Squeezing More Performance from a Shared Cache Budget
via Some Bunny Magic

Cache-Centric Multi-Resource Allocation for Storage Services

Chenhao Ye, Shawn Zhong

Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau

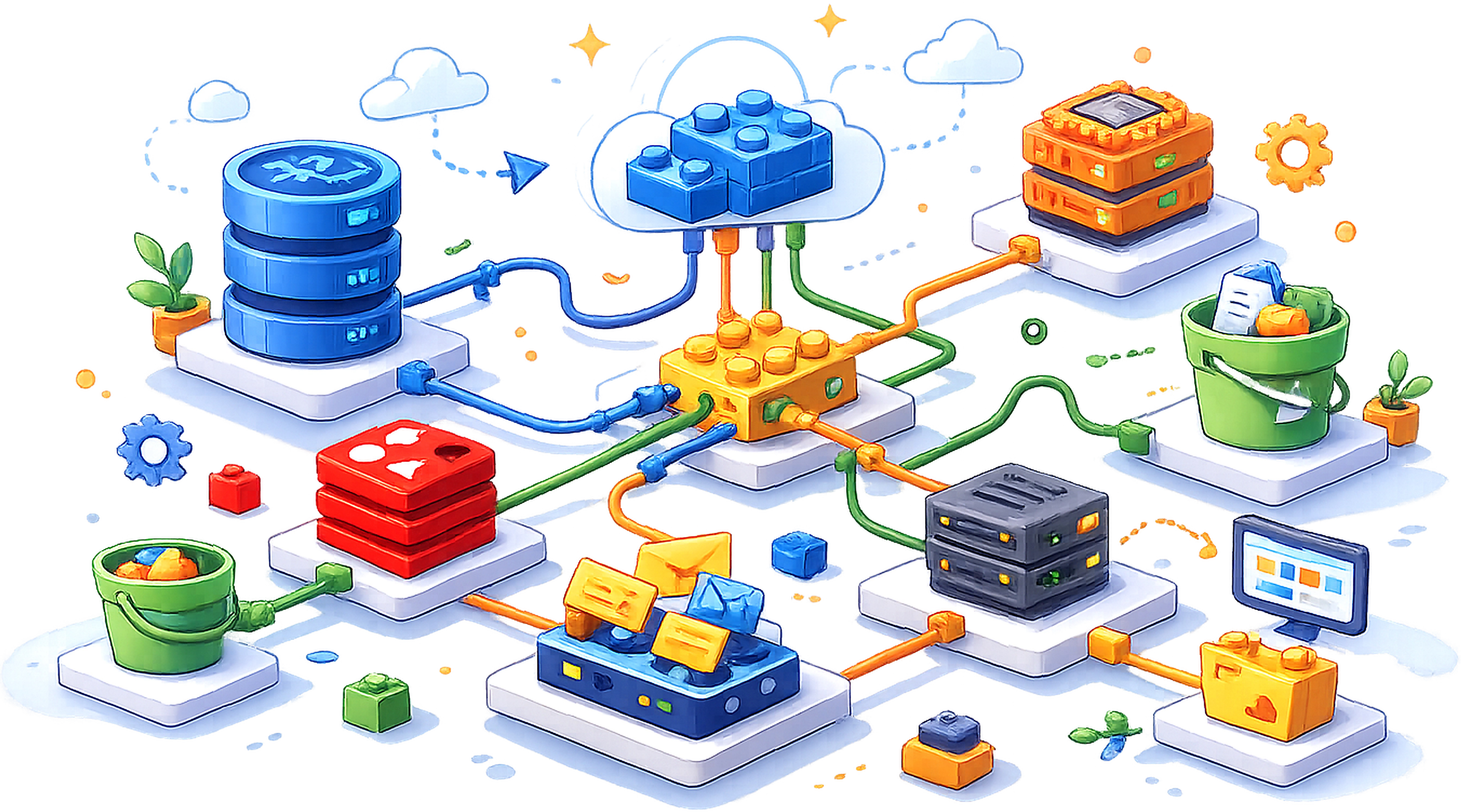
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

ARTIFACT EVALUATED USENIX AVAILABLE

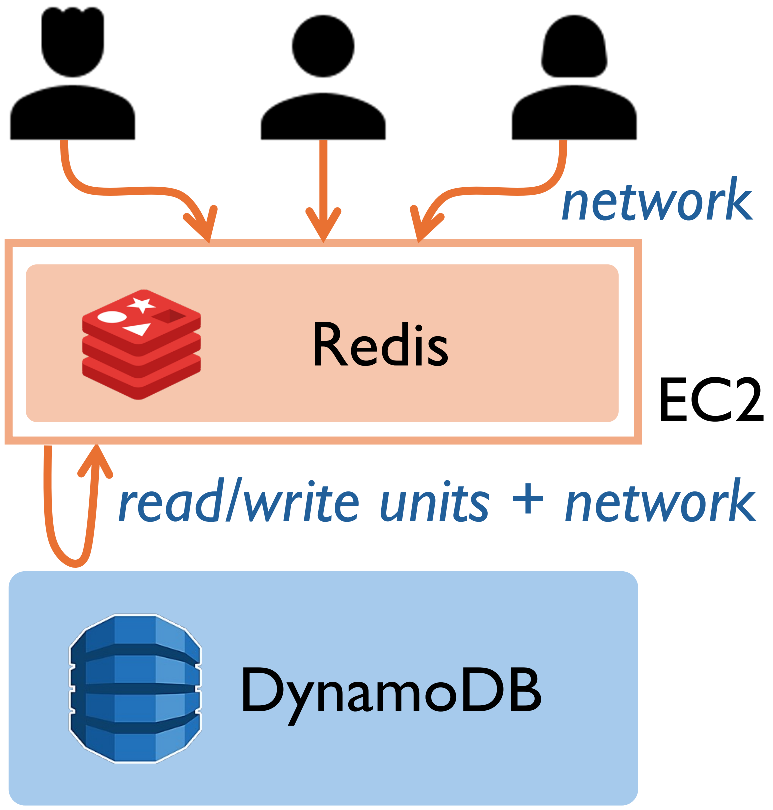
ARTIFACT EVALUATED USENIX FUNCTIONAL

ARTIFACT EVALUATED USENIX REPRODUCED

Building a System in Cloud is Easy as LEGO



But Sharing a LEGO-ed System is Challenging



Each subsystem has its own resources/pricing units

Example: DynamoDB is priced by read/write units

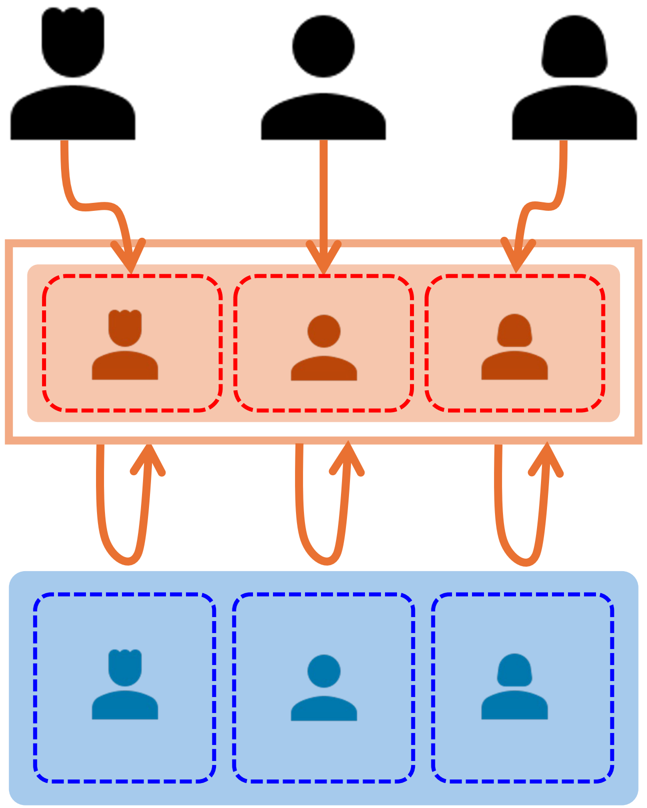
Every read/write request costs units based on request size

So many resource types to allocate!

