WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON

Layout-Aware Data Organization For Heterogeneous Hierarchies

Vinay Banakar

February 20th, 2026

PhD Defense

Committee: Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Kimberly Keeton, Michael Swift, Shivaram Venkataraman, Dimitris Papailiopoulos

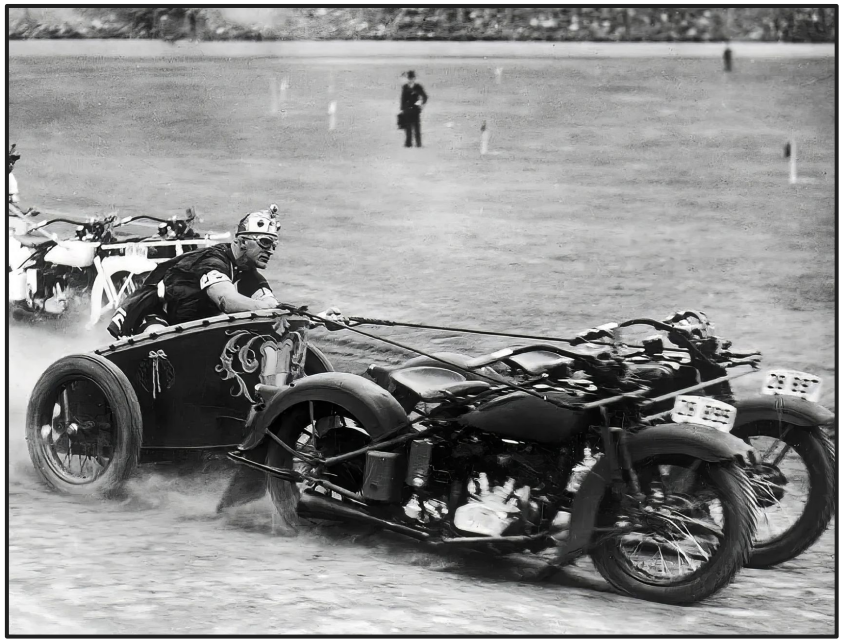
PAST

Abstraction is both the most powerful and the most dangerous concept in computer science.

For centuries, people have built abstractions that ignore some properties of an object so as to focus on the important ones.
For example, racing long ago required you to know only certain important things, such as whether the horse moved and if the chariot would be strong enough, etc.

PAST

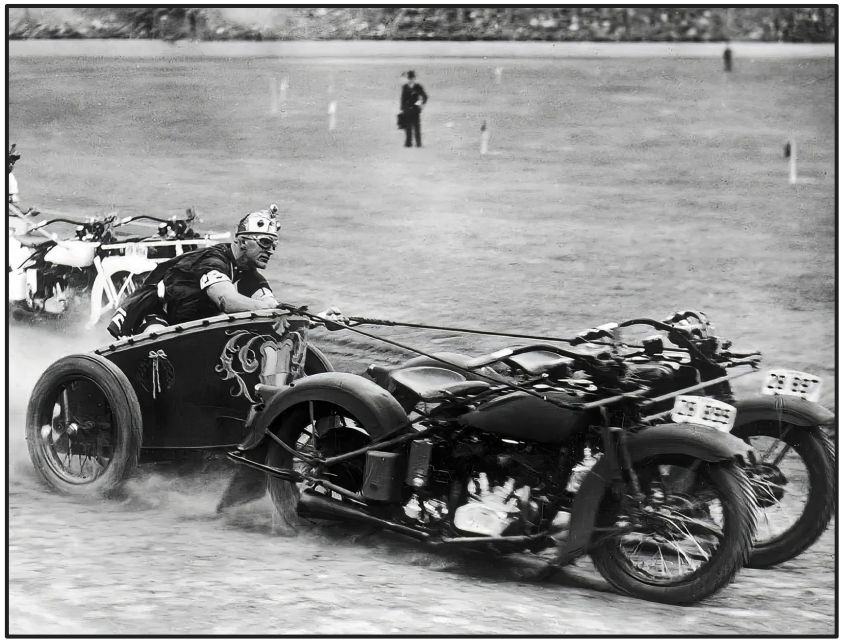
PRESENT



But when new objects and new trends arrive, breaking away from the well-established abstraction is hard.
While some of the abstractions still apply, treating bikes as horses loses essential information.

PAST

PRESENT



FUTURE?

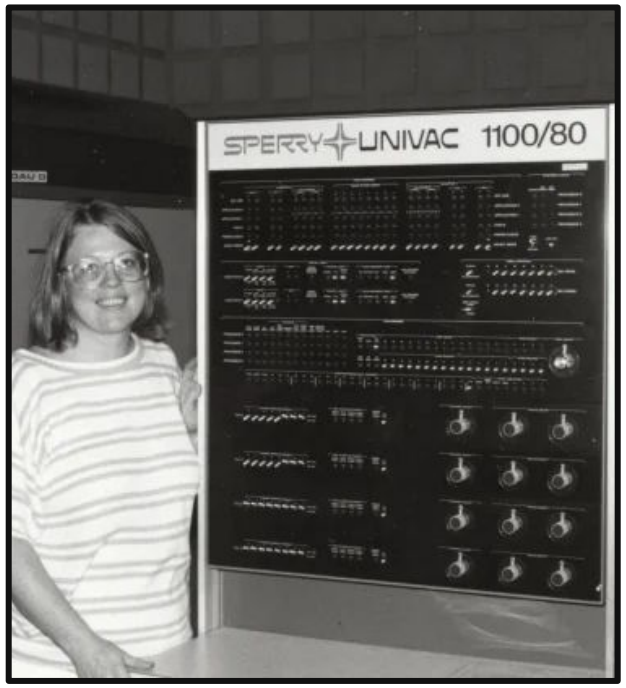
So, rethinking the abstraction from the ground up or adapting the abstraction for modern needs can increase both performance and efficiency.

OS Page is a foundational abstraction

- Convenient: isolation, virtualization, demand paging, etc

Atlas supercomputer, 1962

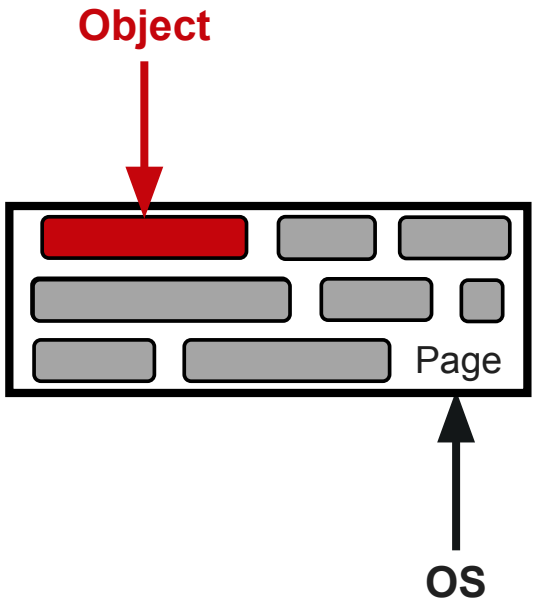
Glen Culler w/ IBM 360, 1966



UNIVAC 1100 at UW-Madison, 1970

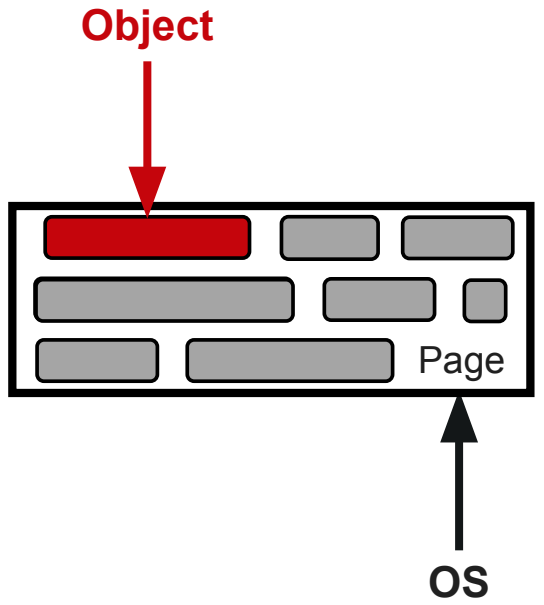
OS Page is a leaky abstraction

- Convenient: isolation, virtualization, demand paging, etc
- Page granularity fixes OS's observability and operability
- Applications operate on objects, fields, or records



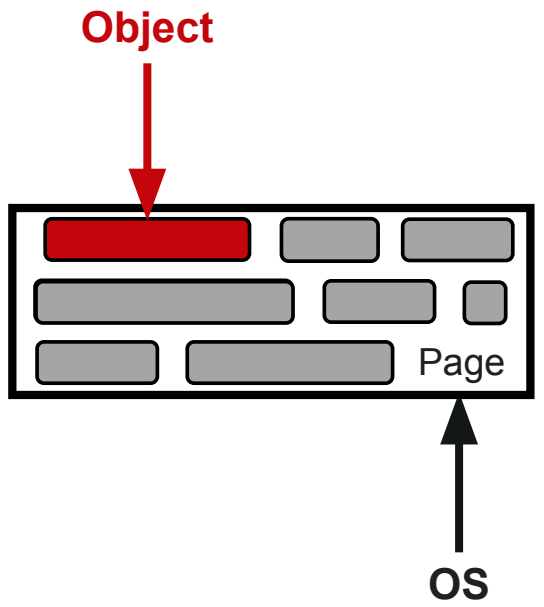
OS Page is a leaky abstraction

- Convenient: isolation, virtualization, demand paging, etc
- Page granularity fixes OS's observability and operability
- Applications operate on objects, fields, or records
- **Leaky:** forces coarse-grained decisions on fine-grained data



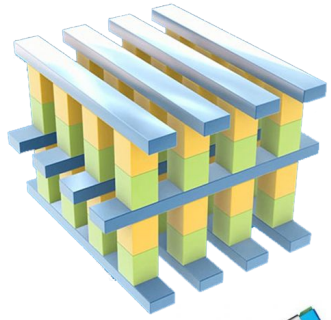
OS Page is a leaky abstraction

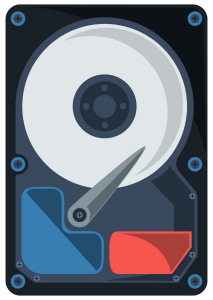
- Convenient: isolation, virtualization, demand paging, etc
- Page granularity fixes OS's observability and operability
- Applications operate on objects, fields, or records
- **Leaky:** forces coarse-grained decisions on fine-grained data

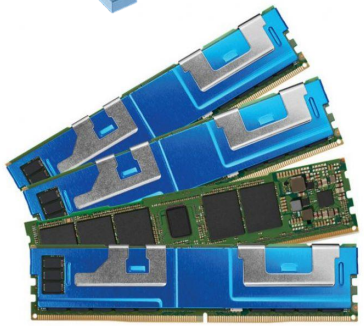


Semantic gap between application and OS memory model

Trend #1: Storage is not the same anymore







Block addressed	Byte addressed
Good sequential bandwidth	Good random bandwidth
Symmetric read-write performance	Asymmetric read-write performance
No concurrency complexity	Concurrency complexity

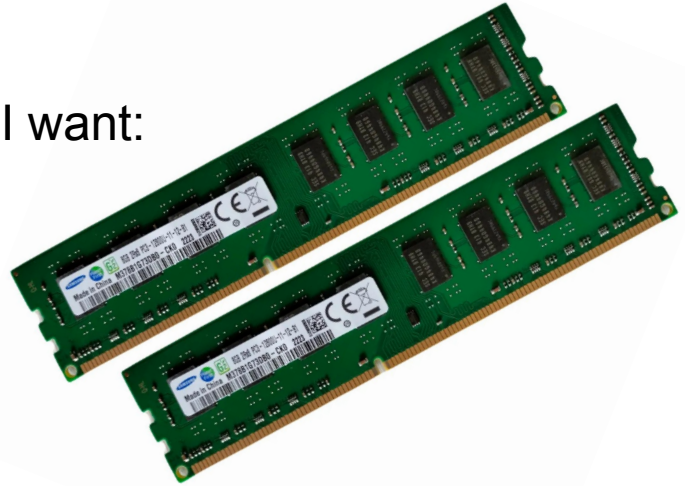
Trend #2: Memory is expensive

50% of server costs at Azure and Meta[1]

$ per bit flatlined for **10** years [2]

DDR5 costs **265%** more [2]

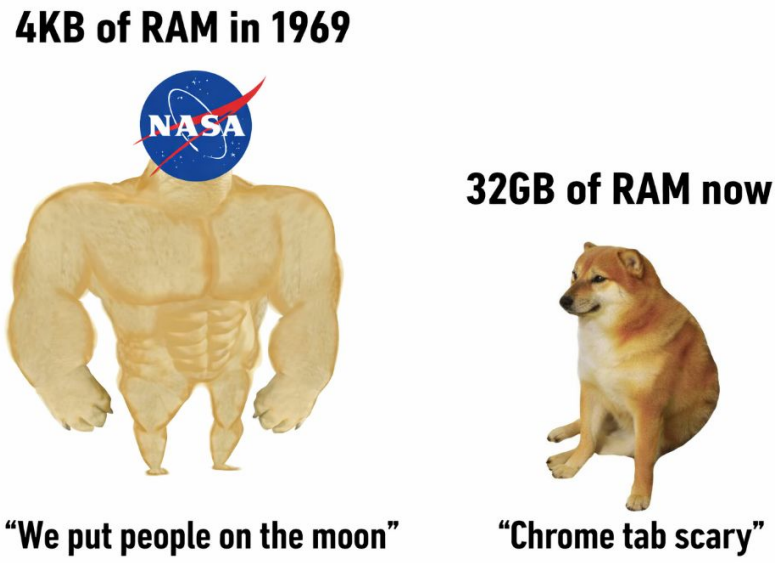
the RAM I want:



the RAM I can afford:

[1] Pond: CXL-Based Memory Pooling Systems for Cloud Platforms
[2] DRAMeXchange (6 months -Feb 2026)

Trend #3: Memory is constrained and wasted



65% used at Google [1] and Alibaba [2]

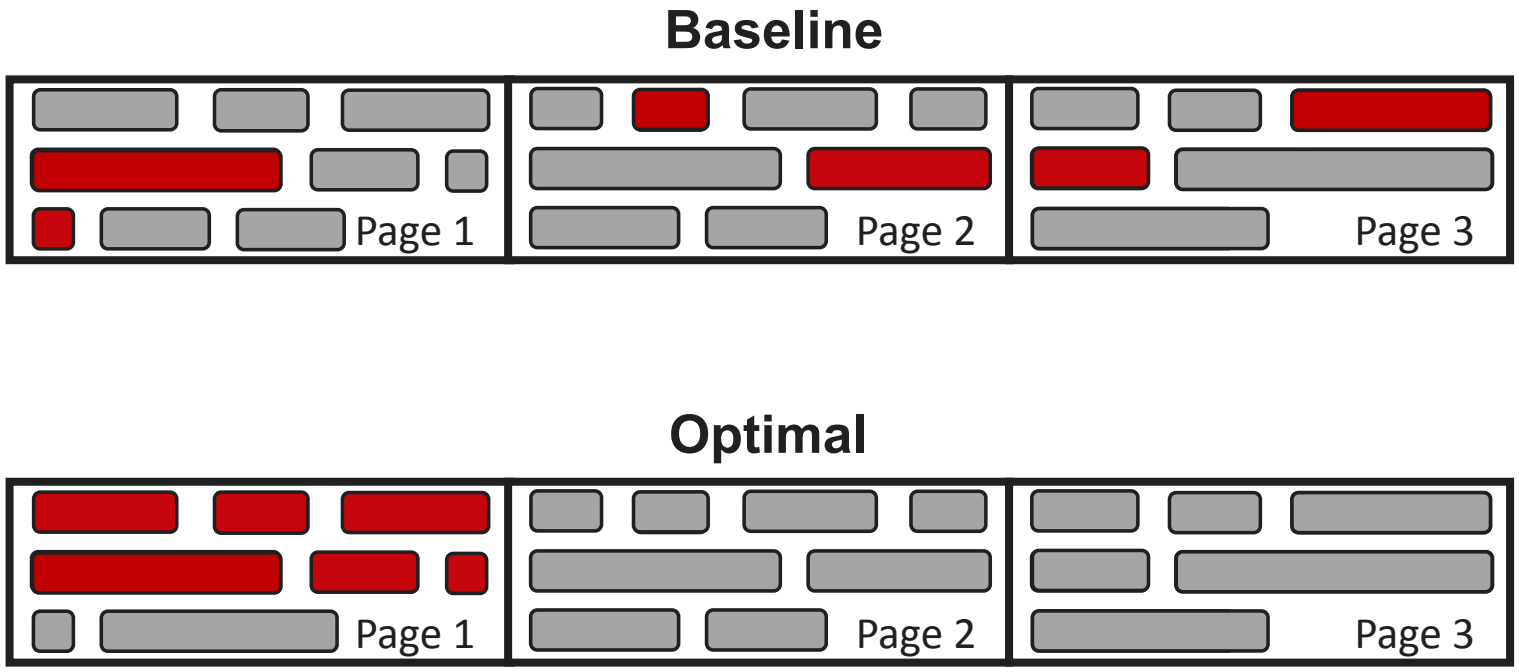
50% of accesses touch **1%** data at Alibaba [4]

95-98% used at Meta [3]

<3% data touched in 24 hours at Meta [5]

