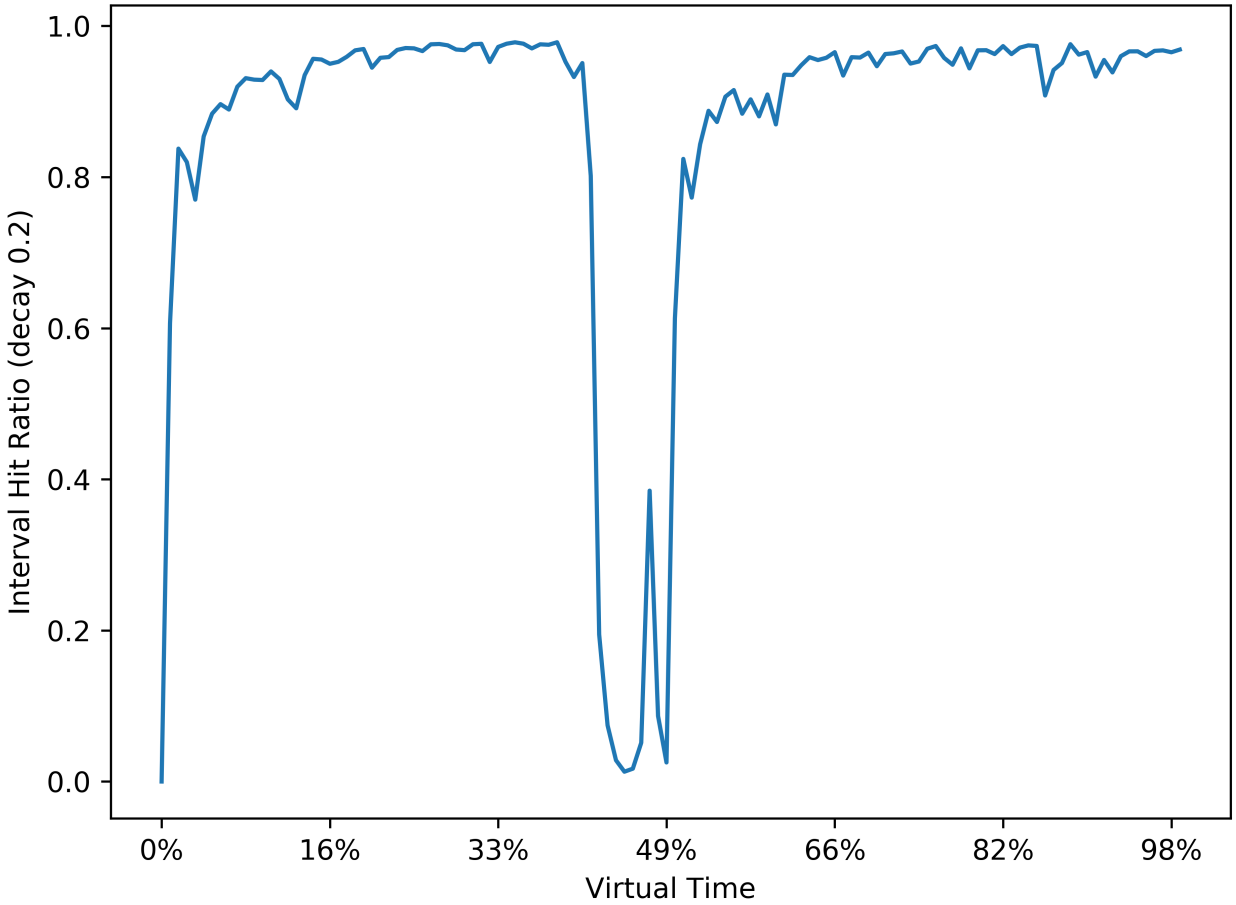


Figure: Hit rate in MSR MDS trace on given interval.

MIMIRCACHE: Cache Analysis in Python

Example output #2.