- Redis cache size
- EC2 network bandwidth
- DynamoDB read units
- DynamoDB write units

How to fairly share all these resources?

But Sharing a LEGO-ed System is Challenging



How to fairly share all these resources?

Naïve Baseline
Equally partition every resource
Everyone gets 1/N of all resources

Fair: As if no sharing, everyone exclusively owns their portion

Suboptimal: Ignore demand heterogeneity

Can we do better?

We Know How to Do It Better… *Right?*

Jointly allocating multiple resources is not a new problem

Dominant Resource Fairness (DRF):
Equalize the share of the dominant resource type

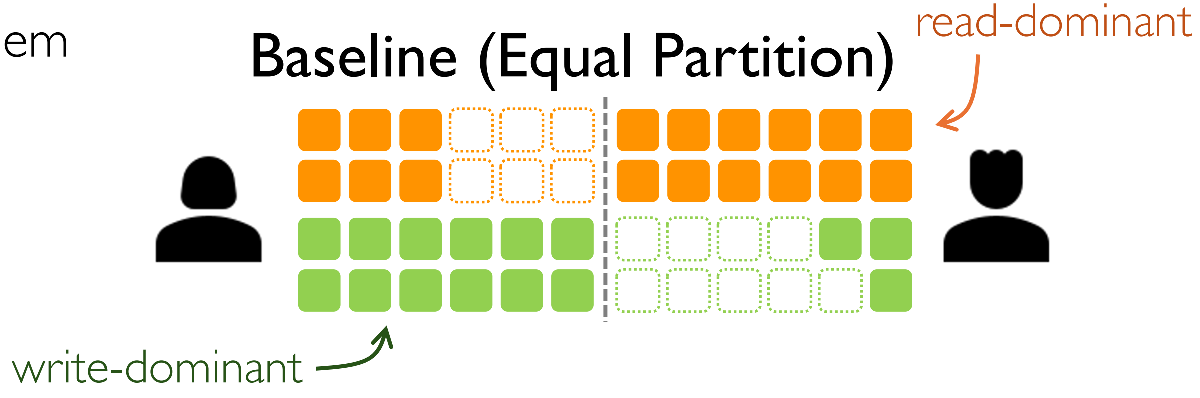
Notation: **Demand Vector**

Resources required to serve one request
(on average)

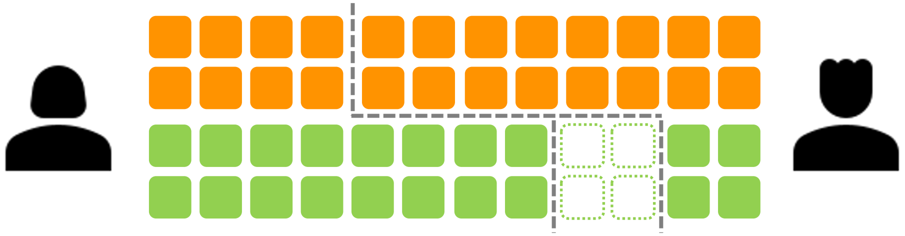
<1 Read Unit, 2 Write Units>

<4 Read Units, 1 Write Unit>

Baseline (Equal Partition)



Dominant Resource Fairness (DRF)



2/3 write units

2/3 read units

Both have 33% high throughput!

~~Problem solved?~~

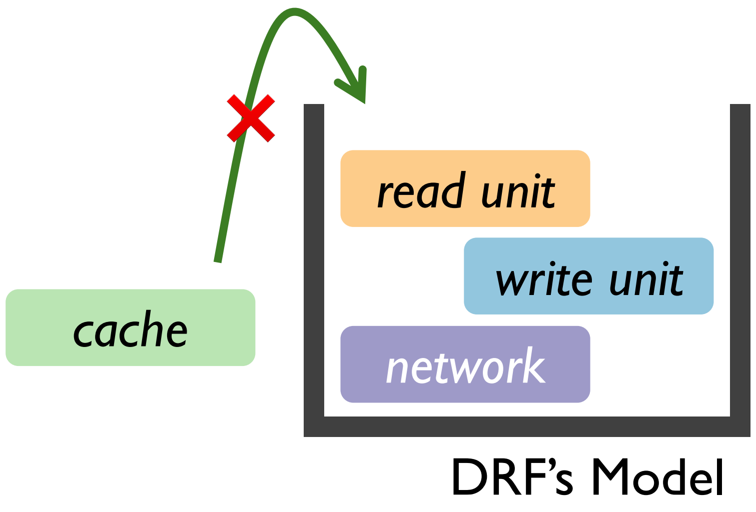
DRF Fails; It's All Cache's Fault

- Cache does not match DRF's assumption
  - DRF: throughput is linear to resource amount

    E.g., 2x throughput requires 2x every resource
  - Cache: throughput is complex & non-linear with cache size

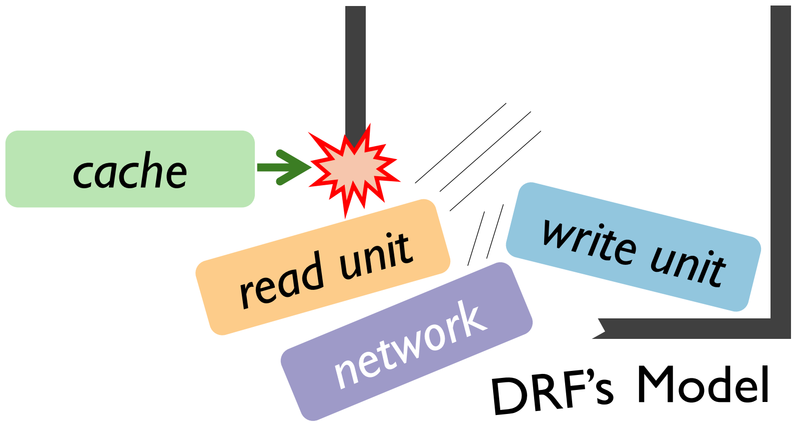
    E.g., minor cache size change may:

    ➜ huge miss ratio change ➜ huge throughput change



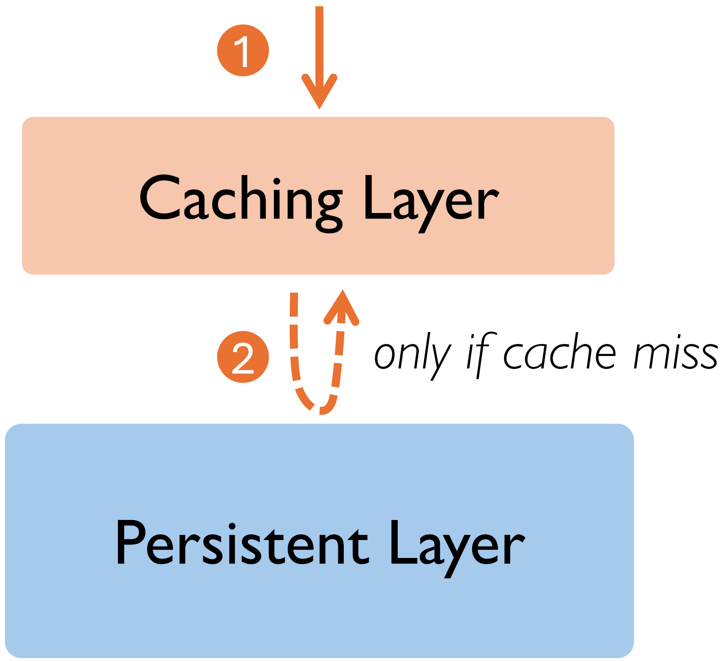
- Cache complicates other resources' allocation