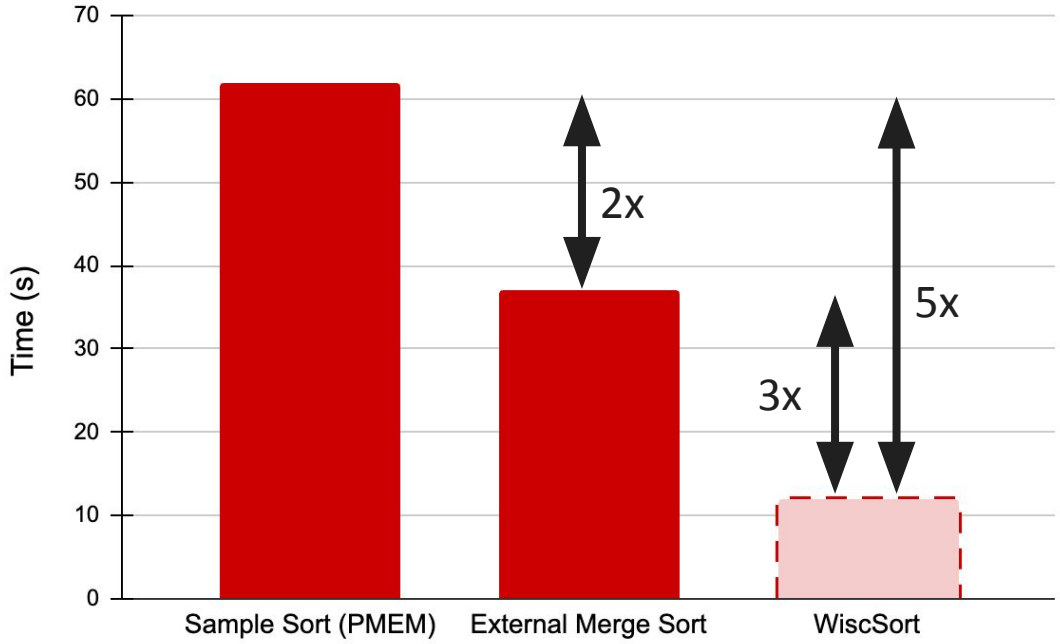
[1] Borg: the Next Generation
[2] Imbalance in the Cloud: an Analysis on Alibaba Cluster Trace
[3] TPP: Transparent Page Placement for CXL Tiered-Memory
[4] HotRing: A Hotspot-Aware In-Memory Key-Value Store
[5] Benchmarking RocksDB KV Workloads at Facebook

Layout Optimization is key to overcome leaky page abstraction



Static Layout Optimization

Applications designed to take advantage of the new storage devices

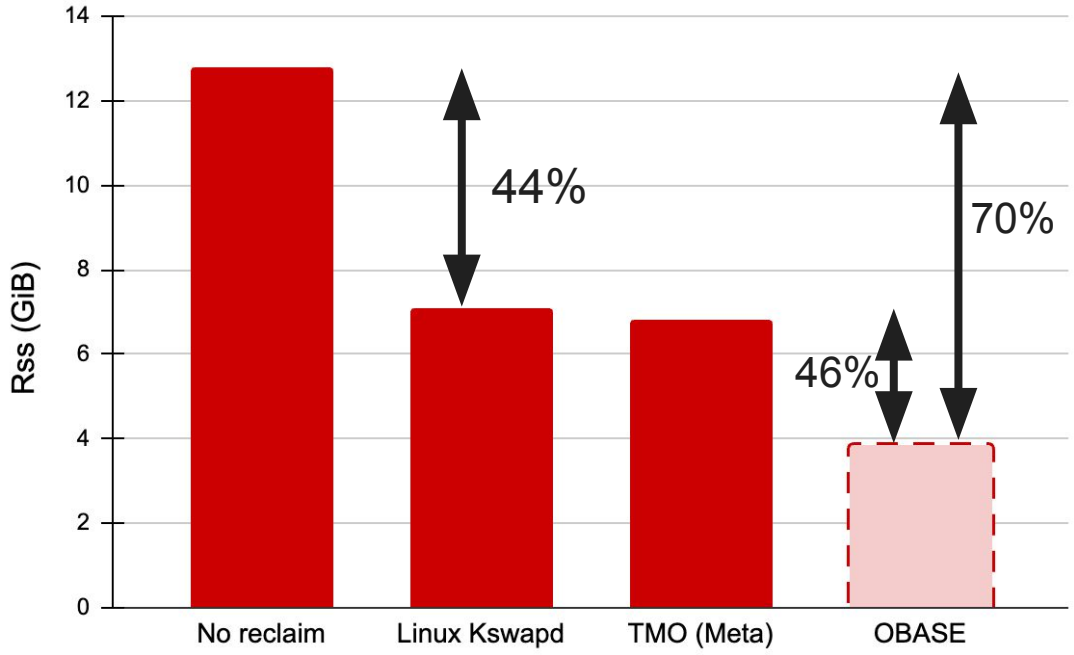


WiscSort: External Sorting For Byte-Addressable Storage

VLDB 2023

Dynamic Layout Optimization

Transparently improve memory efficiency of applications for skewed workloads



OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering

SOSP DIMES 2025 | Under review at OSDI 2026

Outline

Static Layout Optimization Redesign applications for modern storage

WiscSort: External Sorting For Byte-Addressable Storage

- How to sort on BAS?
- The BRAID model
- Key Value separation
- Thread-pool controller and interference-aware scheduler
- Evaluation

Case for Dynamic Layout Access semantics aren't always known

- Hotness Fragmentation
- Transient Object Hotness
- Rewards of Improved Page Utilization
- Address-Space-Engineering

Dynamic Layout Optimization Transparently improve application memory efficiency

OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering

- Challenges of Address-Space-Engineering
- Decoupling Layout from Tiering
- Grouping Objects by Access Intensity
- Enabling Object Mobility
- Evaluation

Conclusion

Redesign data intensive applications for modern storage

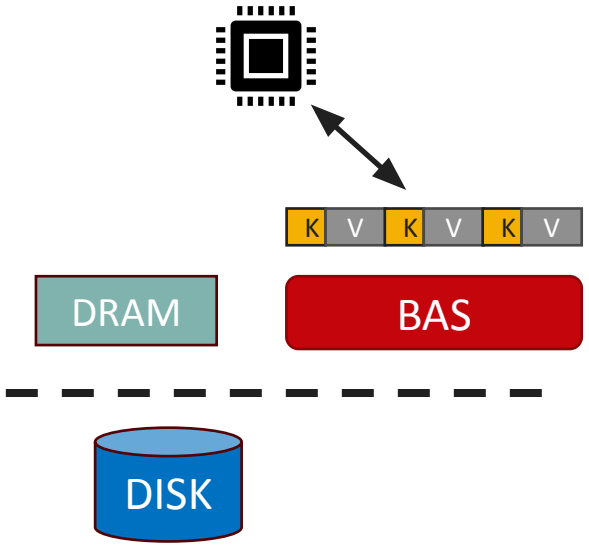
WiscSort - Overview

- Data intensive case study: Sorting
- New model (**BRAID**) to represent byte-addressable storage characteristics
- New sorting mechanisms with the help of **BRAID** for peak performance
- A highly concurrent external sort that utilizes these mechanisms (**WiscSort**)
- Evaluate on industry standard sorting workloads
  2-7x faster than competing state-of-the-art approaches

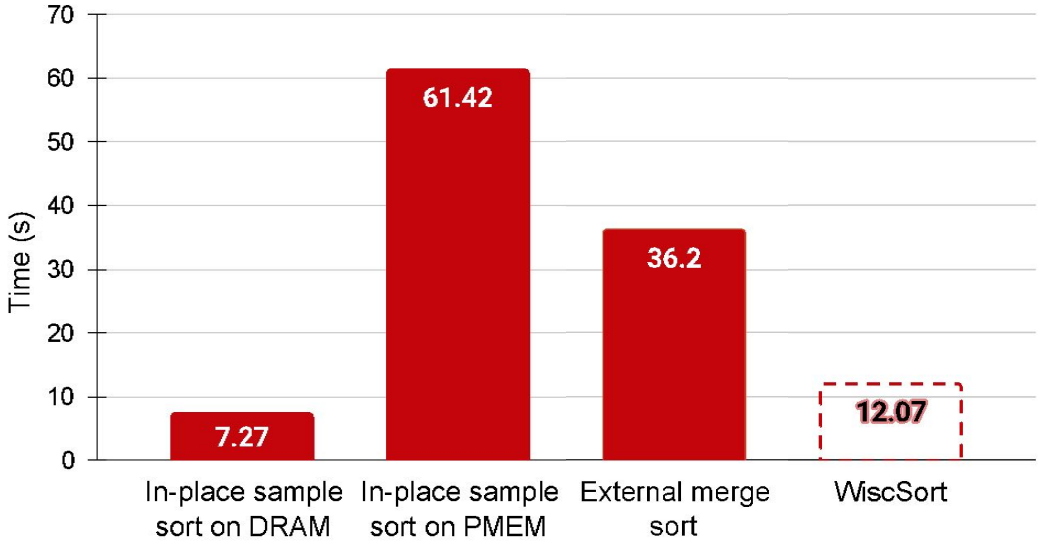
How to sort on Byte-Addressable Storage (BAS)

Web indexing, key-value stores, data analytics, and relational databases
50% of all computer time is spent on sorting and searching [1]

As a slower DRAM

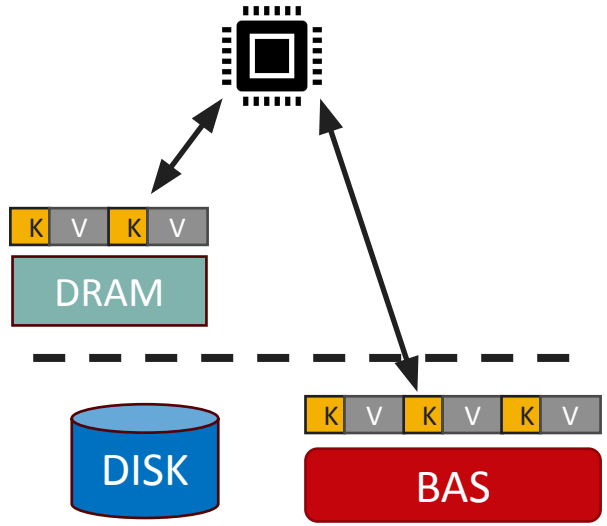


In memory sort



20GB workload containing 200M records of 10B keys and 90B values on Intel Optane DC PMEM

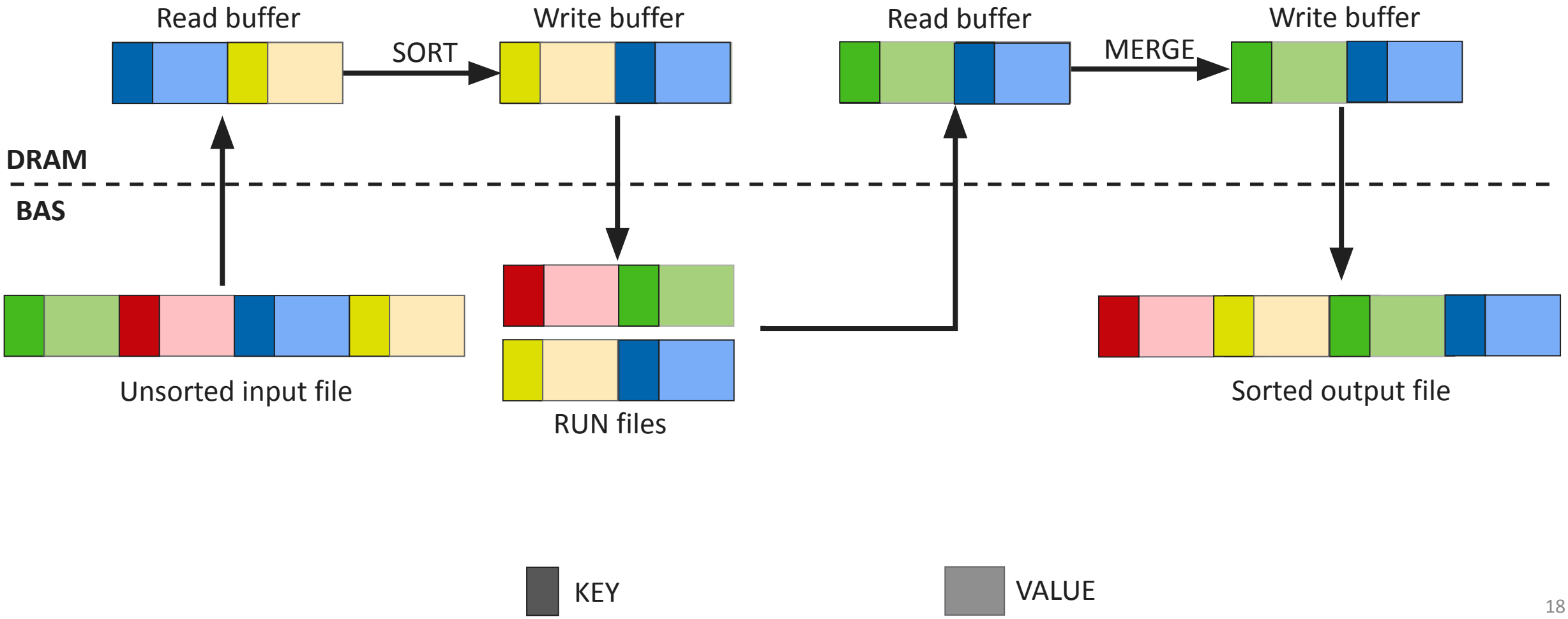
As a faster disk



External sort

[1] Robust and Efficient Sorting with Offset-Value Coding, ACM Transactions on Database Systems 2022

Traditional external merge sort



The BRAID model

Property	Description	Sorting goals
Byte Addressability (**B**)	Granular access (64B, 256B)	**Read only keys**
Higher Random-Read performance (**R**)	Random read is on par with sequential read	**Random reads of keys and values**
Asymmetric Read-Write Cost (**A**)	Reads can be 4x faster than writes	**Limit or avoid writing RUN files**
Read-Write Interference (**I**)	Read bandwidth degrades up to 2x when there are overlapping writes	**Separate concurrent reader and writer threads**
Device-Constrained Concurrency (**D**)	Max concurrent writes can be 2x slower than peak write performance	**Manage read and write concurrency differently**

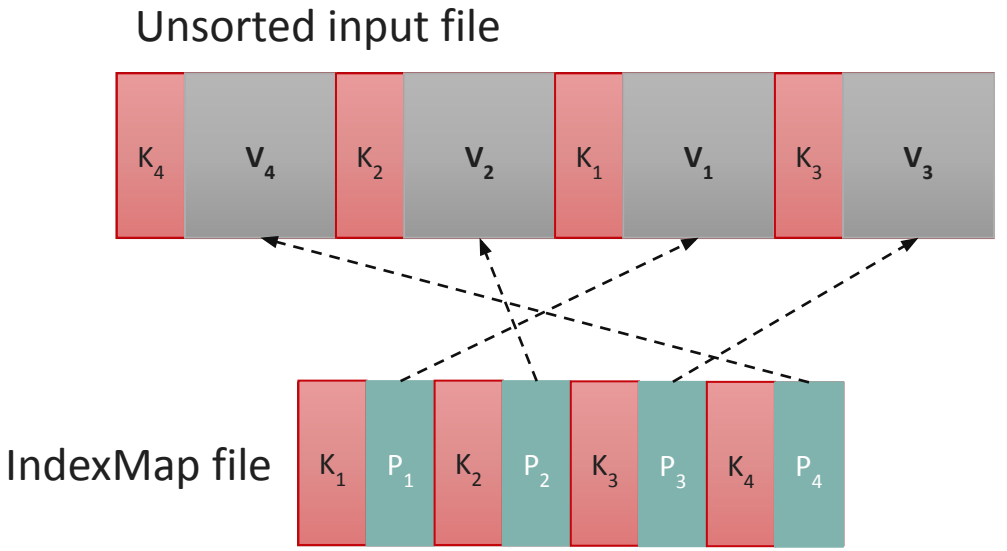
WiscSort

A high-performance external sorting system for **BRAID** devices

1. Key and value separation (**BRA**)
2. Interference-aware scheduler (**I**)
3. Thread-pool controller (**D**)

(BRAID)
WiscSort: Key value separation

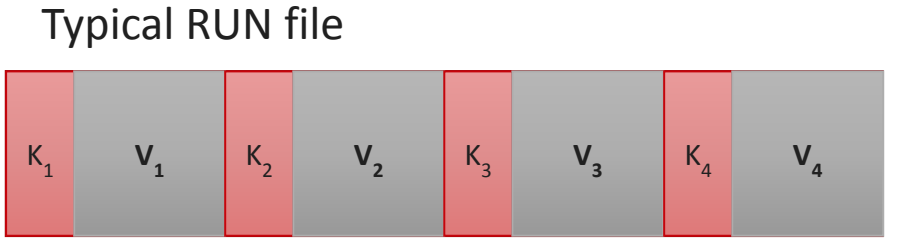
- Two phases: RUN and MERGE
- Workloads have small keys and large values [2]
- Strided reads of keys to generate IndexMap
- Reduced writes to BAS device during RUN phase
- Random reads to gather values later



Two variants:

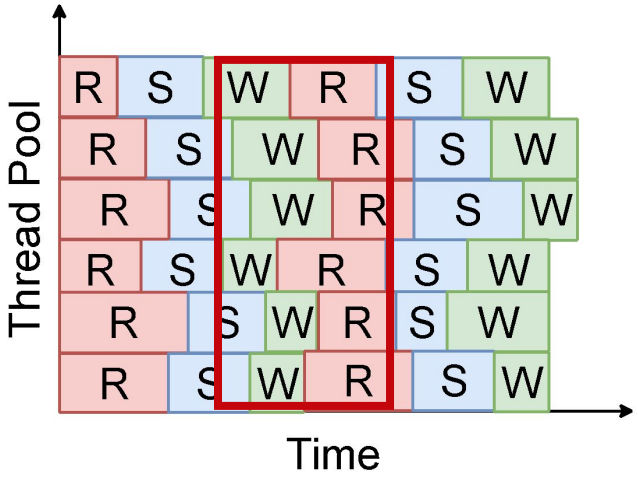
- WiscSort **OnePass**: $N(K+P) < M$
- WiscSort **MergePass**: $N(K+P) > M$

N is number of keys, K, P, M are key, pointer and memory sizes.