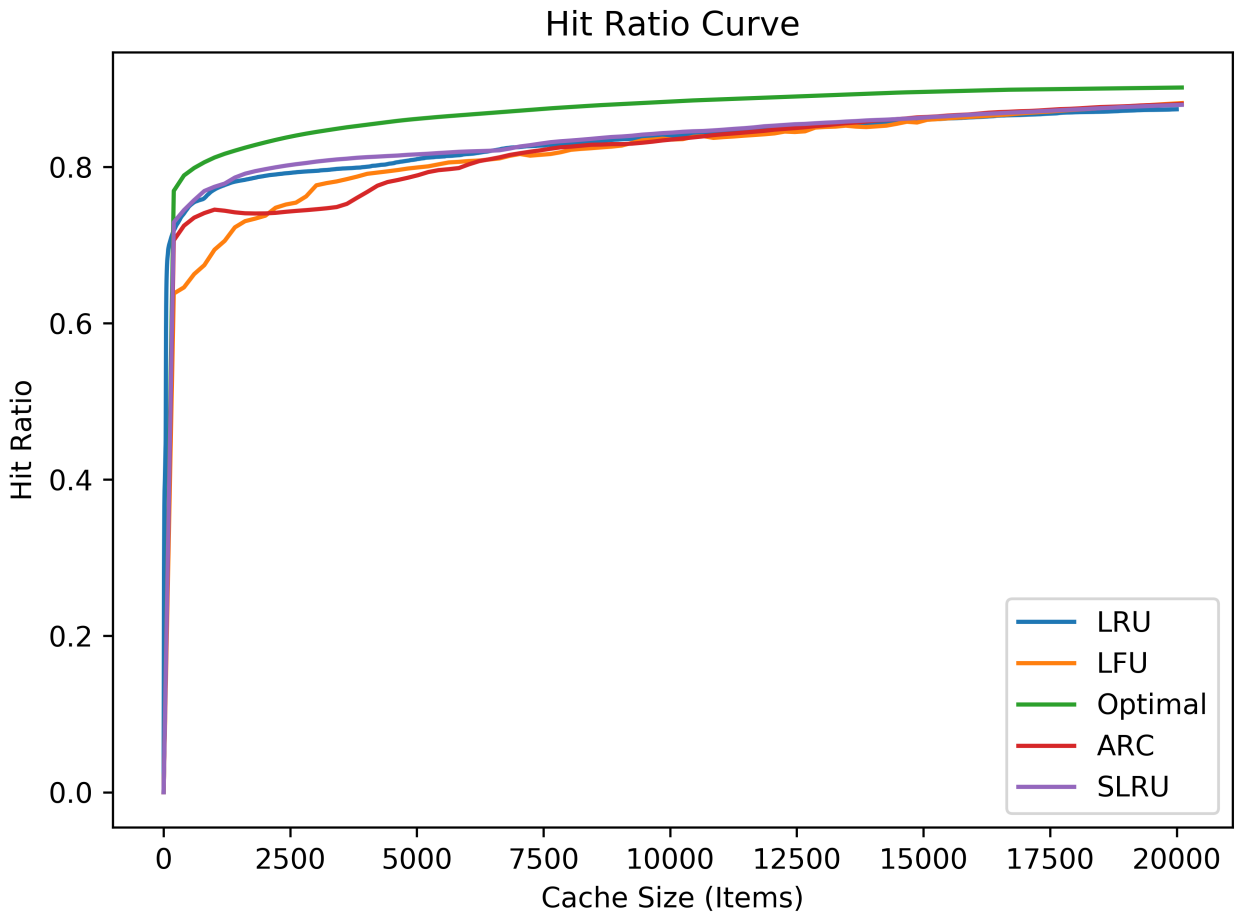


Figure: Hit rate of various cache replacement policies on MSR MDS trace.

MIMIRCACHE: Cache Analysis in Python

Example output #3.

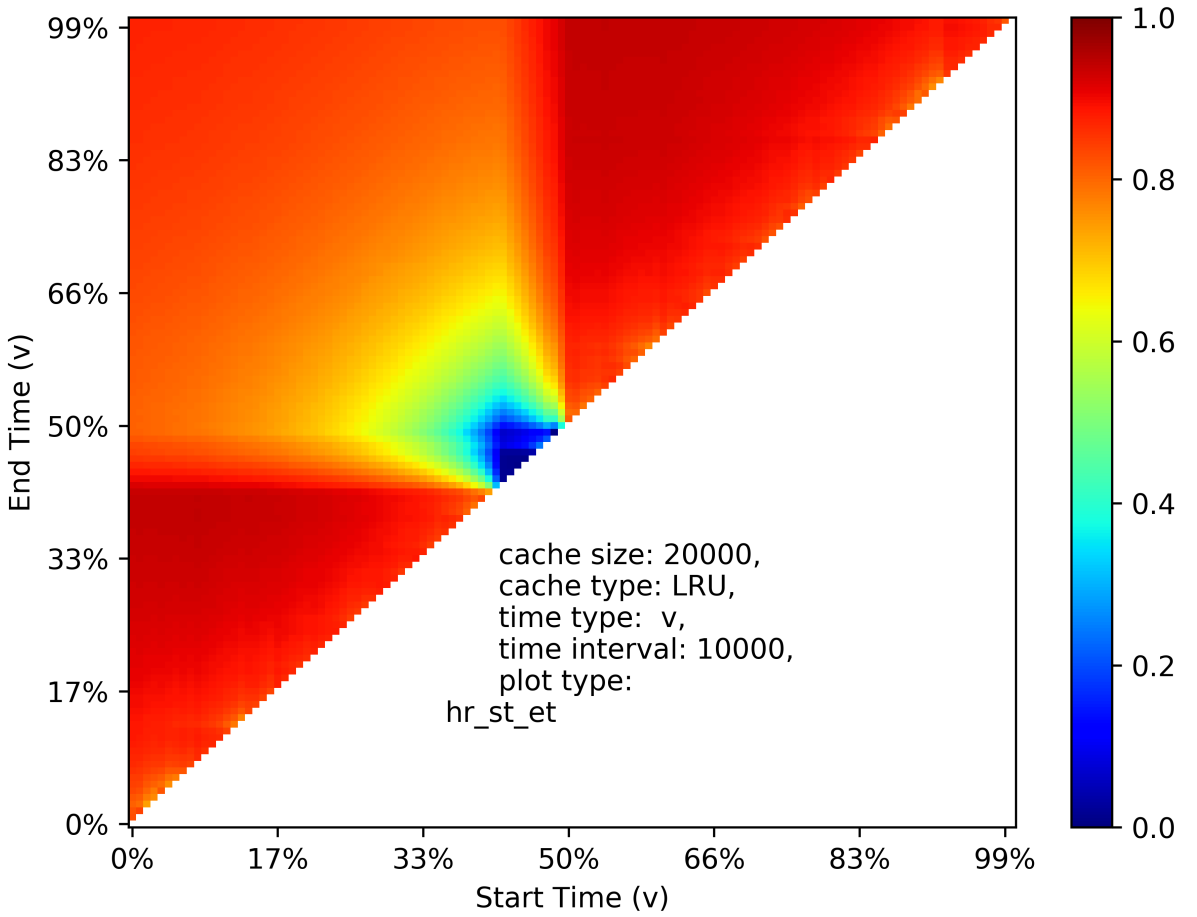


Figure: Heatmap with each pixel showing hit ratio on the portion of MSR MDS trace between logical time x and y.