(cont'd next slide)



DRF Fails; It's All Cache's Fault

- Cache complicates other resources' allocation



Cache-Correlated Resources

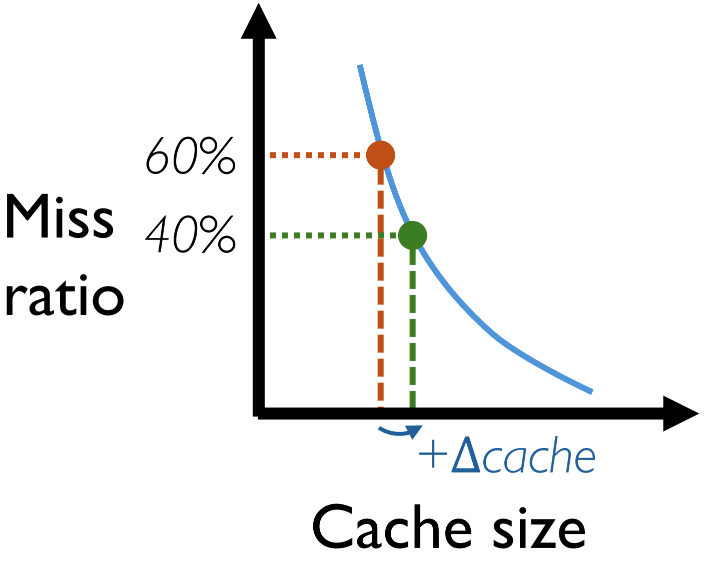
For every resource below cache,
its consumption depends on step 2's probability
miss ratio

Larger cache → Lower miss ratio → Lower demand

Demand vector is *parameterized* by cache size

Opportunity From Cache-Correlation

Miss Ratio Curve (MRC)



Target throughput: 20K req/s

60% miss ratio: *requires 12K read units*

slightly more cache ↓ ↑ *huge save on read units*

40% miss ratio: *requires 8K read units*

Where to get "*slightly more cache*"???

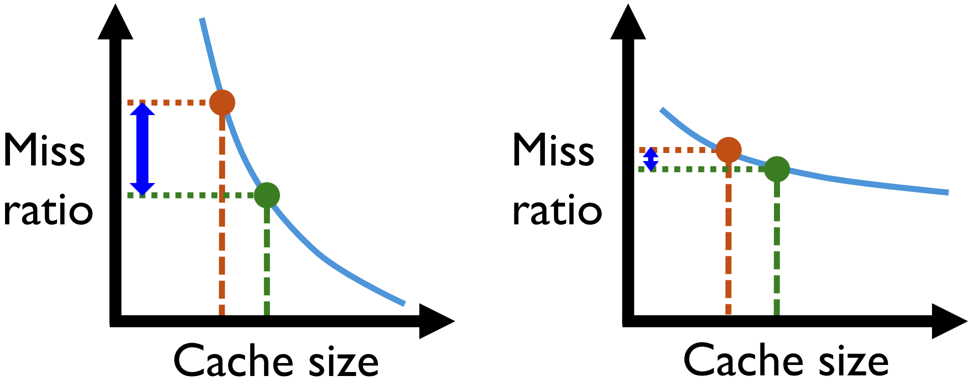
spoiler alert: *from other tenants*

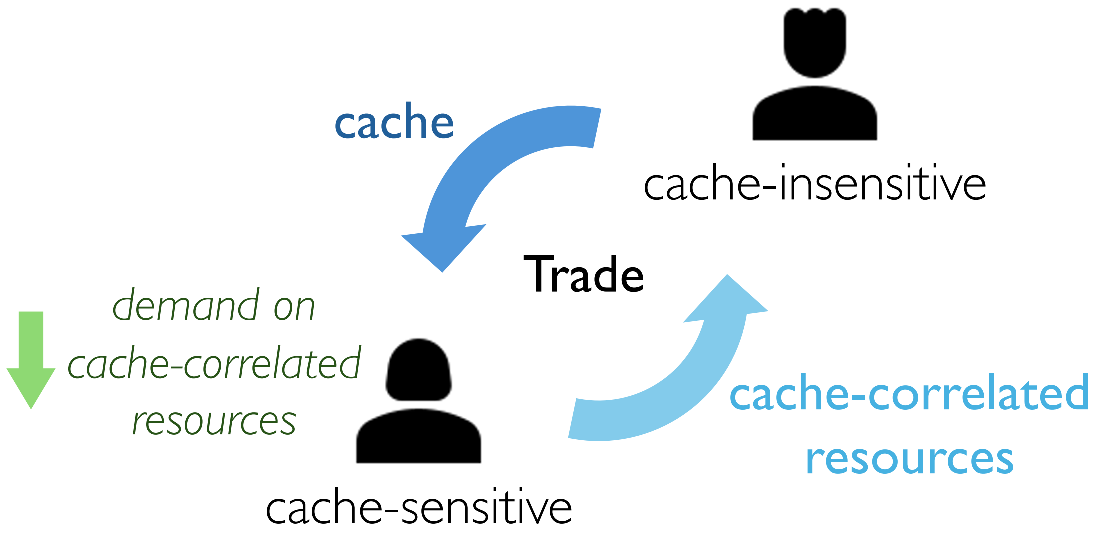
Exploit Heterogeneous Cache Sensitivity

Opportunity: Exploit cache correlation & sensitivity across tenants

Observation:
Not every tenant benefits from cache equally

Cache-sensitive workload | Cache-insensitive workload

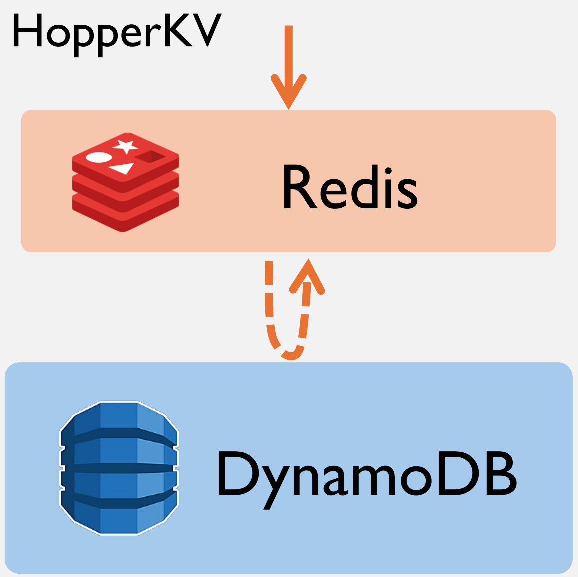




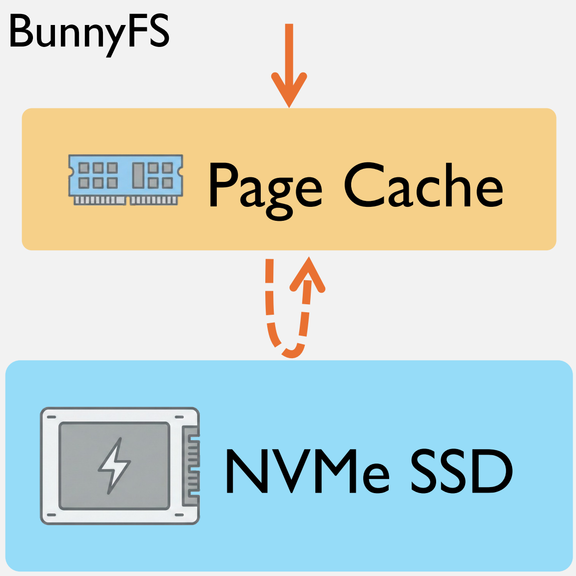
→ Everyone gets higher throughput!