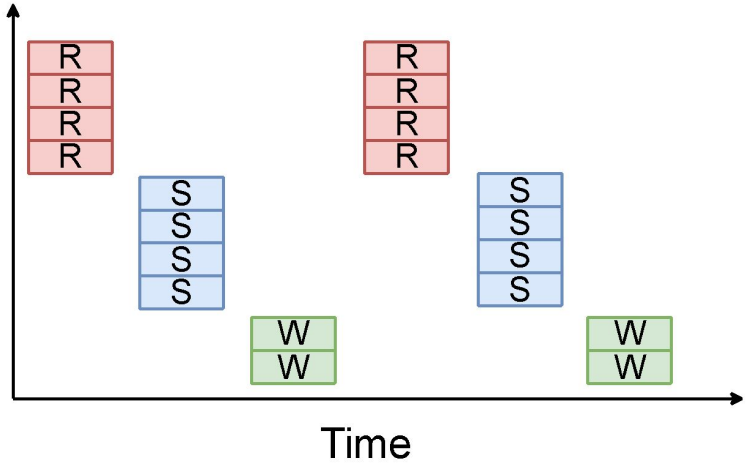
[2] Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook, ATC 2020

(BRAID)
WiscSort: Thread-pool controller and Interference aware scheduler



No synchronization

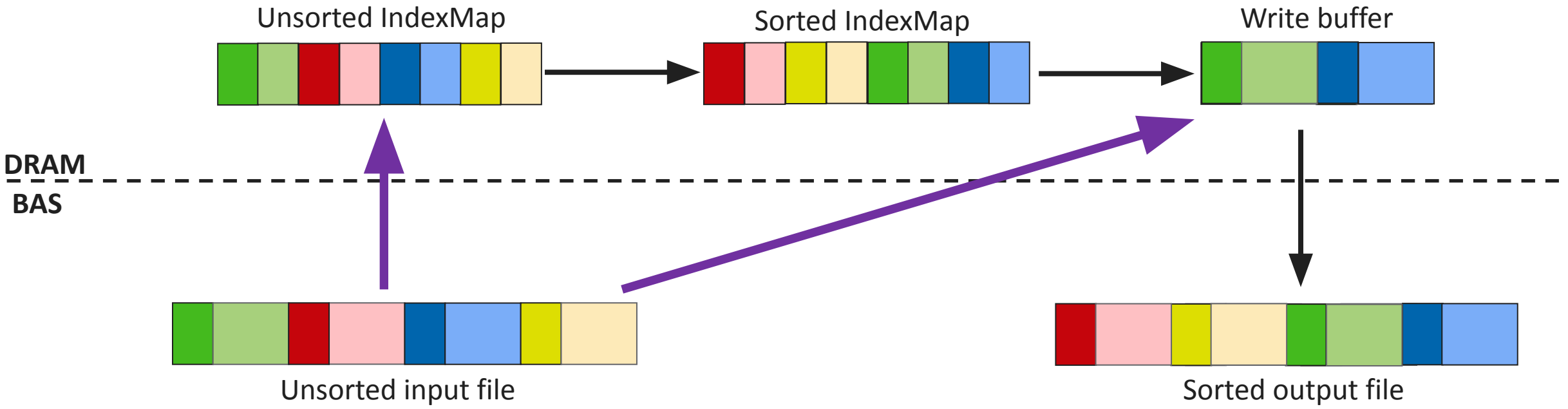
- Independent threads performing read, sort, write
- No way to control concurrency per operation
- Straggler threads can exist causing interference



Read-write thread pools w/ Non-overlapping requests

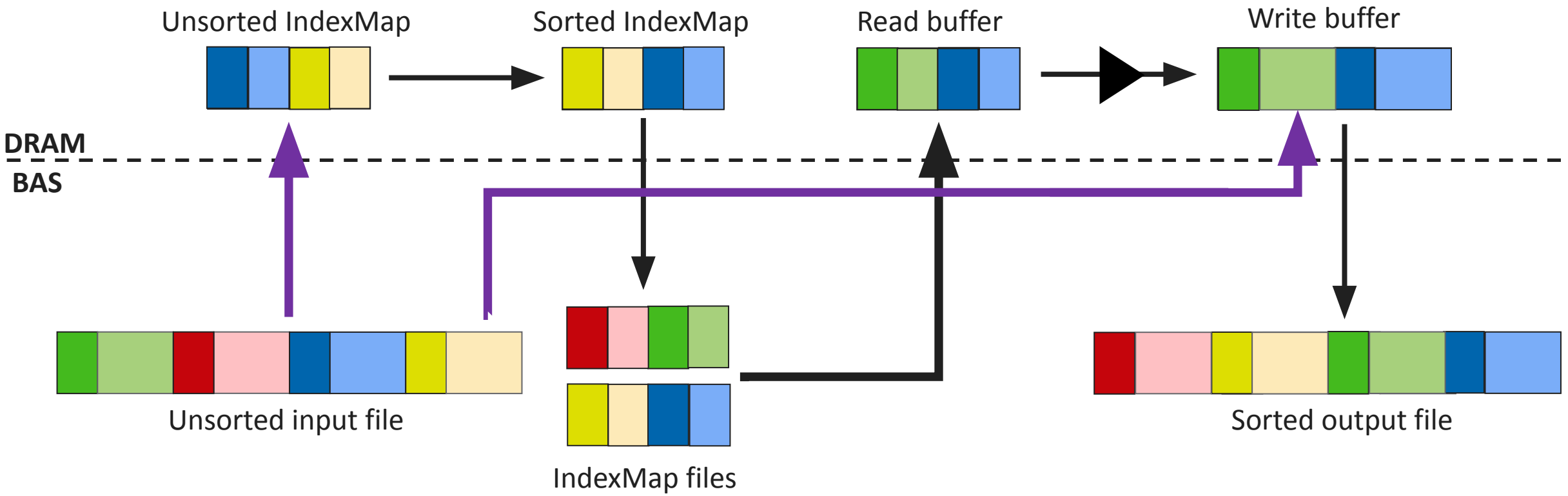
- Optimal concurrent read-write operations must be isolated to avoid degradation (**ID**)
- Buffer act as a logical barrier between different kinds of operations.

WiscSort OnePass – N(K+P) < M



KEY VALUE POINTER

WiscSort MergePass – N(K+P) > M



Evaluation

1. **What is WiscSort performance on Sortbenchmark?**
   Does WiscSort utilize the BRAID device effectively?

2. **What is the benefit of concurrency optimizations?**
   How do competing approaches compare?

3. **How does the benefit vary with different K:V ratios?** Does strided key gather perform better always?

4. **How does WiscSort perform on different BRAID devices?**
   With different characteristics (BD, BRD, $BARD$)?

CPU	Intel(R) Xeon Gold 5218 2nd Gen
Memory	2x 16GB @2400MHz DRAM
BAS	4x 128GB Intel Optane DC PMEM 100 @2666MHz
OS	Ubuntu 20 Linux 5.0

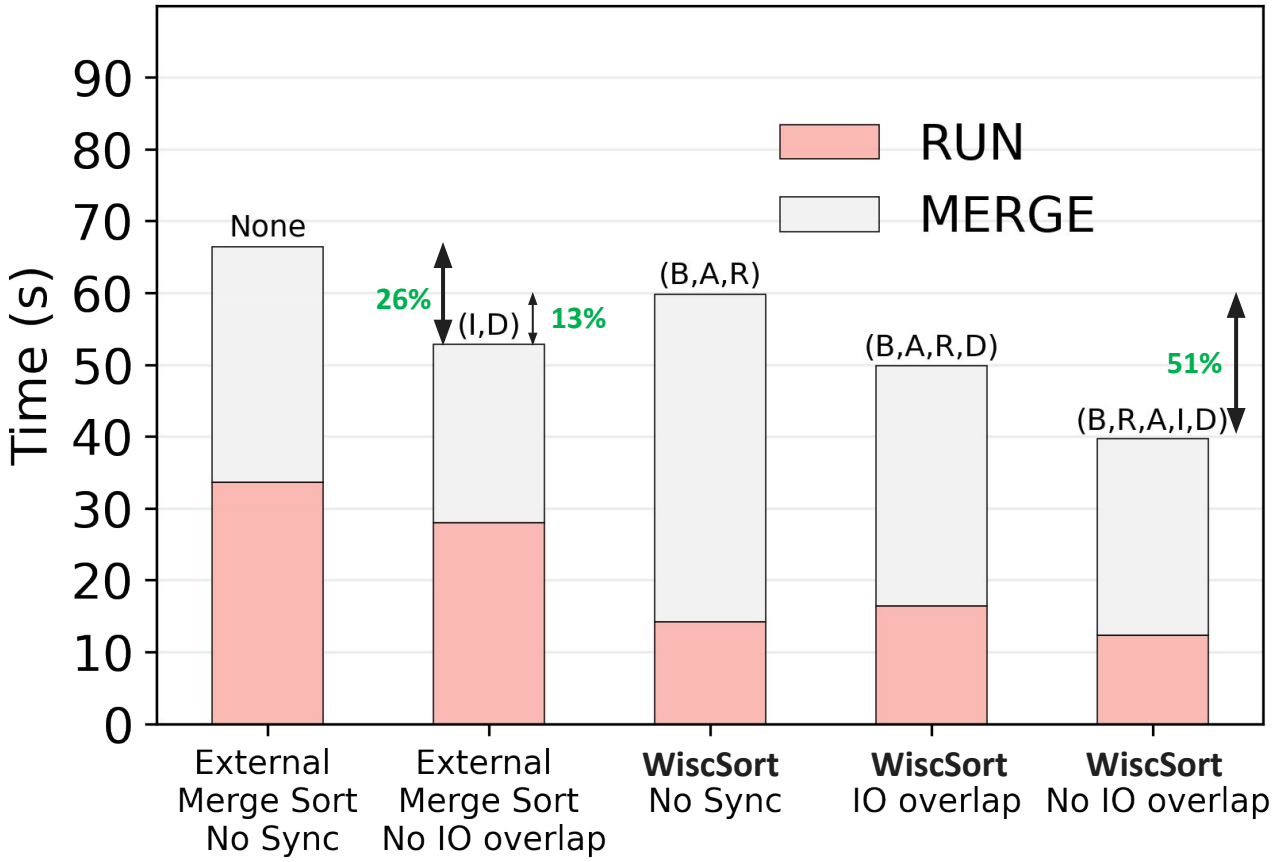
Concurrency and interference optimizations

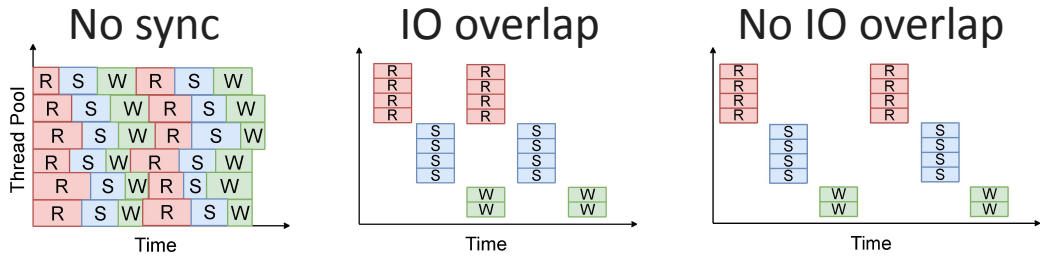
Even existing block-based solutions can be 26% faster by being aware of the concurrency constraints

I+D properties are more important than **B+A+R**

Being concurrency aware while exploiting byte-addressability and random read performance can give up to 50% improvement.

Sorting 400M records of 10B K: 90B V each.





The BRAID model

Property	Sorting Goal	Goal
Byte Addressability (**B**)	**Read only keys**	**Reduce I/O amplification**
Higher Random-Read performance (**R**)	**Random reads of keys and values**	**Avoid redundant reads**
Asymmetric Read-Write Cost (**A**)	**Limit or avoid writing RUN files**	**Trade more reads for fewer writes**
Read-Write Interference (**I**)	**Separate concurrent reader and writer threads**	**Avoid read-write interference**
Device-Constrained Concurrency (**D**)	**Manage read and write concurrency differently**	**Manage access concurrency**

WiscSort Summary

Conventional access strategies are suboptimal

Introduced the **BRAID** model

Comply with **BRAID** for peak performance

WiscSort: key-value separation (**BRA**), interference-aware scheduling (**I**), and thread-pool control (**D**).

WiscSort: **2-7x** faster than state-of-the-art.

Outline

Static Layout Optimization Redesign applications for modern storage

WiscSort: External Sorting For Byte-Addressable Storage

- How to sort on BAS?
- The BRAID model
- Key Value separation
- Thread-pool controller and interference-aware scheduler
- Evaluation

Case for Dynamic Layout Access semantics aren't always known

- Hotness Fragmentation
- Transient Object Hotness
- Rewards of Improved Page Utilization
- Address-Space-Engineering

Dynamic Layout Optimization Transparently improve application memory efficiency

OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering

- Challenges of Address-Space-Engineering
- Decoupling Layout from Reclamation
- Grouping Objects by Access Intensity
- Enabling Object Mobility
- Evaluation

Conclusions

Questions?

Access semantics aren't always known

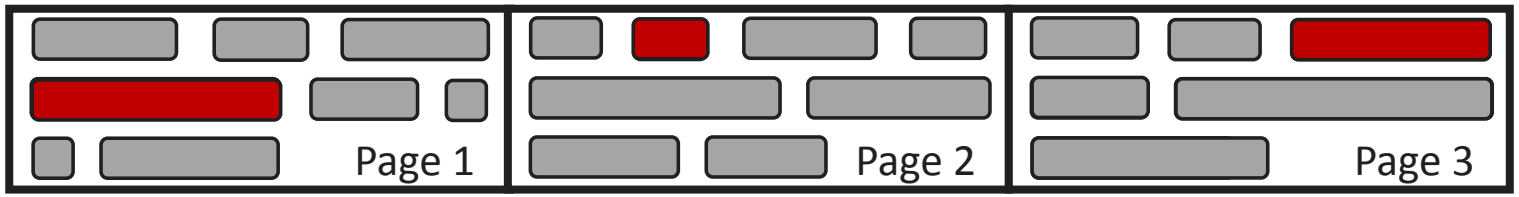
Case for Dynamic Layout - Overview

- Pervasive fragmentation that is neither internal nor external
- New metric (Page Utilization) to quantify **hotness fragmentation**
- **Active pages are mostly cold** in both open-source and production workloads
- Object hotness is neither knowable at allocation nor stable over time
- The principles of **Address-Space-Engineering**

Wasted Memory - Hotness Fragmentation

- Object (▢) creation order determines their location on the virtual address space
- Real world workloads exhibit highly skewed access patterns
- 90% of requests at Meta [1], Twitter [2], and Alibaba [3] access KV objects <1 KiB in size

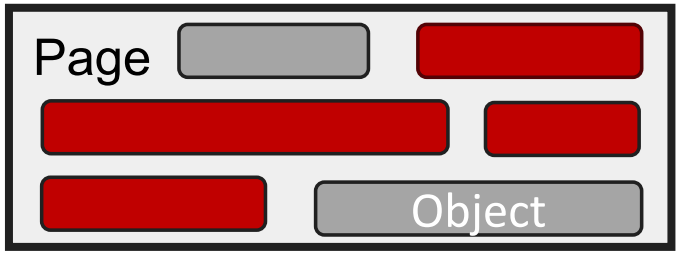
Frequently (hot) and infrequently (cold) accessed objects intermingled on the same page



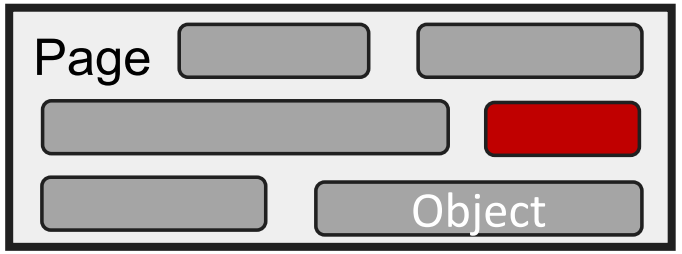
[1] Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook, ATC 2020
[2] A large scale analysis of hundreds of in-memory cache clusters at Twitter, OSDI 2020
[3] HotRing: A Hotspot-Aware In-Memory Key-Value Store, FAST 2020

Quantify Hotness Fragmentation

$$\text{Page Utilization} = \frac{\text{Total Unique Bytes Accessed}}{\text{Total Unique Pages Accessed x Page Size}}$$



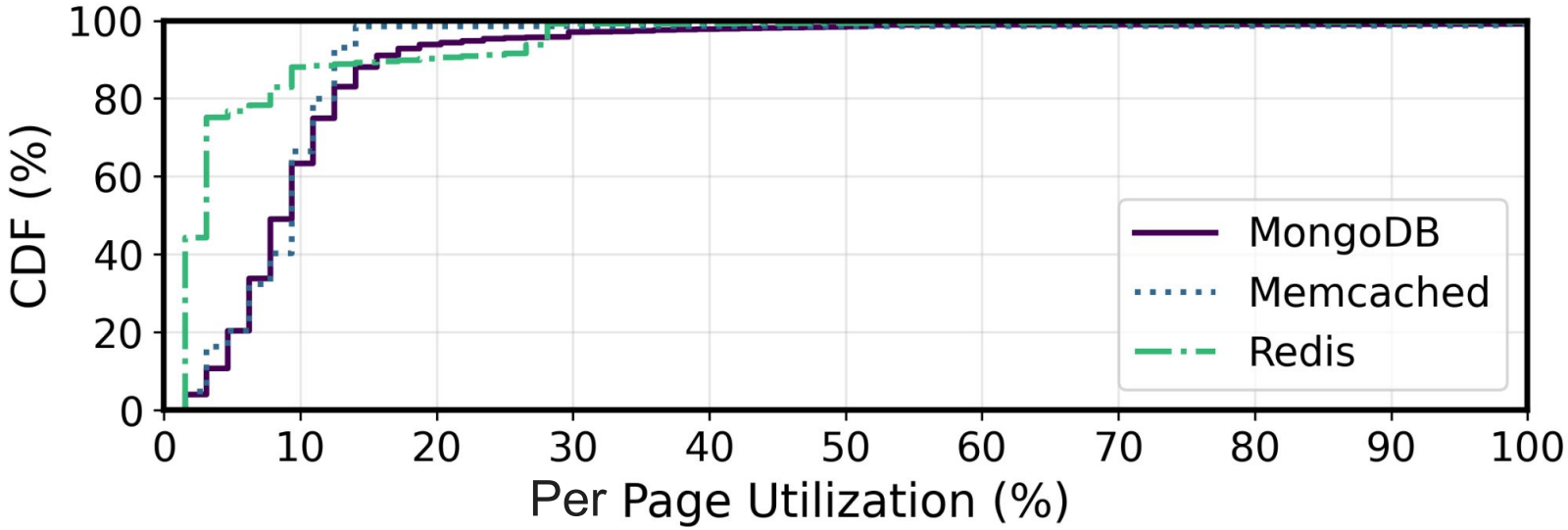
High Page Utilization



Low Page Utilization

Page Utilization directly measures the **hotness fragmentation** of an application for a workload