Questions: How much to trade? What is a fair deal?

Contributions



cloud-native KV store



semi-microkernel filesystem

Algorithm: HARE
universal & abstract model

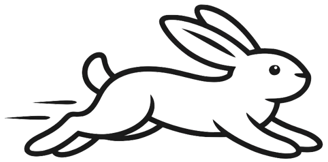
Systems: HopperKV & BunnyFS
case studies to demonstrate HARE's generality

Outline

- HARE Algorithm
  Universal Model: *statistics in, allocation out*
- HopperKV System
  Practical System: *collect statistics and allocate resources dynamically & robustly*
- Evaluation
  Results: *scalable & adaptive; up to 1.9x higher throughput*
- Conclusion

Outline

- HARE Algorithm
  Universal Model: *statistics in, allocation out*



- HopperKV System
  Practical System: *collect statistics and allocate resources dynamically & robustly*
- Evaluation
  Results: *scalable & adaptive; up to 1.9x higher throughput*
- Conclusion

Harvest-Redistribute (HARE) Overview

0. Initialization
   Start with baseline

1. Harvest Phase
   Optimize cache partition iteratively

2. Redistribute Phase
   Allocate rest of resources to improve throughput

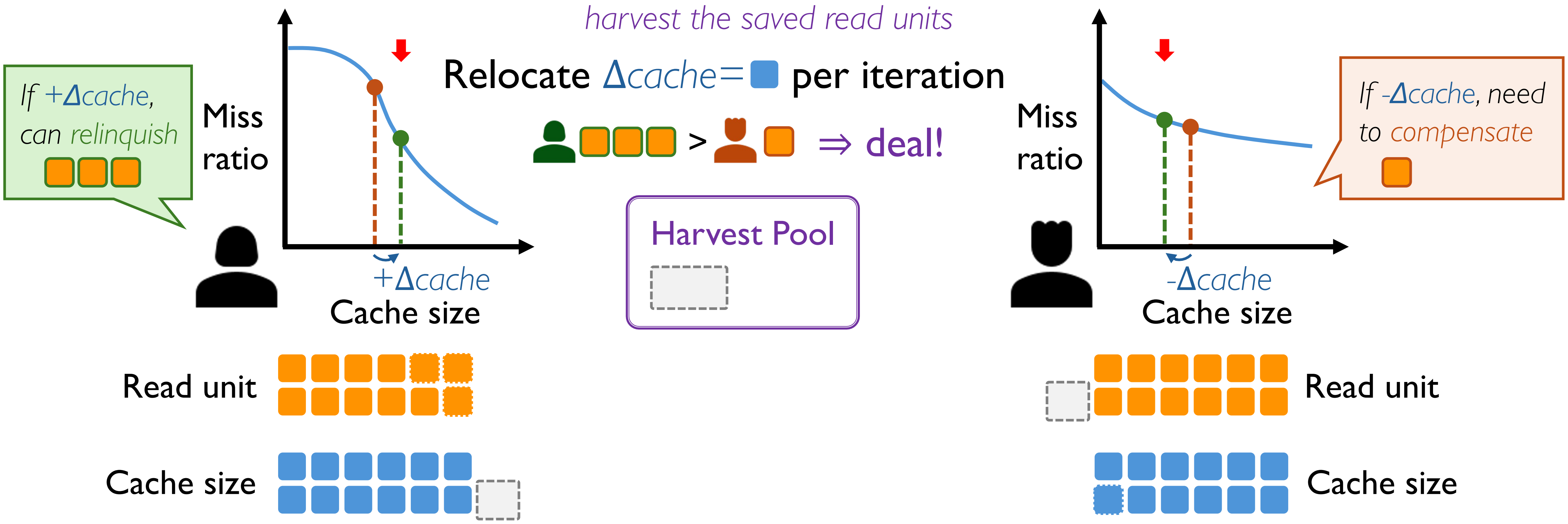
HARE: Initialization

Every tenant is allocated the baseline resource (equally partitioned)

Simplified example with only two resources: <u>cache</u> and <u>backend DB read units</u>

HARE: Harvest Phase

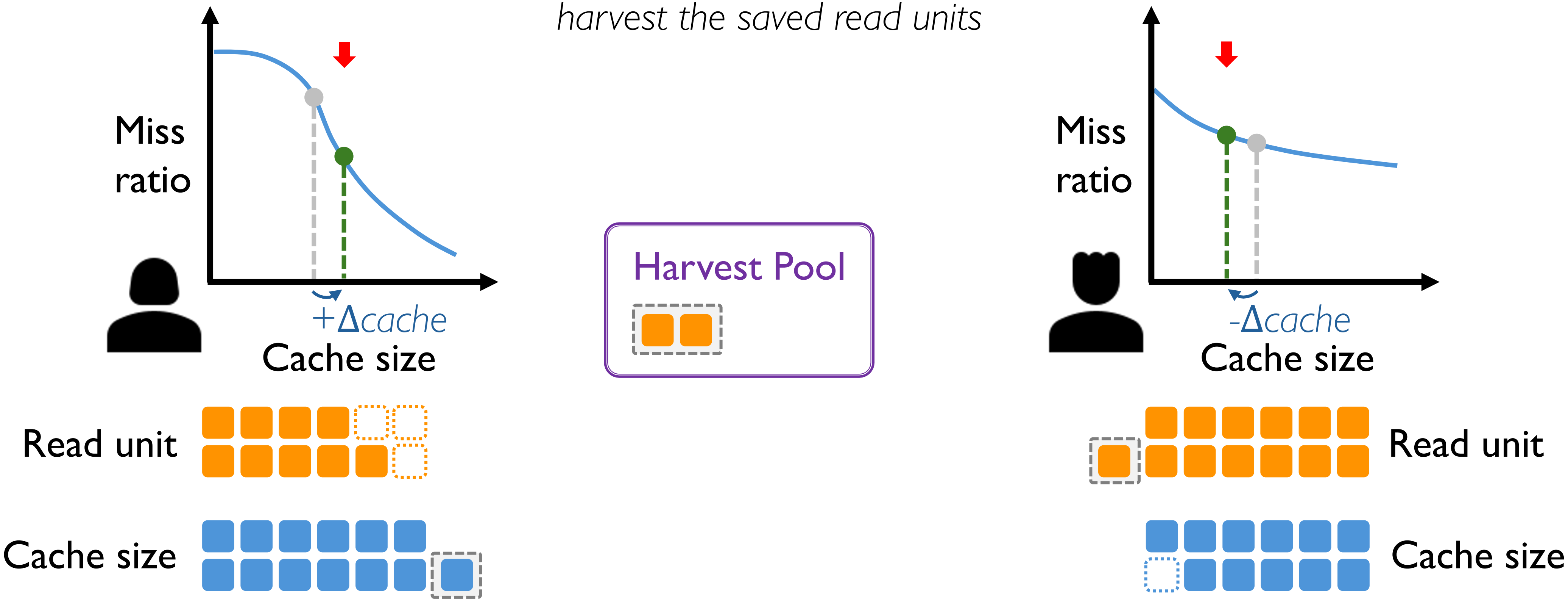
Iteratively relocate cache among tenants, such that all tenants maintain baseline throughput, but consume fewer read units in total



HARE: Harvest Phase

Iteratively relocate cache among tenants, such that all tenants maintain baseline throughput, but consume fewer read units in total

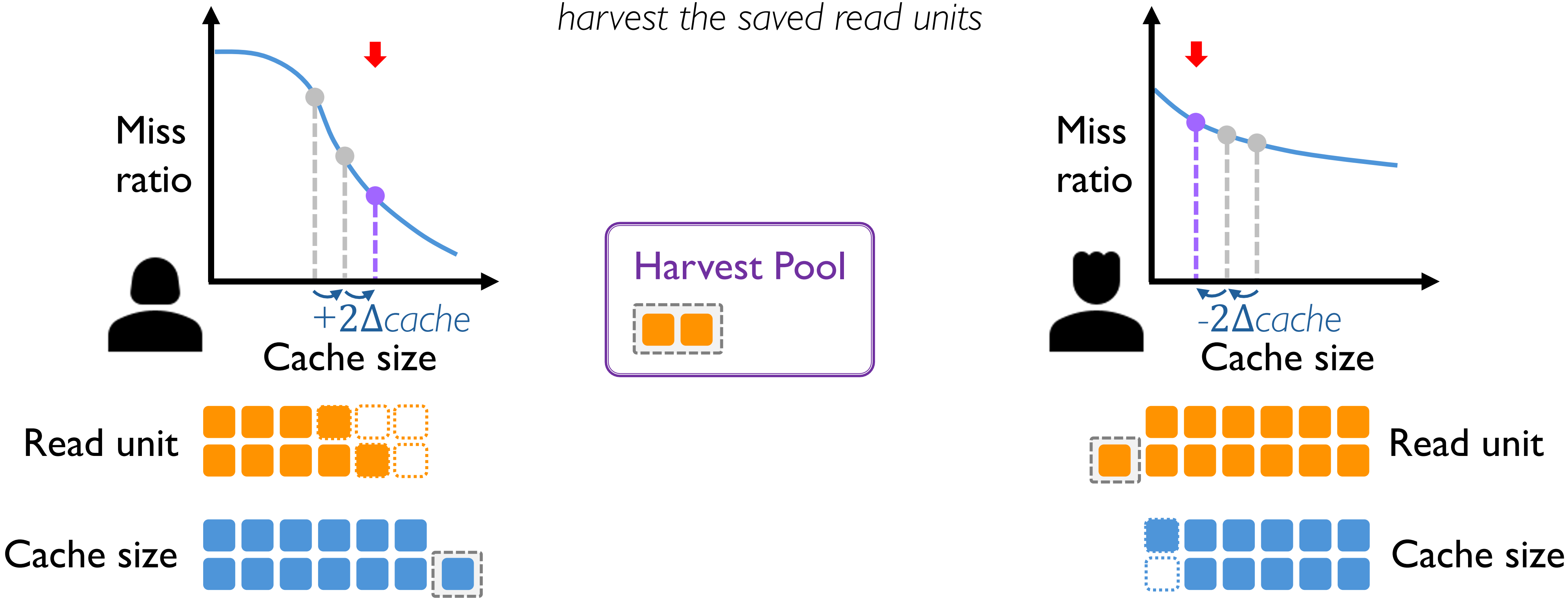
harvest the saved read units



HARE: Harvest Phase

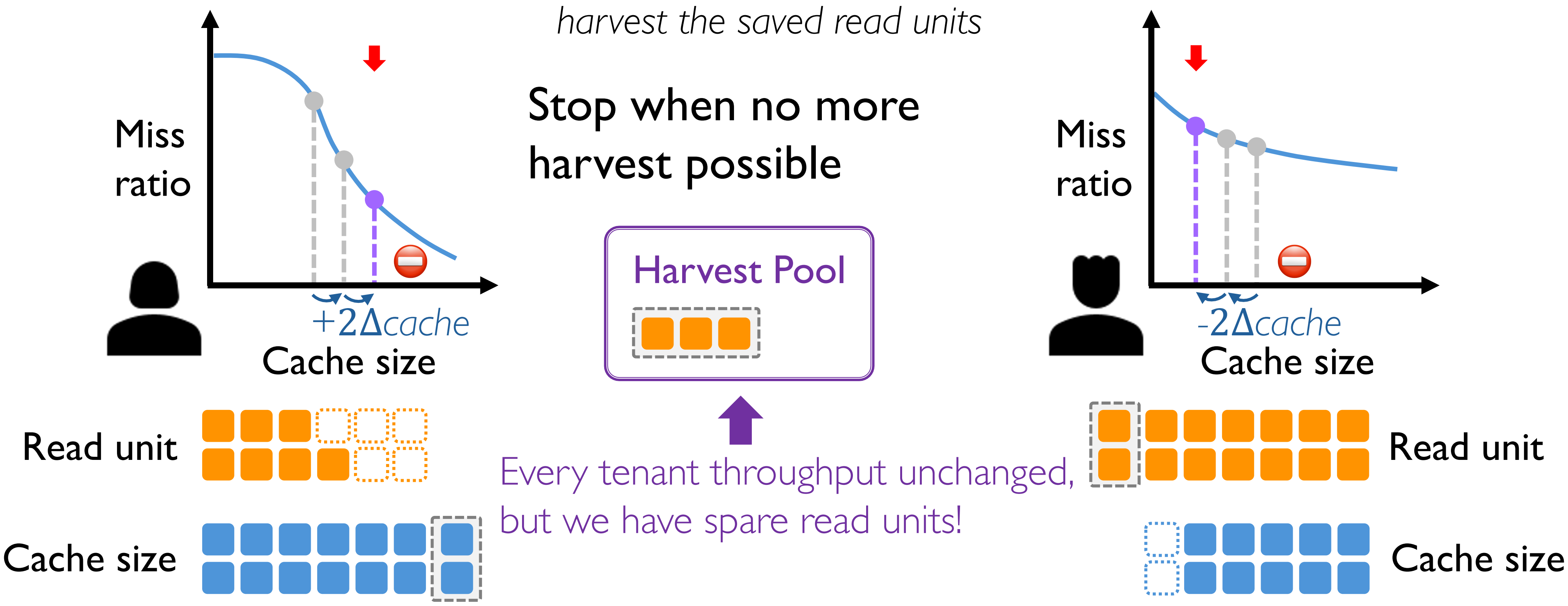
Iteratively relocate cache among tenants, such that all tenants maintain baseline throughput, but <u>consume fewer read units in total</u>

harvest the saved read units



HARE: Harvest Phase

Iteratively relocate cache among tenants, such that all tenants maintain baseline throughput, but consume fewer read units in total