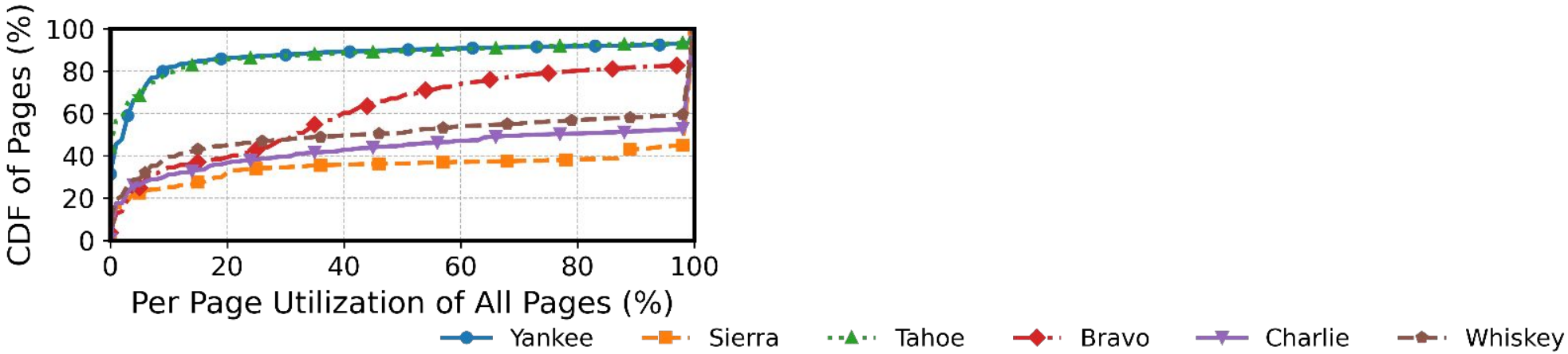
Active pages are mostly cold (open source)



Page Utilization for 360s epochs running YCSB-C with Zipfian distribution

- **75%** of accessed pages in Redis utilize **3%** or less of their capacity
- **95%** of pages in MongoDB and Memcached use less than **15%**

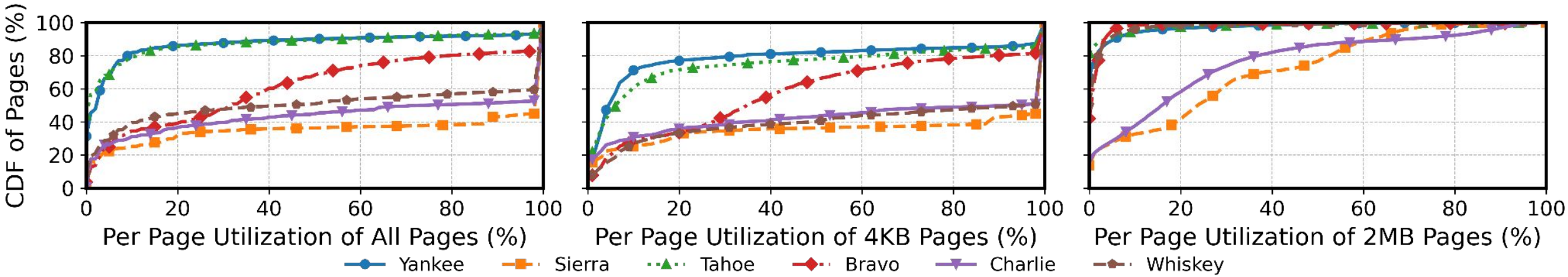
Active pages are mostly cold (production)



Page Utilization for upto 30s of Google production workloads

- All pages: **Half** of all pages waste over **70%** of their capacity on cold data

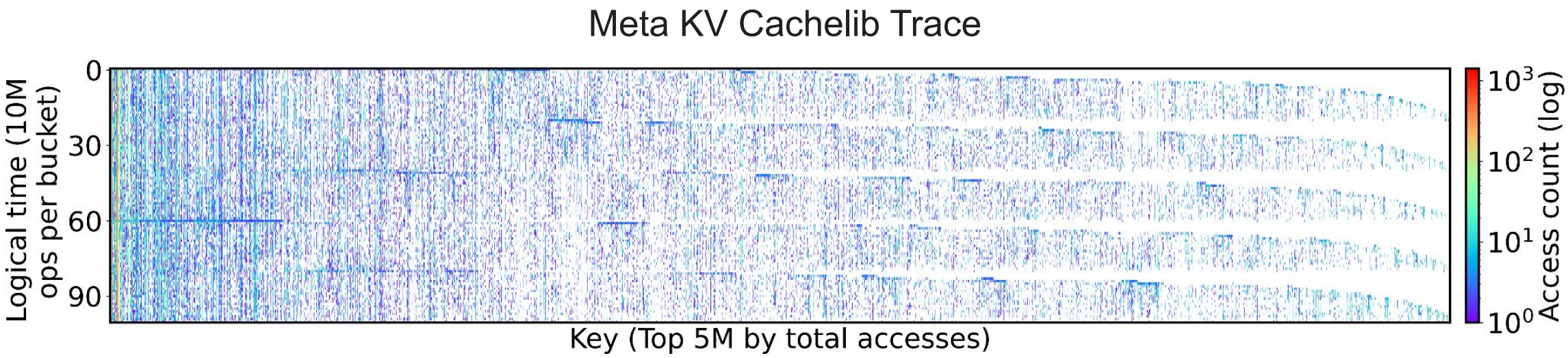
Active pages are mostly cold (production)



Page Utilization for upto 30s of Google production workloads

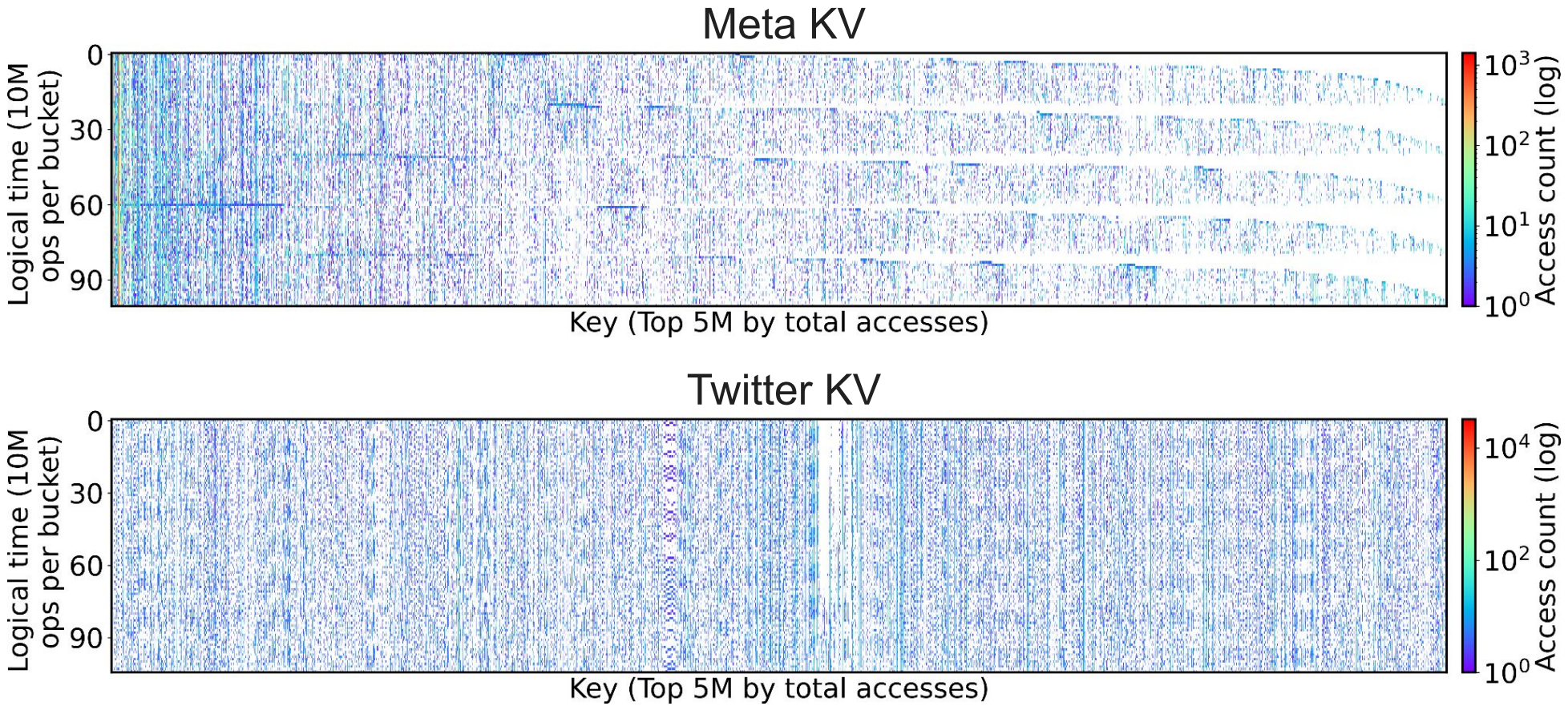
- All pages: **Half** of all pages waste over **70%** of their capacity on cold data
- 4KB pages: **60–80%** of pages utilize less than **20%** of their capacity
- 2MB pages: **90%** of huge pages utilize less than **10%** of their capacity

Object Hotness Is Transient



- Bursts of activity followed by long idle gaps
- Object hotness is neither knowable at allocation nor stable over time

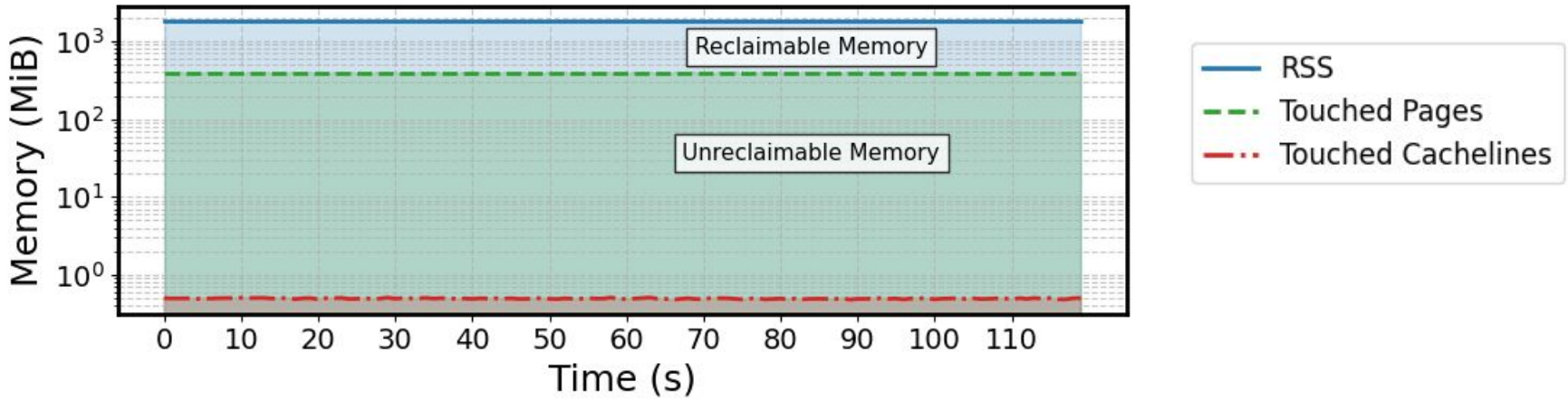
Object Hotness Is Transient



- Bursts of activity followed by long idle gaps
- Object hotness is neither knowable at allocation nor stable over time

Rewards of Improved Page Utilization

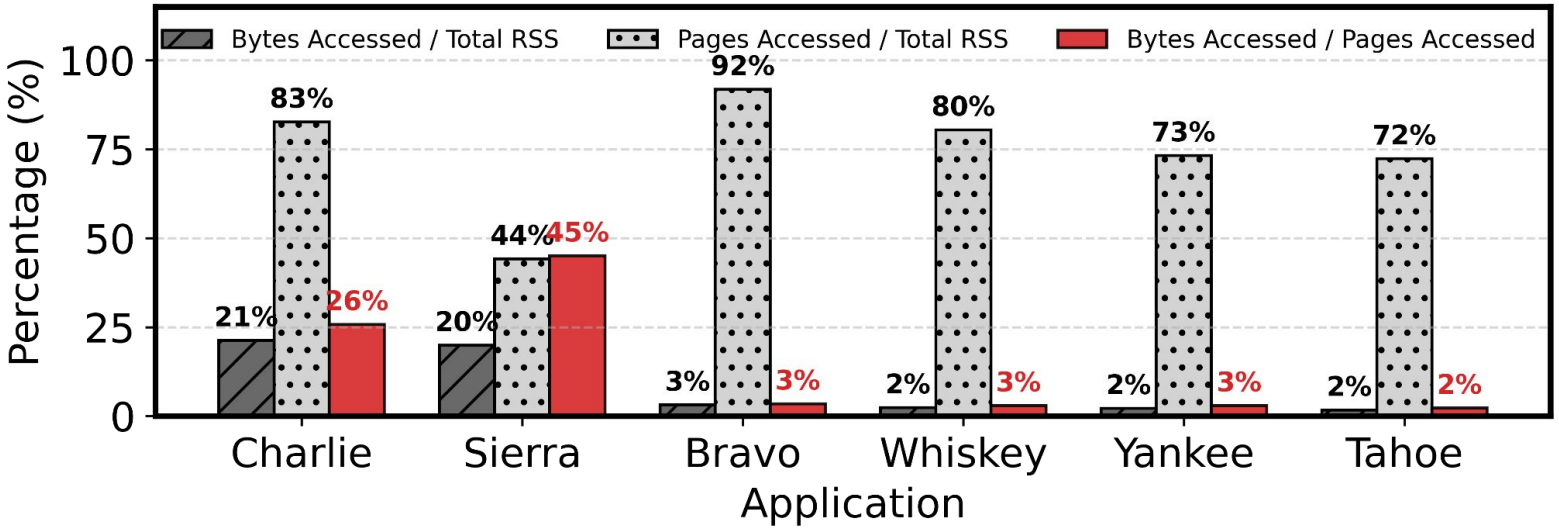
#1 Increased Reclaimable Memory: Fewer pages needed to serve skewed workloads



Redis touches ~0.5 MiB of cachelines but 1.2 GiB remains resident for YCSB

Rewards of Improved Page Utilization

#1 Increased Reclaimable Memory: Fewer pages needed to serve skewed workloads

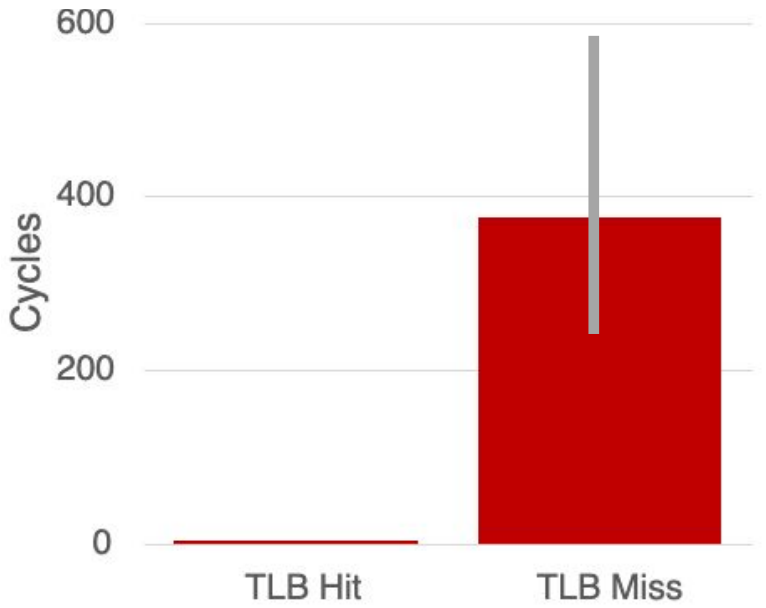


- 1-21% of bytes are accessed, yet 64%-96% of pages are touched, because a few hot objects pin entire pages in DRAM
- Over 97% of memory within touched pages cannot be reclaimed despite being cold

Rewards of Improved Page Utilization

#1 Increased Reclaimable Memory: Fewer pages needed to serve skewed workloads

#2 Targeted Huge Pages: Saves CPU cycles without sacrificing reclaimable memory



- 11% of CPU cycles at Google spent on dTLB load misses [1]
- Applying THP indiscriminately increases footprint by 69% [2]

[1] Characterizing a Memory Allocator at Warehouse Scale, ASPLOS 2024
[2] Coordinated and efficient huge page management with ingens, OSDI 2016

Rewards of Improved Page Utilization

#1 Increased Reclaimable Memory: Fewer pages needed to serve skewed workloads

#2 Targeted Huge Pages: Saves CPU cycles without sacrificing reclaimable memory

#3 Require fewer machines: More cost effective and sustainable data centers

- Jobs with small working sets but large allocation footprints are spread across multiple machines [3]
- DRAM produces **12x** more emissions per bit than SSDs [4]

[1] Characterizing a Memory Allocator at Warehouse Scale, ASPLOS 2024
[2] Coordinated and efficient huge page management with ingens, OSDI 2016
[3] Borg: the Next Generation, EuroSys 2020
[4] FairyWREN: A Sustainable Cache for Emerging Write-Read-Erase Flash Interfaces, OSDI 2024

Rewards of Improved Page Utilization

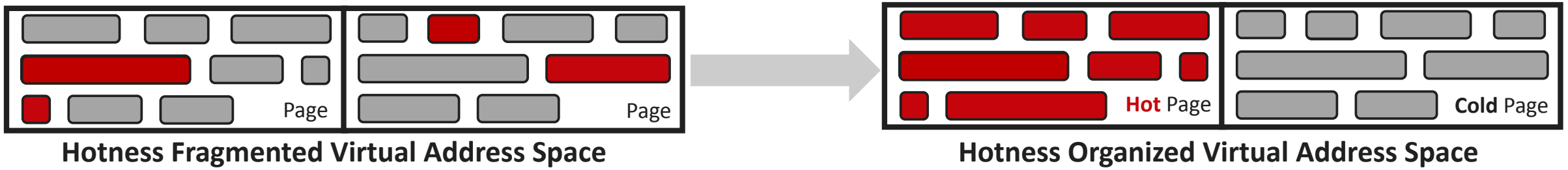
#1 Increased Reclaimable Memory: Fewer pages needed to serve skewed workloads

#2 Targeted Huge Pages: Saves CPU cycles without sacrificing reclaimable memory

#3 Require fewer machines: More cost effective and sustainable data centers

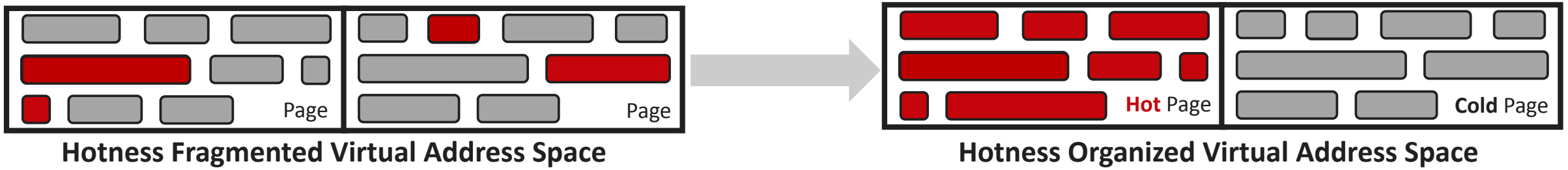
Address-Space-Engineering

Engineer the application's virtual address space to be OS-tiering-friendly, adapting to workload access patterns



Address-Space-Engineering

Engineer the application's virtual address space to be OS-tiering-friendly, adapting to workload access patterns



Principles of workload-optimized address space:

P1. Decoupling Layout from Tiering

P2. Dynamically Grouping Objects by Access Intensity

P3. Enabling Object Mobility

Outline

Static Layout Optimization Redesign applications for modern storage

WiscSort: External Sorting For Byte-Addressable Storage

- How to sort on BAS?
- The BRAID model
- Key Value separation
- Thread-pool controller and interference-aware scheduler
- Evaluation

Case for Dynamic Layout Access semantics aren't always known

- Hotness Fragmentation
- Transient Object Hotness
- Rewards of Improved Page Utilization
- Address-Space-Engineering

Dynamic Layout Optimization Transparently improve application memory efficiency

OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering

- Challenges of Address-Space-Engineering
- Decoupling Layout from Tiering
- Grouping Objects by Access Intensity
- Enabling Object Mobility
- Evaluation

Conclusions

Questions?

Transparently improve application memory efficiency

OBASE - Overview

- Challenges of realizing **Address-Space-Engineering**
- A compiler-runtime system that dynamically reorganize the virtual address space for a workload (OBASE)
- Key components of OBASE
- OBASE increases memory savings of swapping solutions without sacrificing performance
- OBASE increases performance of multiple page based tiering solutions

Object Based Address-Space-Engineering

A compiler-runtime frontend that dynamically reorganizes the address space for a workload in unmanaged language

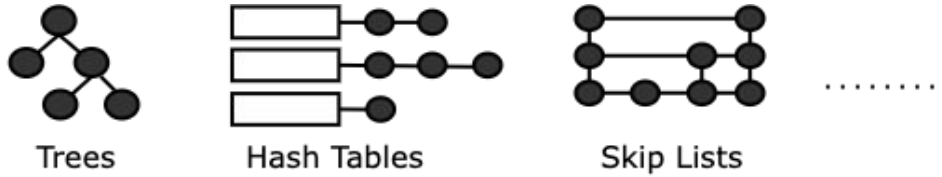
P1. Decoupling Layout from Tiering
Backend Independence

P2. Grouping Objects by Access Intensity
Tracking and Grouping Objects
Dynamic Adaptation

P3. Enabling Object Mobility
Safe Concurrent Migration

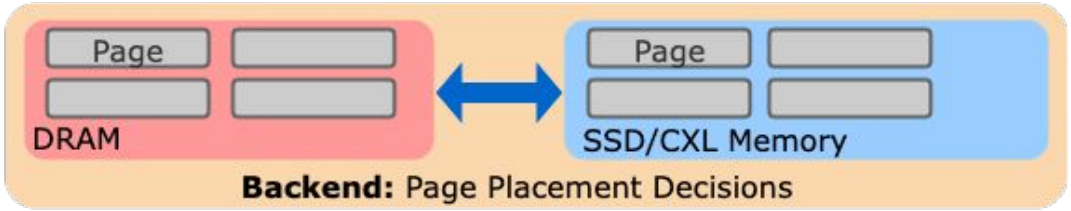
Address-Space-Engineering

P1. Decoupling Layout from Tiering



Requirements:

- Complement tiering / swapping solutions like: Kswapd, TMO, Zswap, TPP, ARMS, etc
- Require no new OS abstractions
- Require no specialized hardware knowledge

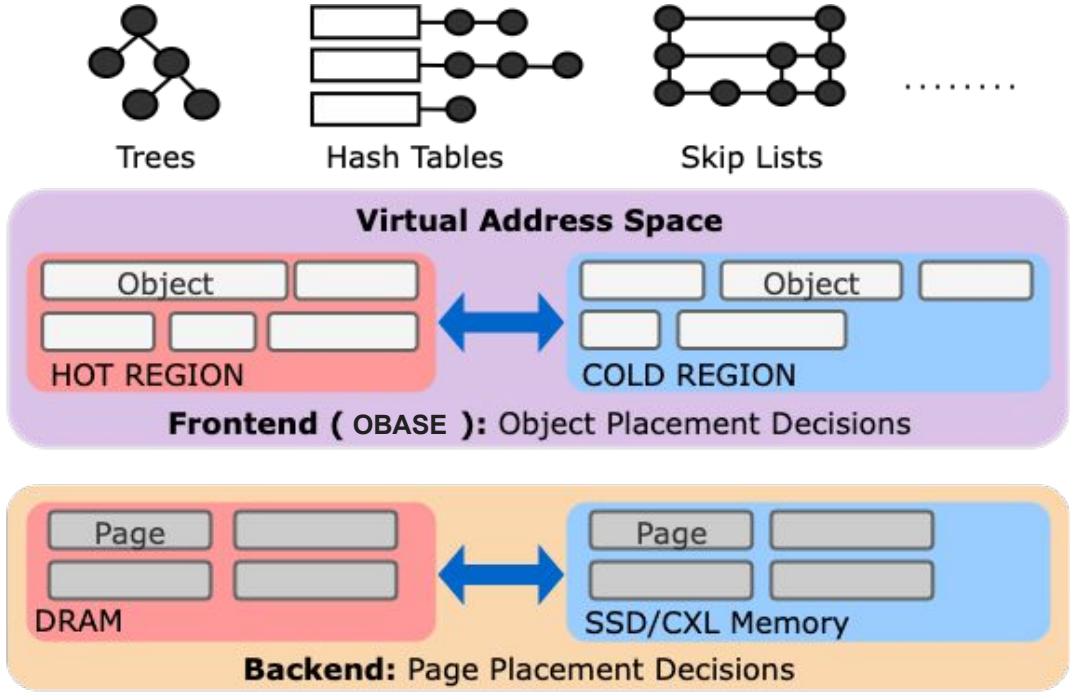


Object Based Address-Space-Engineering

P1. Decoupling Layout from Tiering

Backend Independence

- Per process user space runtime
- Static analysis for compile time annotations
- **Frontend** organizes address space layout and **backend** acts upon it



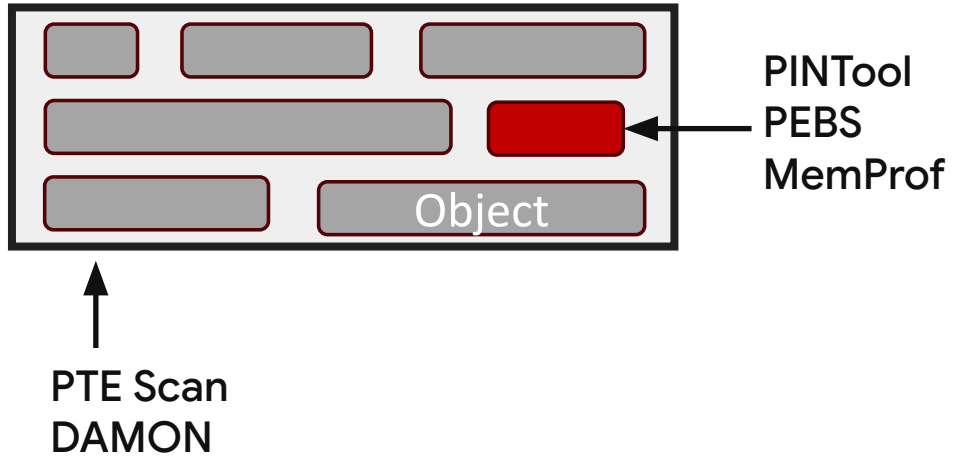
TMO, AutoNuma, TPP, MEMTIS, etc

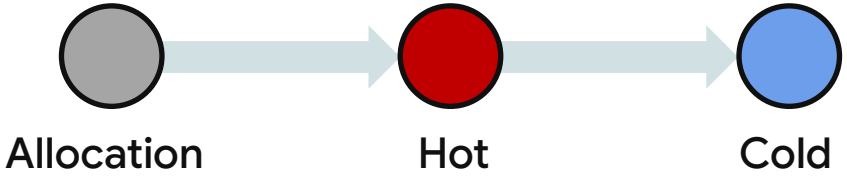
Address-Space-Engineering

P2. Grouping Objects by Access Intensity

Requirements:

- Tracking activity at allocation granularity
- Activity tracking must have low overhead
- Static hints at allocation-time is insufficient

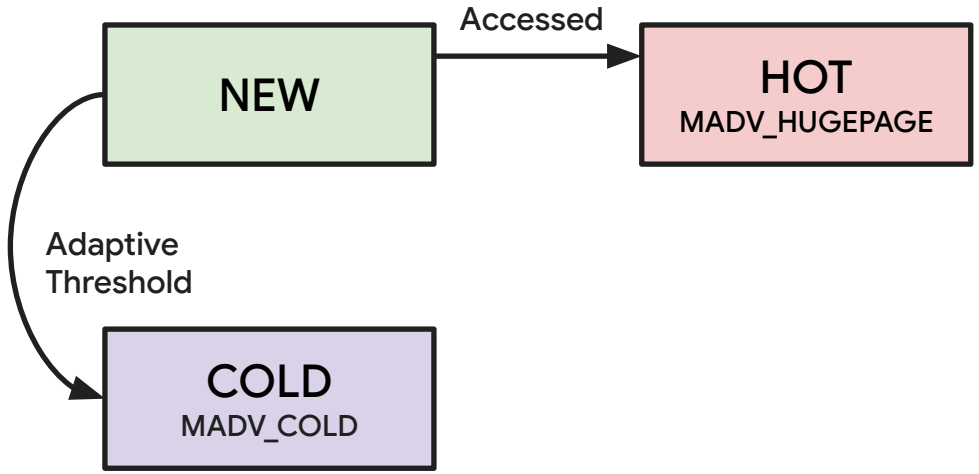




Object Based Address-Space-Engineering

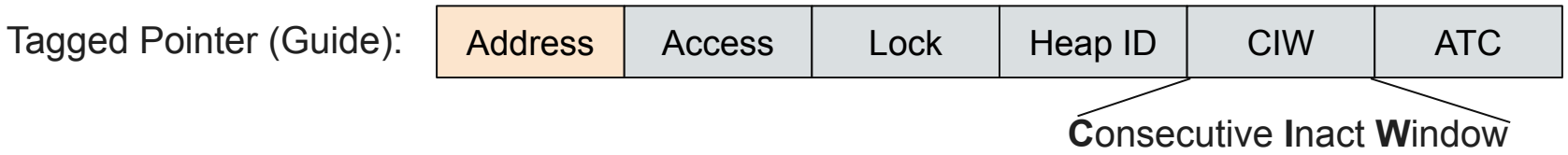
P2. Grouping Objects by Access Intensity

OC in action every 120s



Tracking and Grouping Objects

- Tagged pointers to track activity on dereference
- **Object Collector** (**OC**) periodically scans managed objects to maintain freshness and group to heaps
- Custom Jemalloc enables **OC** hints and ensures different heap regions are contiguous

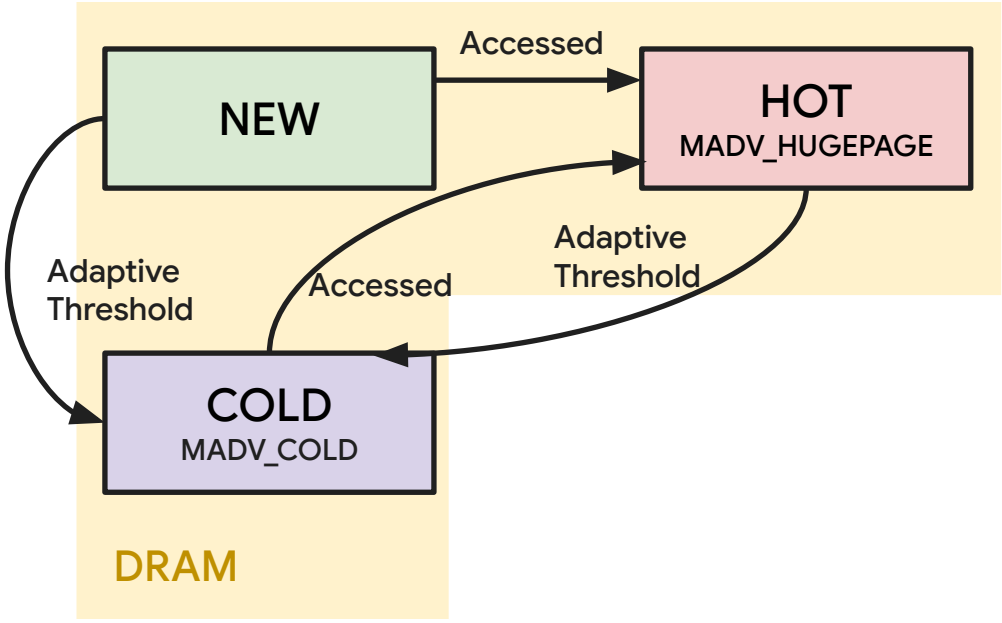


Object Based Address-Space-Engineering

P2. Grouping Objects by Access Intensity

Adaptive Workload Response

- Promotion Rate (PR) as performance proxy
- Adapting cold threshold to reach target PR
- Proactive reclamation using `MADV_PAGEOUT`



PR = (unique COLD pages accessed / working set size) × (60 / scan interval)

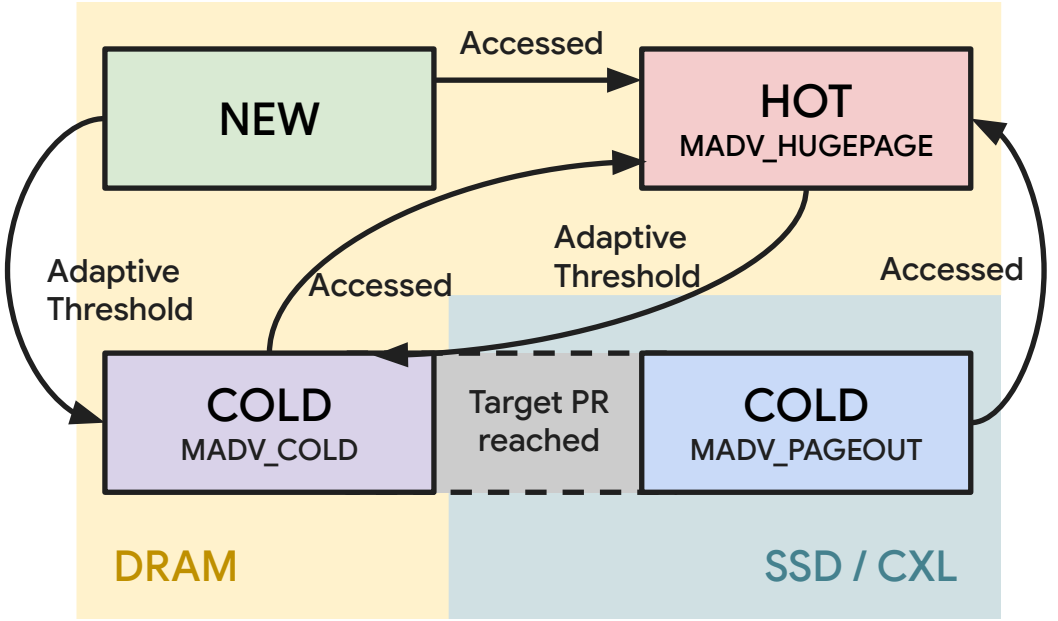
Target Promotion Rate = 1% (based on swap and CXL tiering deployments)

Object Based Address-Space-Engineering

P2. Grouping Objects by Access Intensity

Adaptive Workload Response

- Promotion Rate (PR) as performance proxy
- Linear increase/decrease of cold threshold to reach target PR
- Proactive reclamation using `MADV_PAGEOUT`



PR = (unique COLD pages accessed / working set size) × (60 / scan interval)