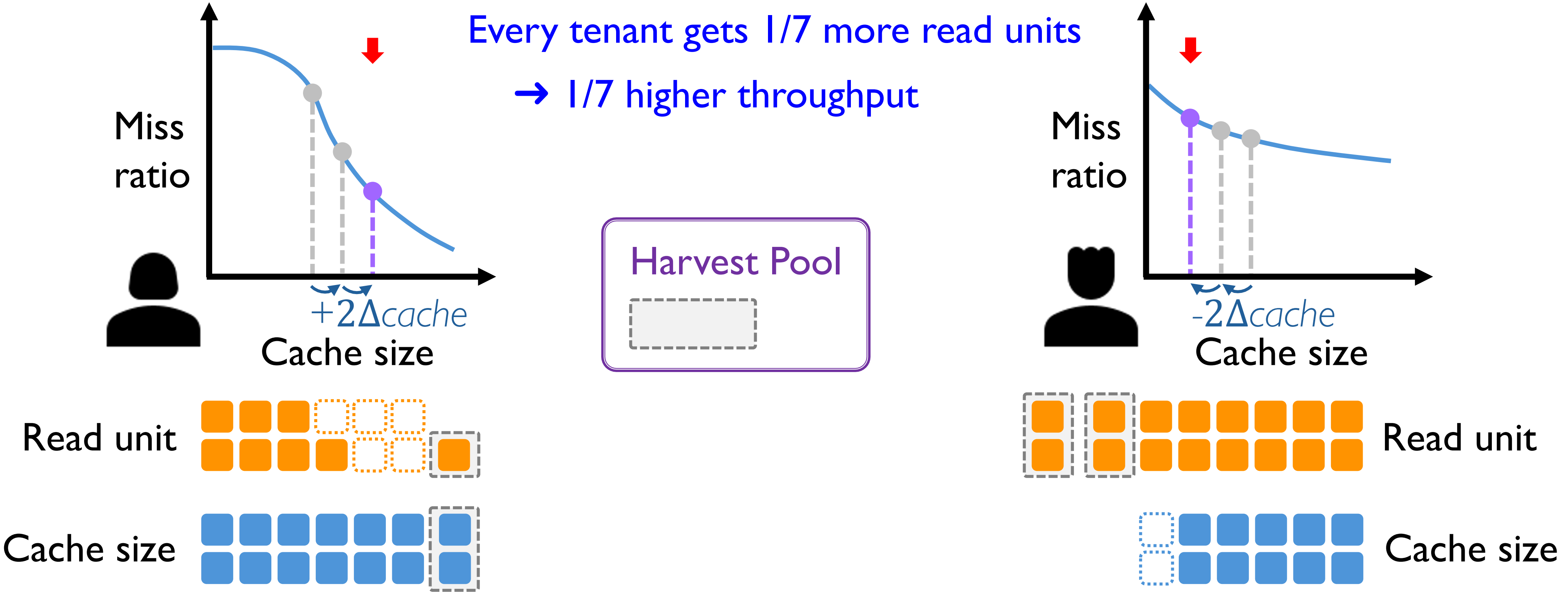


HARE: Redistribute Phase

Allocate the harvested read units, weighed by the amount each tenant already owns

HARE: Redistribute Phase

Allocate the harvested read units, weighed by the amount each tenant already owns



HARE also supports multiple cache-correlated resources
that fully generalizes DRF *(more details in paper)*

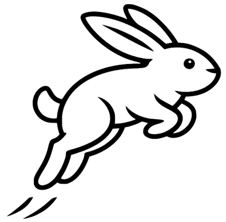
Outline

- HARE Algorithm

  Universal Model: *statistics in, allocation out*

- HopperKV System

  Practical System: *collect statistics and allocate resources dynamically & robustly*



- Evaluation

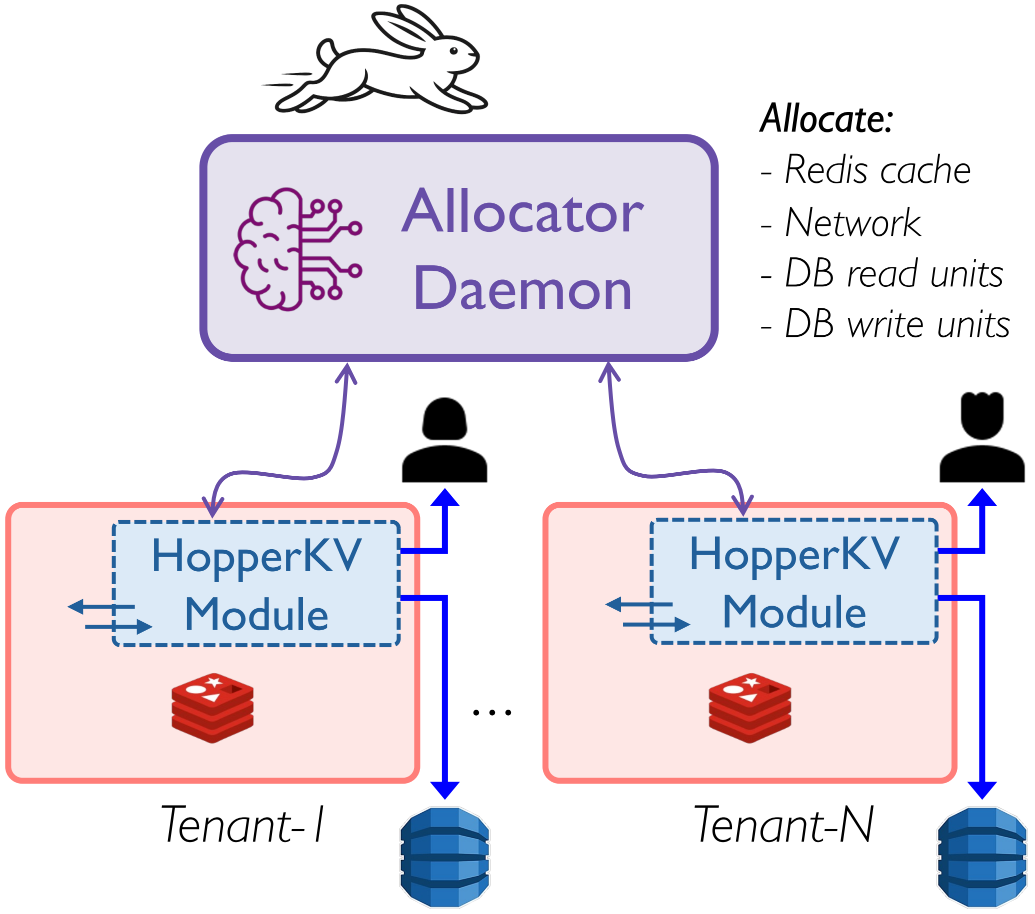
  Results: *scalable & adaptive; up to 1.9x higher throughput*

- Conclusion

HopperKV Architecture

HopperKV consists of

- HopperKV Redis Module
  - Load into unmodified Redis at runtime
  - Execute data plane GET/SET operations
  - Collects tenant workload statistics
- Allocator Daemon
  - Collect all tenants workload statistics from Redis modules
  - Adjust the resource budget of each tenant's Redis instance based on HARE