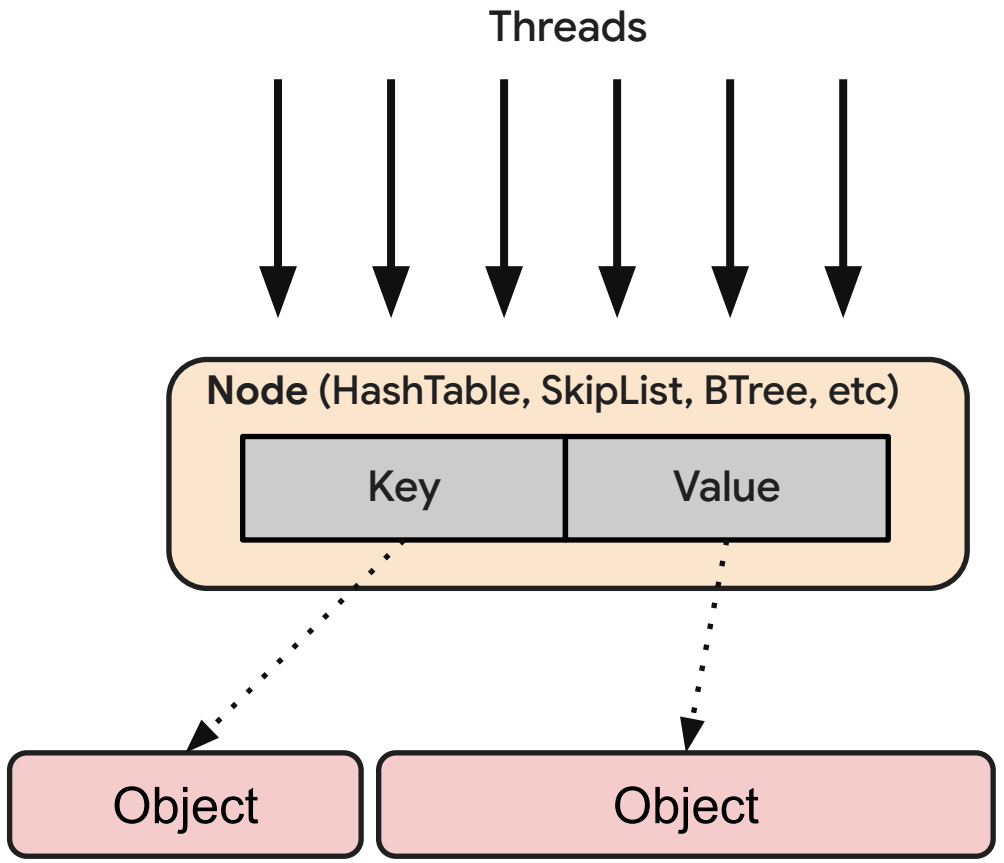
Target Promotion Rate = 1% (based on swap and CXL tiering deployments)

Address-Space-Engineering

P3. Enabling Object Mobility

Background and Requirement:

- Unmanaged languages assume object addresses are fixed after allocation
- Focus on **pointer-based data structures**
- Fast but safe in concurrent environments

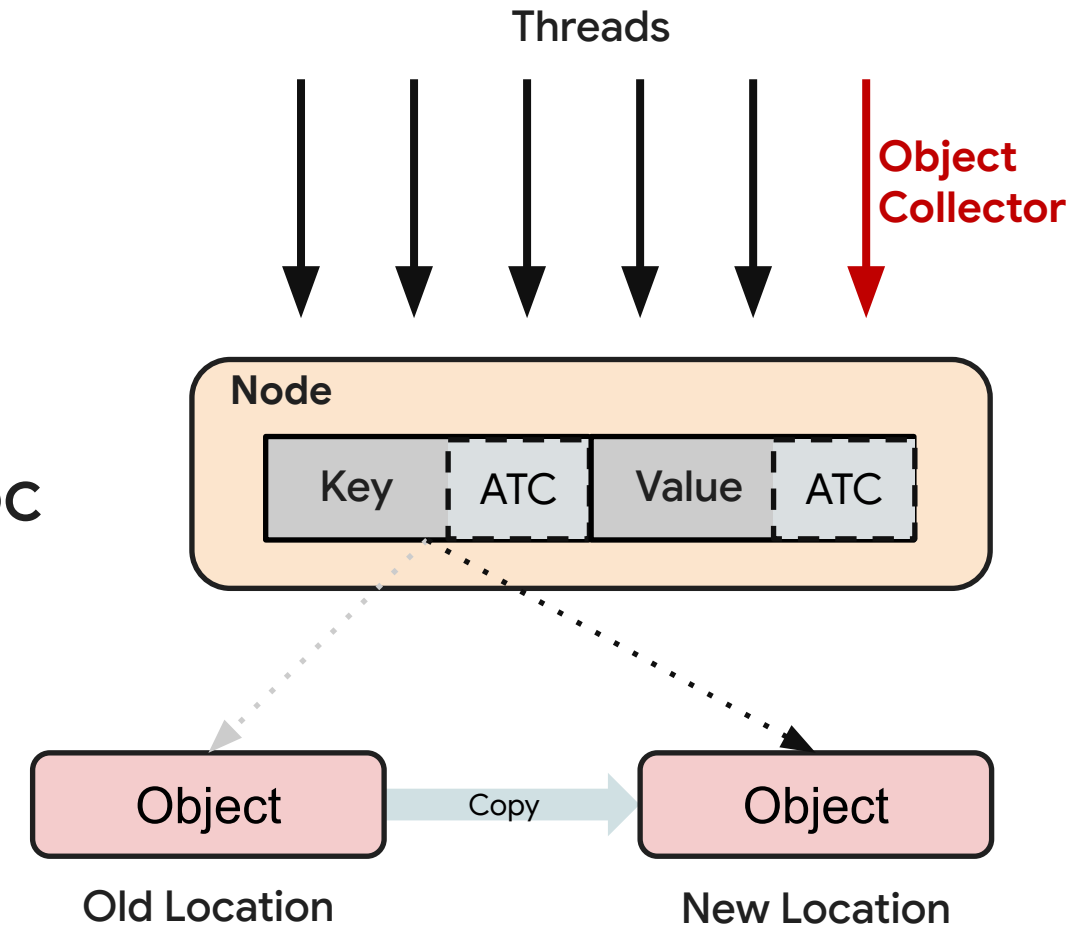


Object Based Address-Space-Engineering

P3. Enabling Object Mobility

Safe Concurrent Migration

- Track objects activity in real-time using Active Thread Count (ATC) embedded in unused bits
- Compiler manages ATC through static analysis. **OC** enables ATC during `ACTIVE` phase.
- Optimistic Object Migration



Evaluation

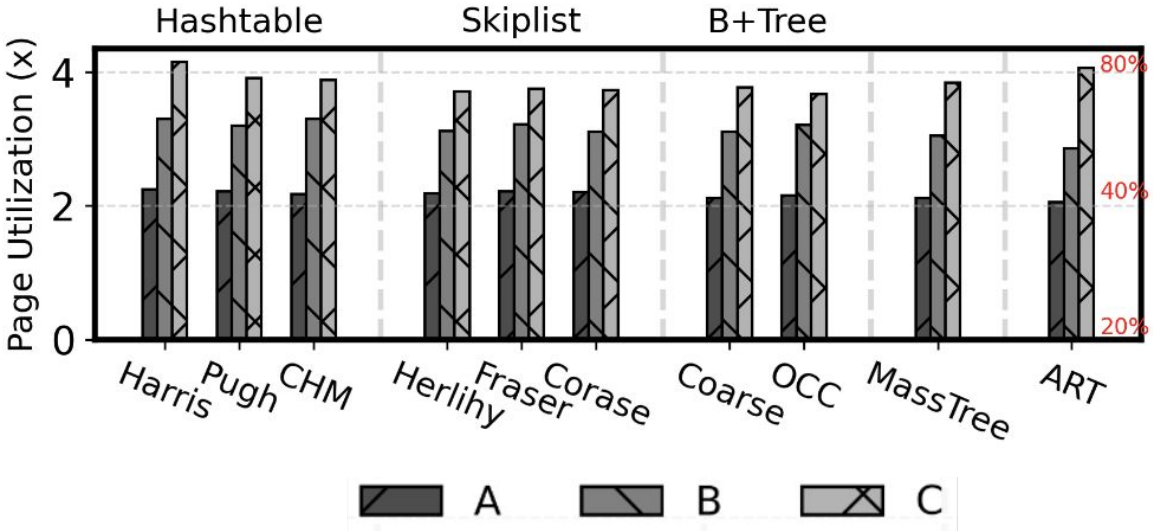
- **Ten** popular pointer based data structures: ART, MassTree, BTree, HashTable, etc
- **Six** unique concurrency mechanisms ranging from global locks to lock-free
- Improve **Six** reclamation/tiering backends (TMO, Memtis, etc)
- Evaluated against YCSB and Production traces from Meta and Twitter
- Overhead, scalability, and adaptive policy studies
- Multi-tenant consolidation

CPU	Intel(R) Xeon Gold 5218 (16 cores)
Memory	2x 16GB @2400MHz DRAM
BAS	4x 128GB Intel Optane DC PMEM 100 @2666MHz
SSD	512 GB P4800x SSD
OS	Ubuntu 22.04

Frontend Effectiveness

10M KV pairs with 30B Keys and 1024B values

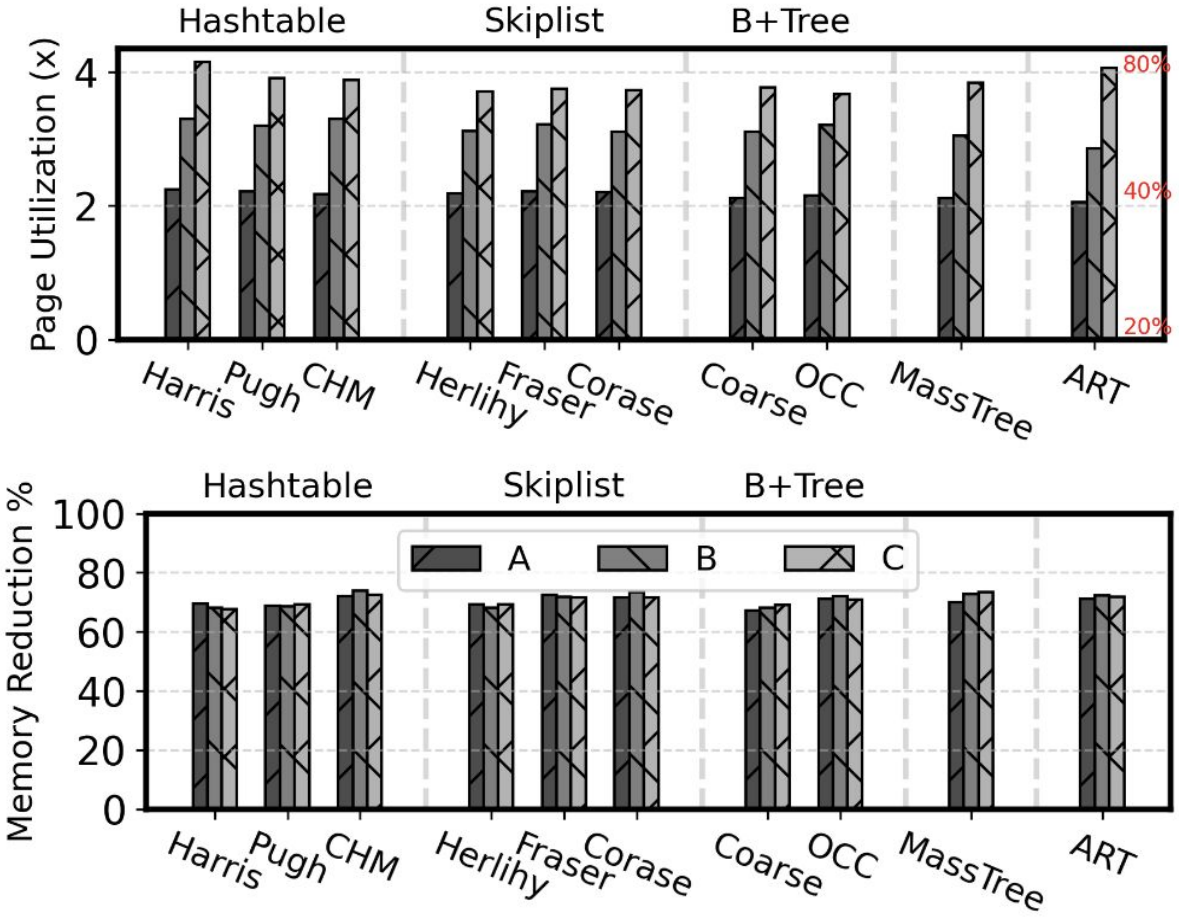
- OBASE **increases page utilization** by
  - **2x** for workload A (50% read 50% write)
  - **3x** for workload B (95% read 5% write)
  - **4x** for workload C (100% read)



Frontend Effectiveness

10M KV pairs with 30B Keys and 1024B values

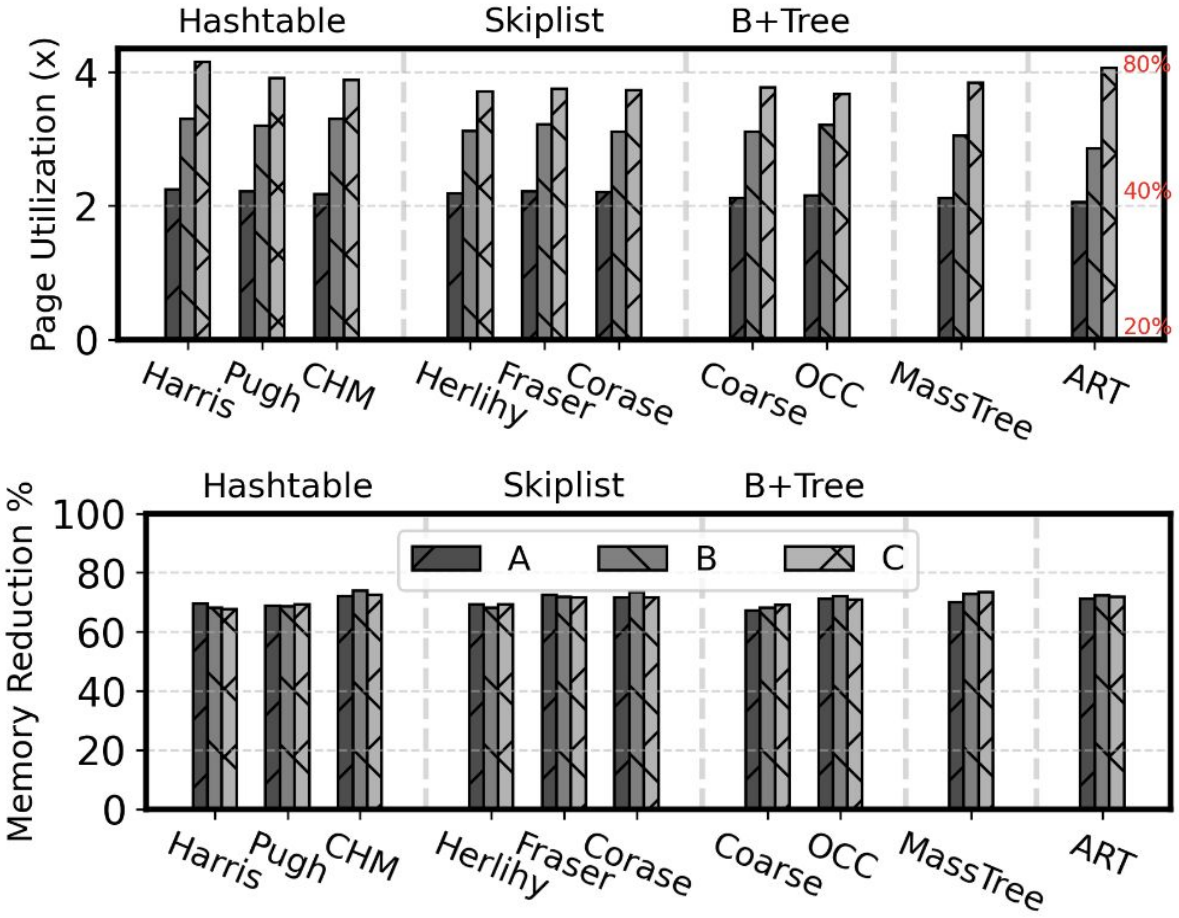
- OBASE **increases page utilization** by
  - **2x** for workload A (50% read 50% write)
  - **3x** for workload B (95% read 5% write)
  - **4x** for workload C (100% read)
- OBASE Hinted reduces **memory usage** by up to **70%** through object-level cold detection and heap organization



Frontend Effectiveness

10M KV pairs with 30B Keys and 1024B values

- OBASE **increases page utilization** by
  - **2x** for workload A (50% read 50% write)
  - **3x** for workload B (95% read 5% write)
  - **4x** for workload C (100% read)
- OBASE Hinted reduces **memory usage** by up to **70%** through object-level cold detection and heap organization
- OBASE tracking overhead lowers average throughput by **2.5%** and increases P90 latency by **5%**



Backend Validation (Reclamation)

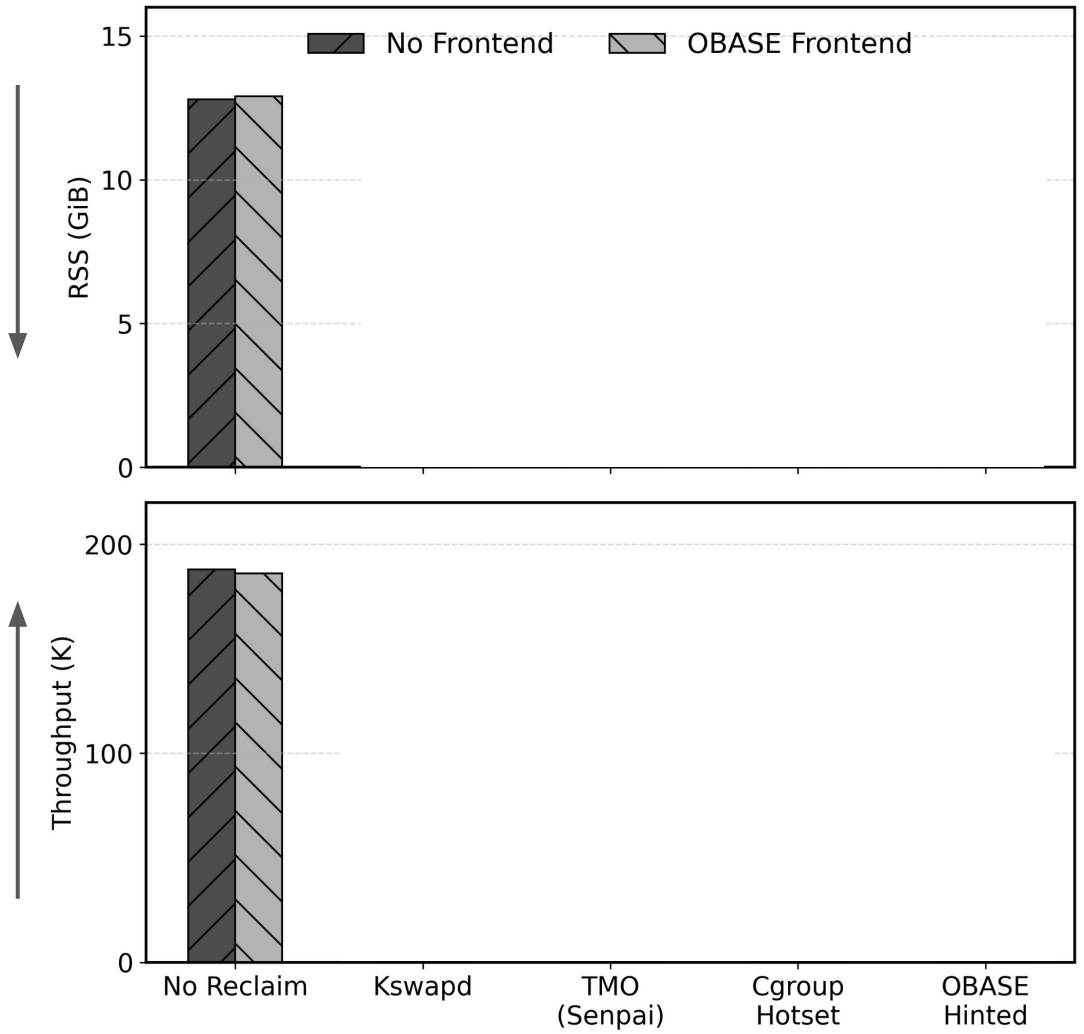
YCSB-C with ~12 GiB footprint and ~4 GiB actively accessed

Reclamation backends:

- **Kswapd**: Reclaims under memory pressure
- **TMO**: Proactively reclaim memory
- **Cgroup hotset**: Cgroup limit set to 4GiB

Standard backends

- Sacrifice performance to save memory (cgroup hotset)
- Sacrifice savings to preserve performance (kswapd and TMO)



Backend Validation (Reclamation)

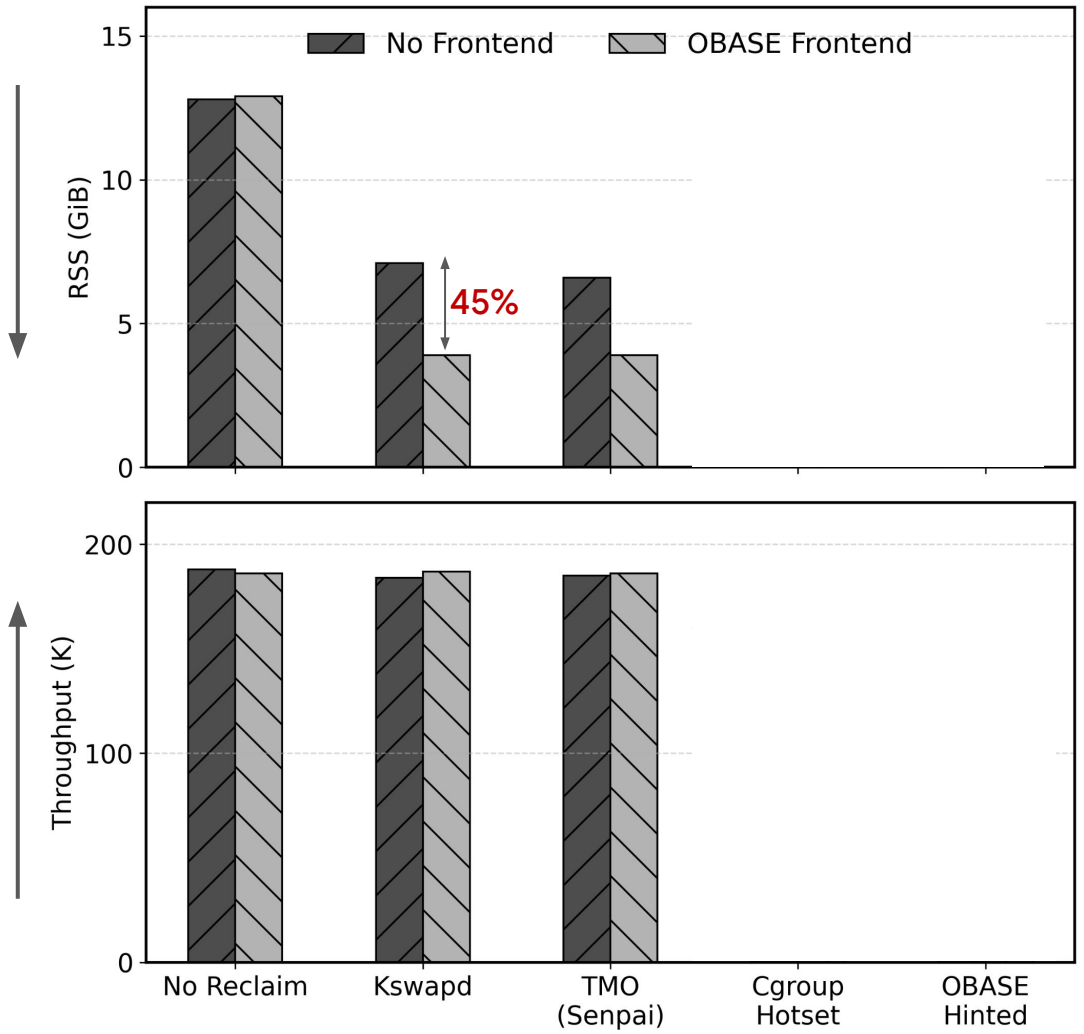
YCSB-C with ~12 GiB footprint and ~4 GiB actively accessed

Reclamation backends:

- **Kswapd**: Reclaims under memory pressure
- **TMO**: Proactively reclaim memory
- **Cgroup hotset**: Cgroup limit set to 4GiB

Standard backends

- Sacrifice performance to save memory (cgroup hotset)
- Sacrifice savings to preserve performance (kswapd and TMO)



Backend Validation (Reclamation)

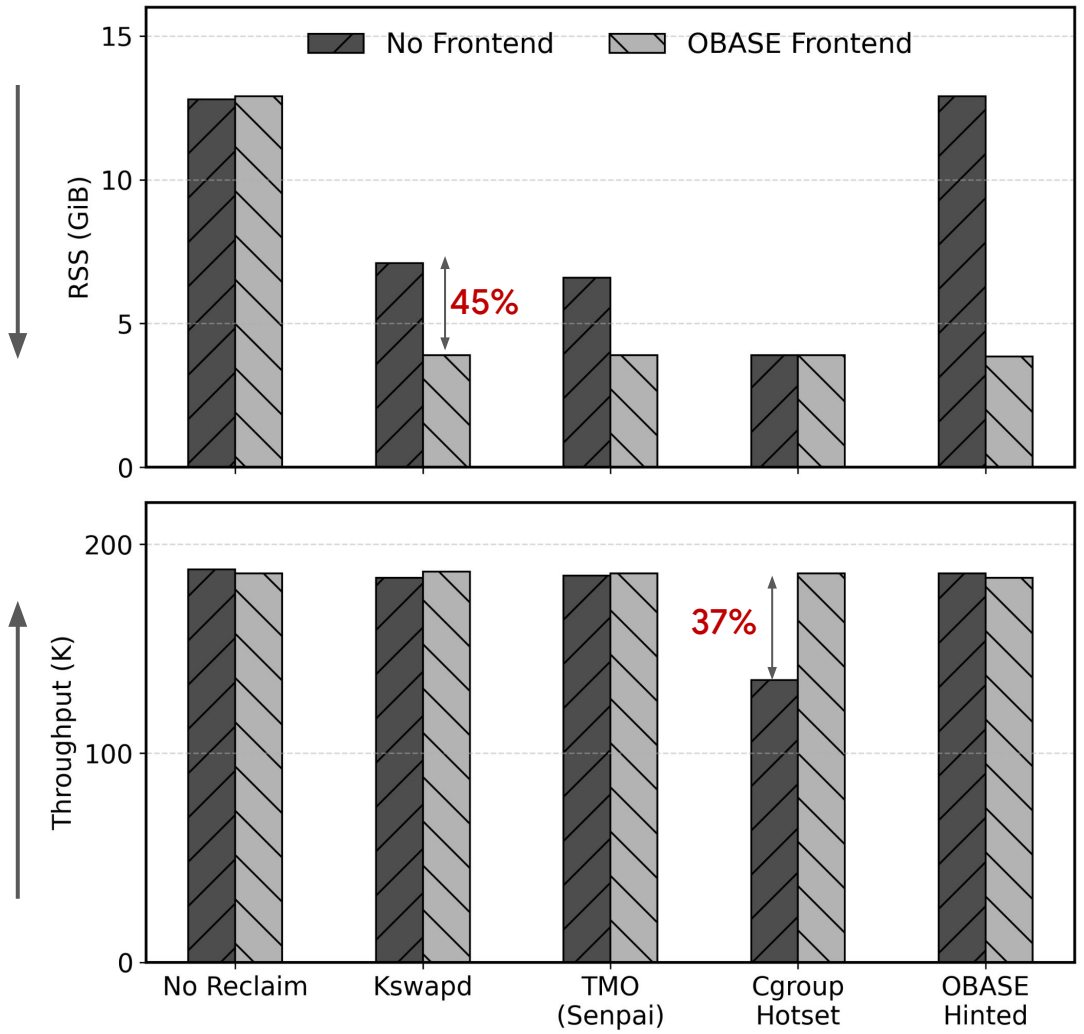
YCSB-C with ~12 GiB footprint and ~4 GiB actively accessed

Reclamation backends:

- **Kswapd**: Reclaims under memory pressure
- **TMO**: Proactively reclaim memory
- **Cgroup hotset**: Cgroup limit set to 4GiB

Standard backends

- Sacrifice performance to save memory (cgroup hotset)
- Sacrifice savings to preserve performance (kswapd and TMO)



Backend Validation (Reclamation)

YCSB-C with ~12 GiB footprint and ~4 GiB actively accessed

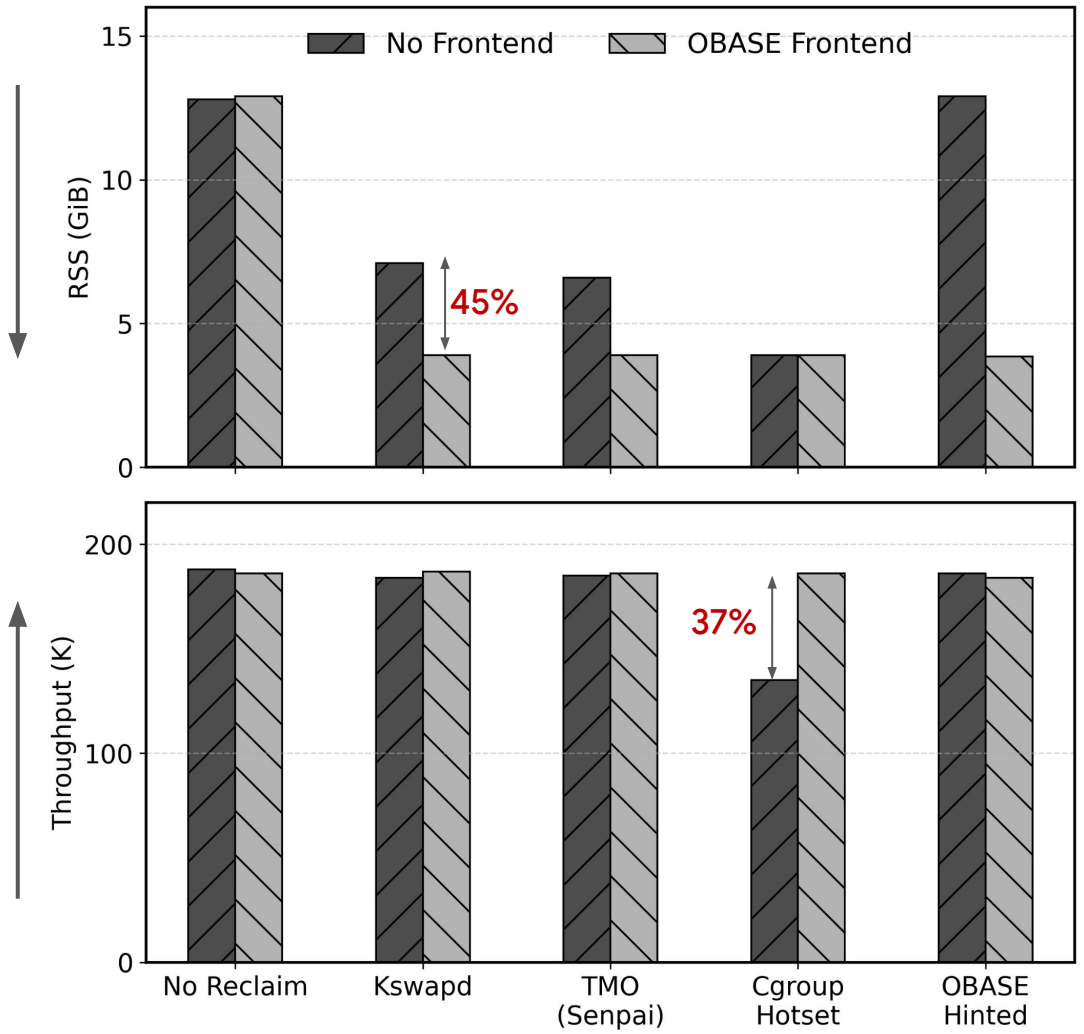
Reclamation backends:

- **Kswapd**: Reclaims under memory pressure
- **TMO**: Proactively reclaim memory
- **Cgroup hotset**: Cgroup limit set to 4GiB

Standard backends

- Sacrifice performance to save memory (cgroup hotset)
- Sacrifice savings to preserve performance (kswapd and TMO)

OBASE achieves max memory savings with no performance degradation

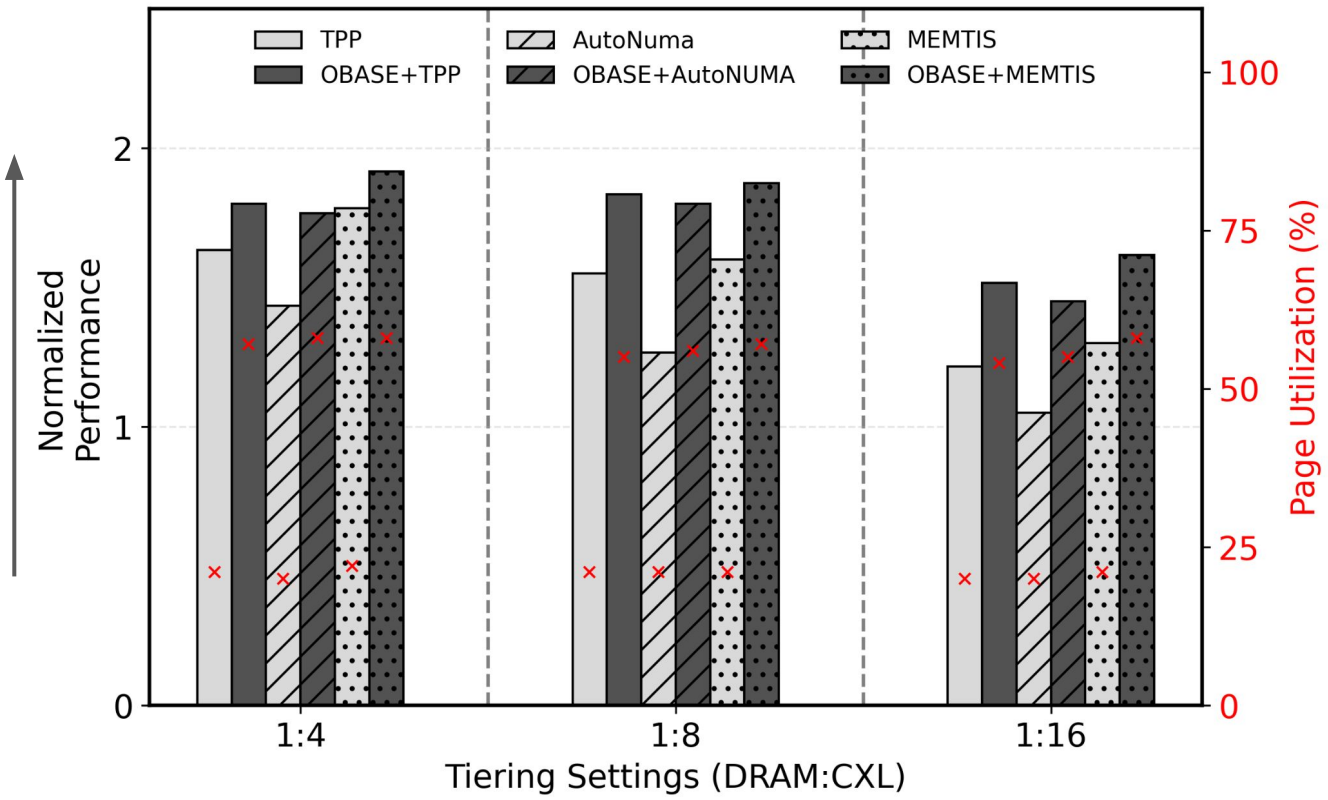


Backend Validation (Tiering)

YCSB-B with ~67 GiB footprint (50M - 30B key and 1KiB value)