Each tenant gets a dedicated Redis instance

HopperKV Practical Challenges & Solutions

- **How to know tenant miss ratio curve (MRC)?**

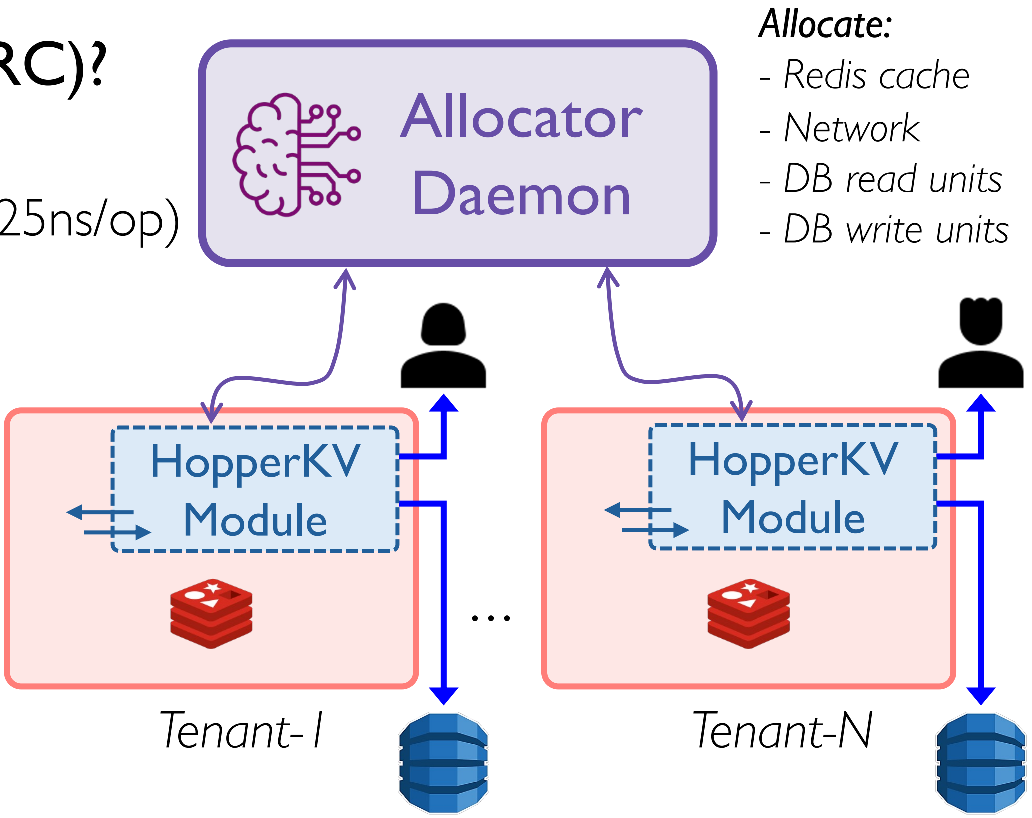
  Spatially sampled ghost cache
  Emulated LRU w/ sampled metadata; lightweight (~25ns/op)

- **How to handle dynamic workloads?**

  Run a new instance of HARE periodically
  Adjust allocation if new one is significantly better

- **How to tolerate noise and inaccuracy?**

  MRC salting technique
  Ensure safe margins of relative error



Each tenant gets a dedicated Redis instance

(more details in paper)

Outline

- HARE Algorithm

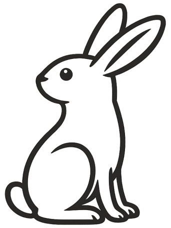
  Universal Model: *statistics in, allocation out*

- HopperKV System

  Practical System: *collect statistics and allocate resources dynamically & robustly*

- Evaluation

  Results: *scalable & adaptive; up to 1.9x higher throughput*



- Conclusion

Evaluation

- When HopperKV achieves benefits over alternatives?

  Microbenchmark: a variety of sharing cases between two tenants

- Can HopperKV scales?

  Scaling Macrobenchmark: up to 16 tenants

- Can HopperKV handle workloads varying over time?

  Dynamic Macrobenchmark: workload changes every minute

- How HopperKV performs under real-world workloads?

  Trace-Replay Macrobenchmark: based on Twitter production traces

(more details in paper)

Experiment Setup

Four tenant workloads:

- YCSB-A: 50% read, 50% write
- YCSB-B: 95% read, 5% write
- YCSB-C: 100% read
- YCSB-E: 95% scan, 5% write

Working set: 6M keys (~3.3GB)

Resources budget (per-tenant):

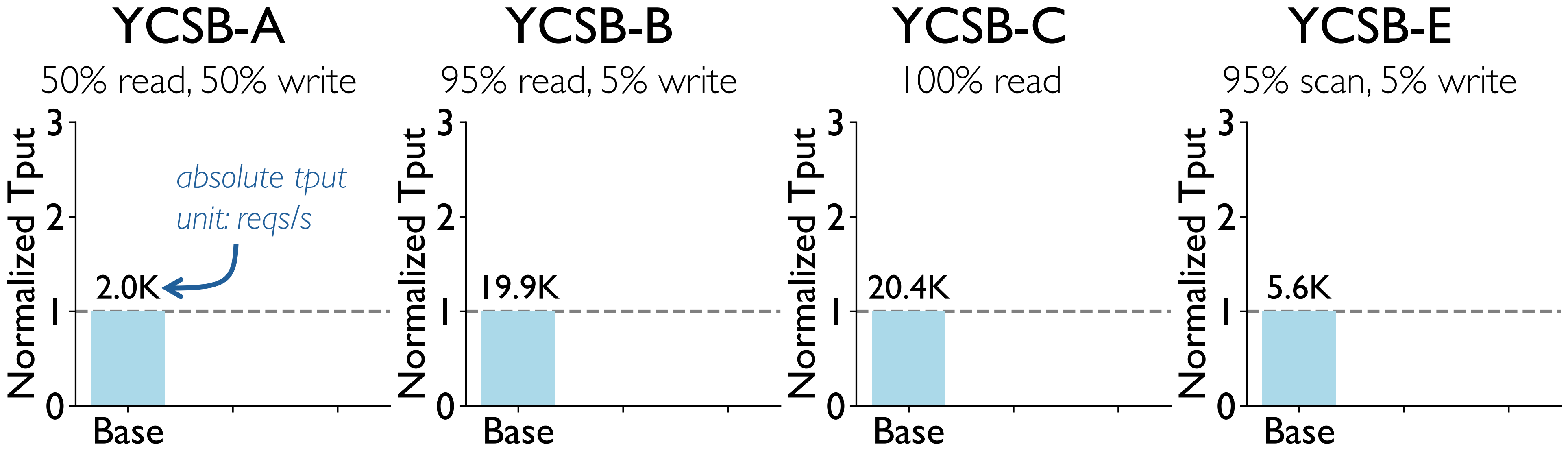
- 2GB cache
- 1K read units/s
- 1K write units/s
- 50MB/s network

Comparison:

- **Baseline:** Equally partition all resources
- **DRF:** Equally partition cache, use DRF for others
- **HARE:** Jointly allocate all resources

(more comparison in paper)

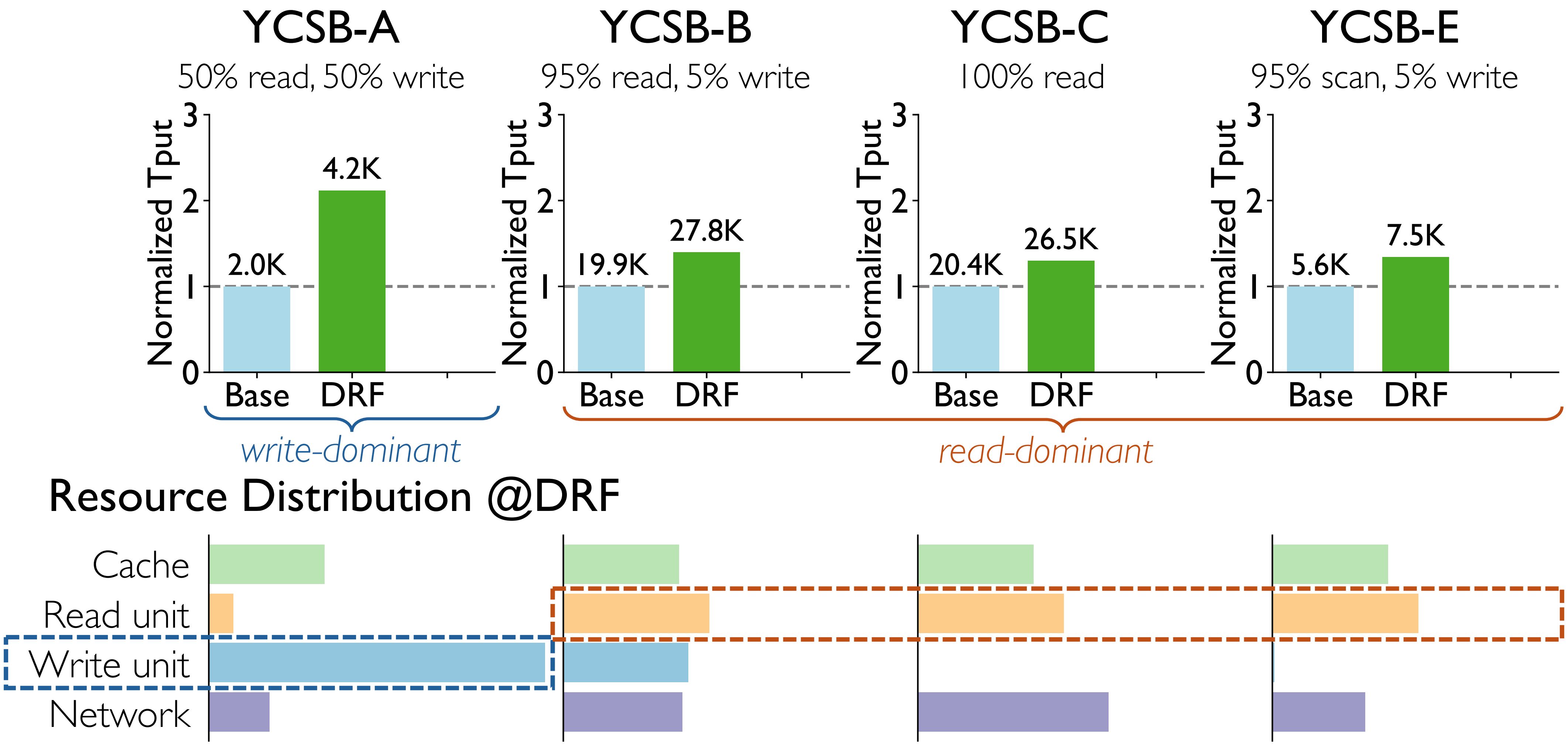
Experiment Results



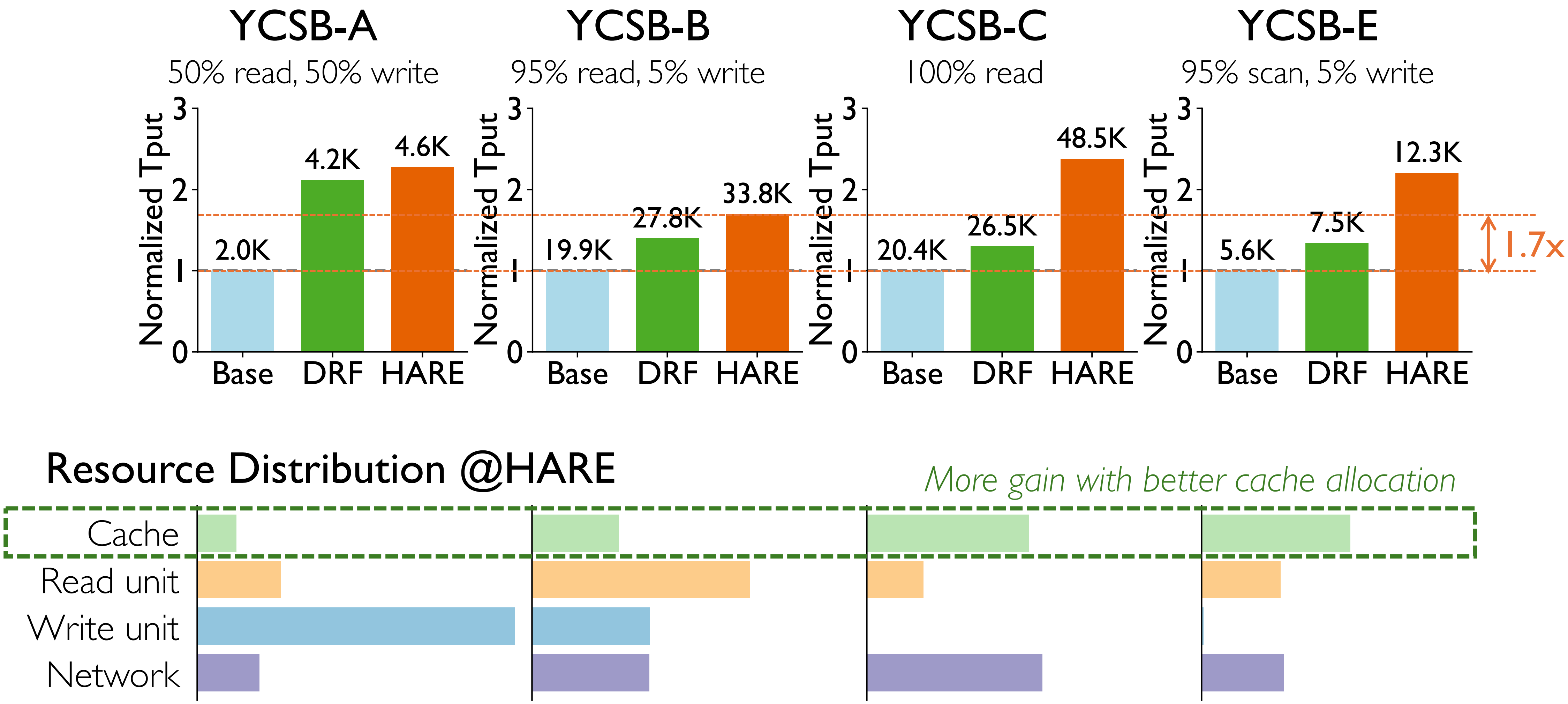
Resource Distribution @Baseline

Equally partition every resource

Experiment Results



Experiment Results



Other Interesting Takeaways in Paper

- DRF: no gain when all tenants have same dominant resource type

  HARE opens up more opportunities with an optimized cache partition

- Existing multi-tenant cache (e.g., Memshare): hurt one tenant to benefit others (unless w/ MRC plateau)

  HARE guarantees no degradation for all tenants

Outline

- HARE Algorithm

  Universal Model: *statistics in, allocation out*

- HopperKV System

  Practical System: *collect statistics and allocate resources dynamically & robustly*

- Evaluation

  Results: *scalable & adaptive; up to 1.9x higher throughput*

- Conclusion

Conclusion

Formulate cache-centric multiple-resource allocation problem

- **HARE**: *Generic algorithm for holistic fair allocation*

  Optimized cache partition for better overall resource efficiency

- **HopperKV & BunnyFS**: *Practical systems utilizing HARE*

  Different contexts, same principal

Call for a holistic view for resource management

- Capture interaction among different resource types
- Exploit for performance and fairness