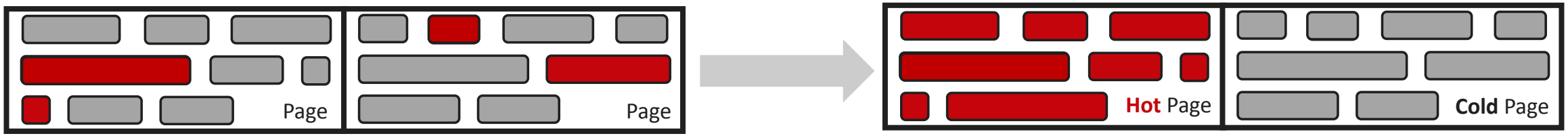
- Tiering backends: TPP, AutoNuma, MEMTIS
- DRAM:CXL ratio across three configurations: 1:4 (14.8 GiB DRAM), 1:8 (7.4 GiB DRAM), and 1:16 (3.9 GiB DRAM)
- Without OBASE hotset spans 16.3 GiB pages but with OBASE it spans 6.33 GiB

OBASE at ratio 1:X performs comparably to baseline at 1:(X/2)



OBASE Summary

- **Semantic gap** between application and OS memory model
- **Hotness fragmentation** is widespread
- Address-Space Engineering for improving page utilization
- **OBASE** as an approach for Address-Space Engineering
- Achieved higher memory savings across diverse data structures and multiple tiering backends without performance degradation



Hotness Fragmented Virtual Address Space **Hotness Organized Virtual Address Space**

Outline

Static Layout Optimization Redesign applications for modern storage

WiscSort: External Sorting For Byte-Addressable Storage

- How to sort on BAS?
- The BRAID model
- Key Value separation
- Thread-pool controller and interference-aware scheduler
- Evaluation

Case for Dynamic Layout Access semantics aren't always known

- Hotness Fragmentation
- Transient Object Hotness
- Rewards of Improved Page Utilization
- Address-Space-Engineering

Dynamic Layout Optimization Transparently improve application memory efficiency

OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering

- Challenges of Address-Space-Engineering
- Decoupling Layout from Reclamation
- Grouping Objects by Access Intensity
- Enabling Object Mobility
- Evaluation

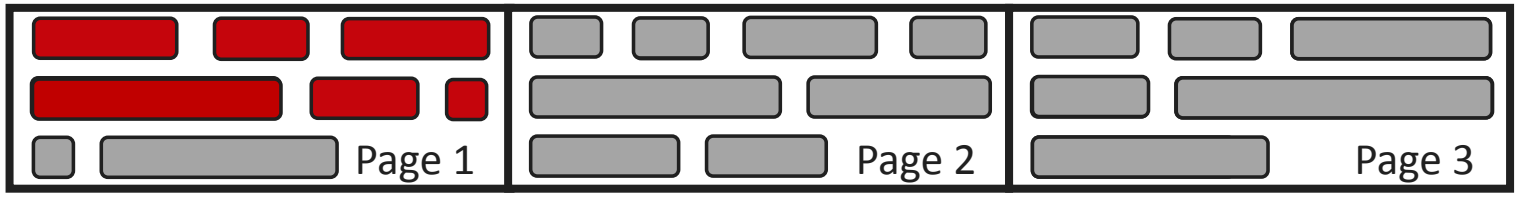
Conclusion

Questions?

Lessons learned

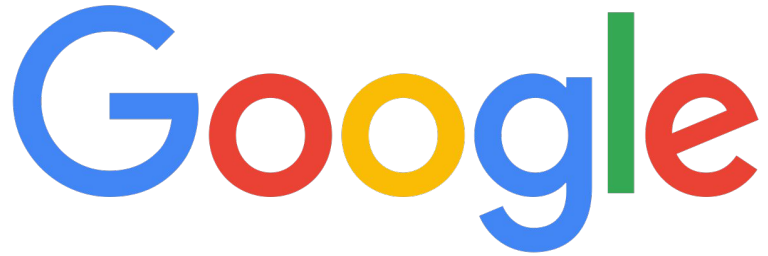
- Model the device before redesigning the algorithm (Model, Then Build?)
- Moving less data outweighs moving data faster
- Measure the problem at the right granularity
- Decompose along information boundaries
- Different approaches serve different regimes

Layout Optimization is key to overcome leaky page abstraction



What's Next?

- Moving to California!
- Joining Google's ML performance team
- Hardware/Software co-design
  for TPU/GPU and Systems for ML

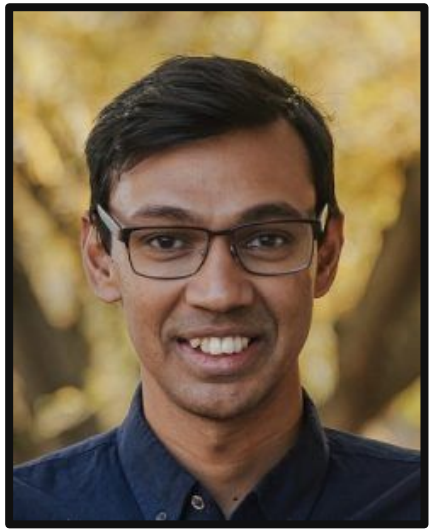


Core ML Performance

Thank you!








Thank you!

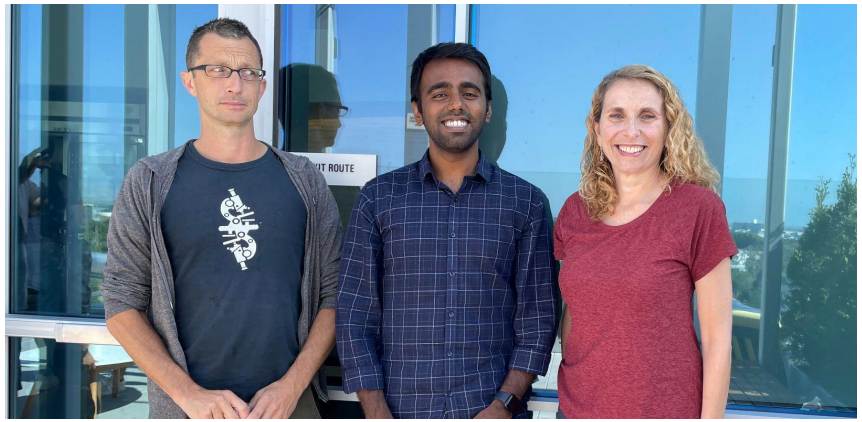
Thank you!

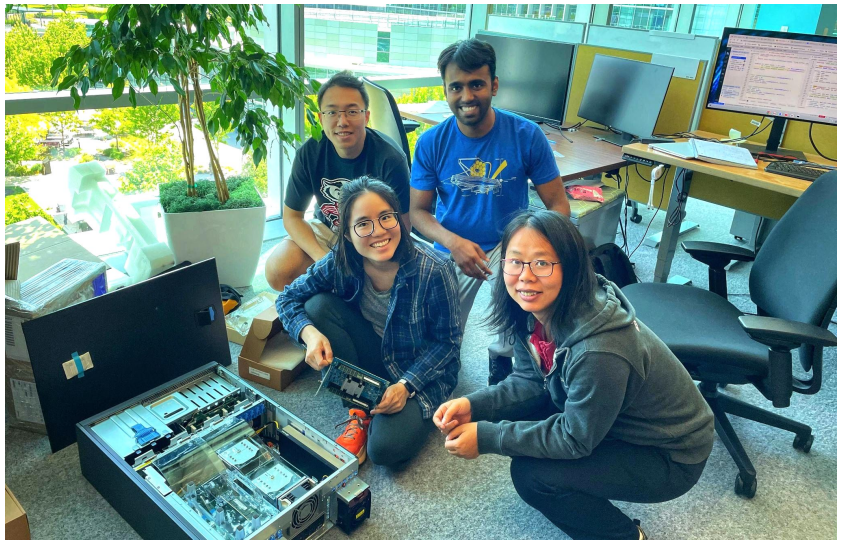

ROKA
SPRINT


OLD NAVY
NYU

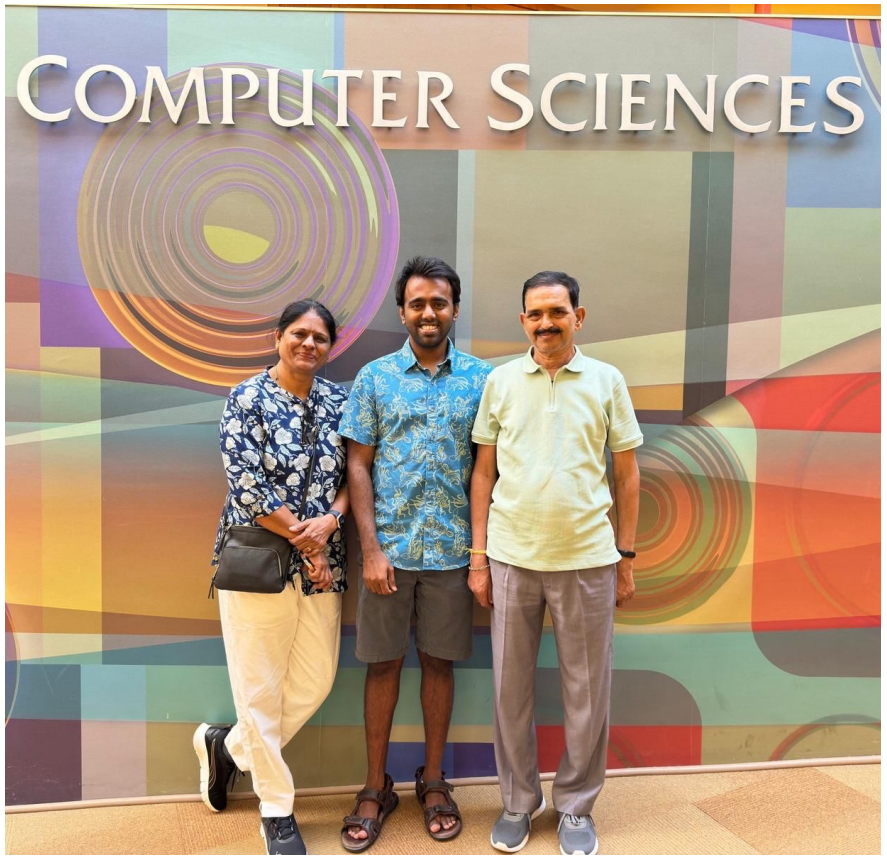

TECHNOLOGY EDUCATION FOUNDATION
DERBY

Thank you!





Thank you!



WiscSort Backup

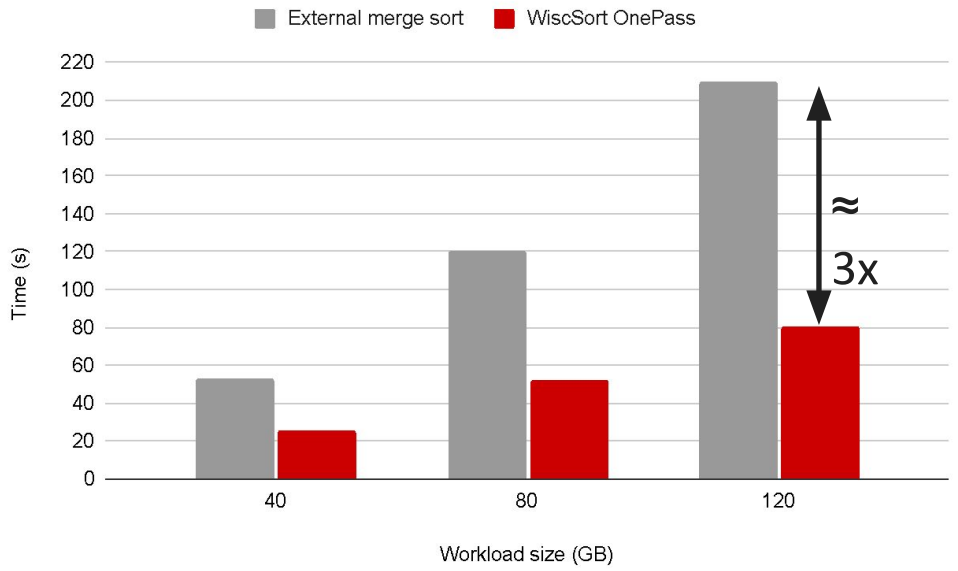
Sortbenchmark

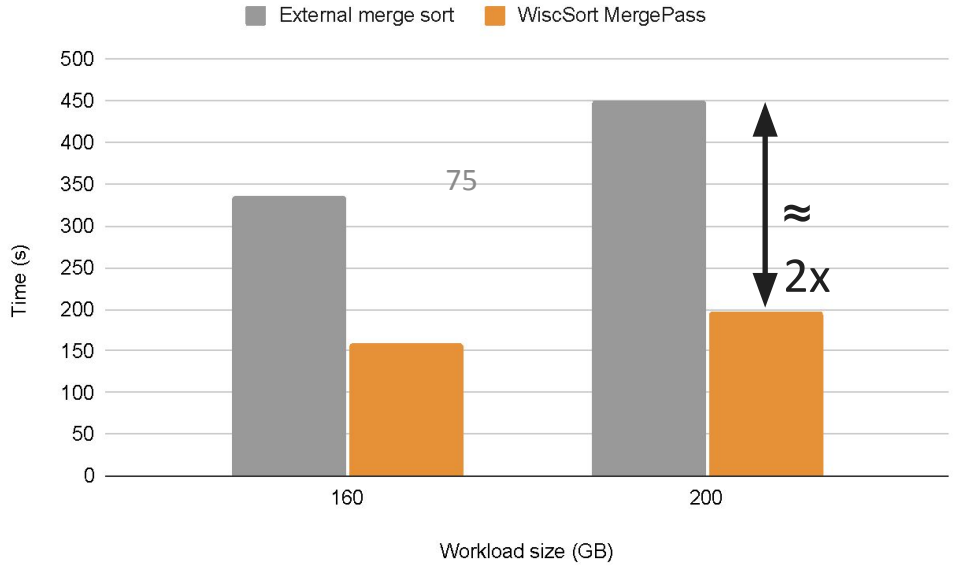
Sorting binary records of 10B key and 90B value

- ~87% of data read and written per block by external merge sort is redundant

- WiscSort OnePass is up to **3x** faster than concurrent optimized external merge sort
- WiscSort OnePass has 50% reduction in total read/write traffic and avoids the MERGE phase computation completely.

- WiscSort MergePass is up to **2x** faster than concurrent optimized external merge sort
- Reduction in I/O traffic: 42.5%



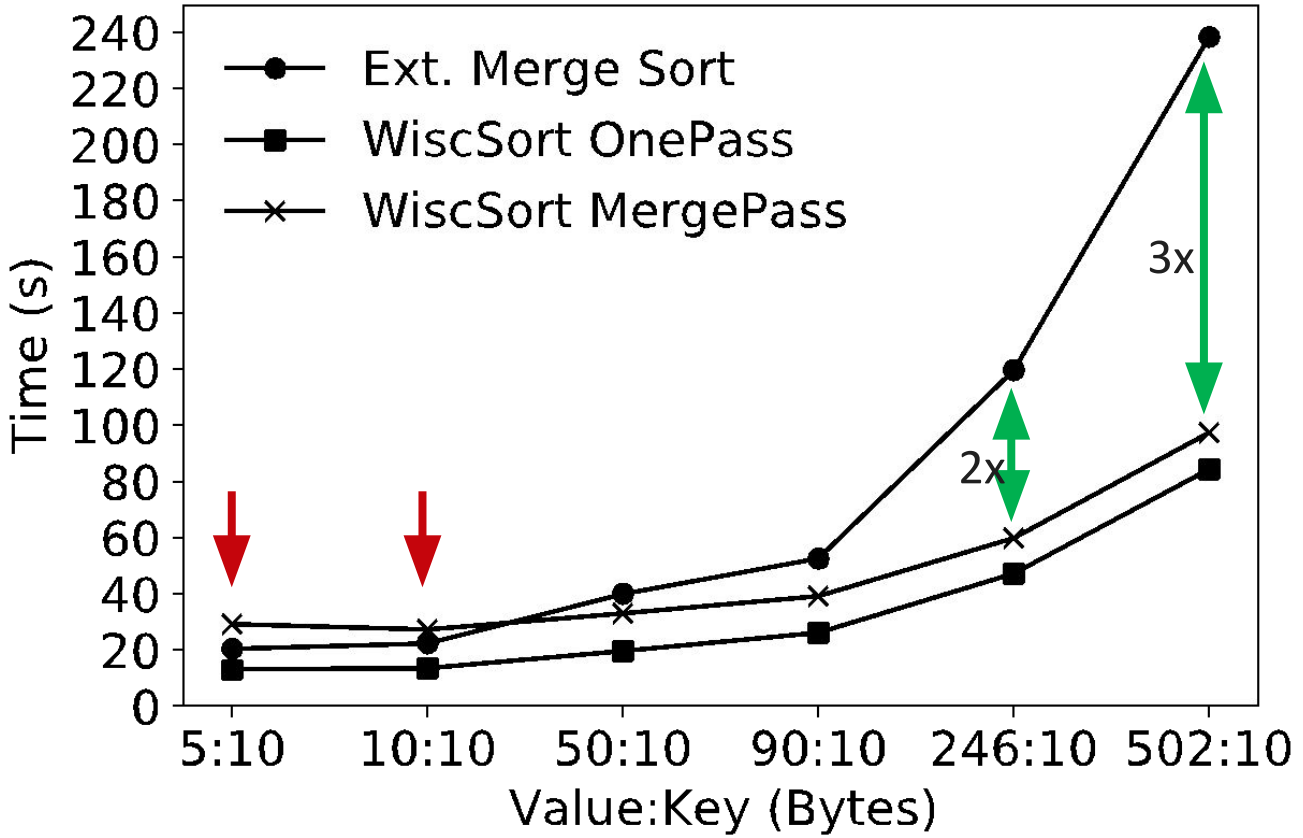


Varying record sizes

Larger $V : K$ has higher relative performance

MergePass outperforms only when $V : K > 1$

OnePass outperforms regardless of the $V : K$ ratios



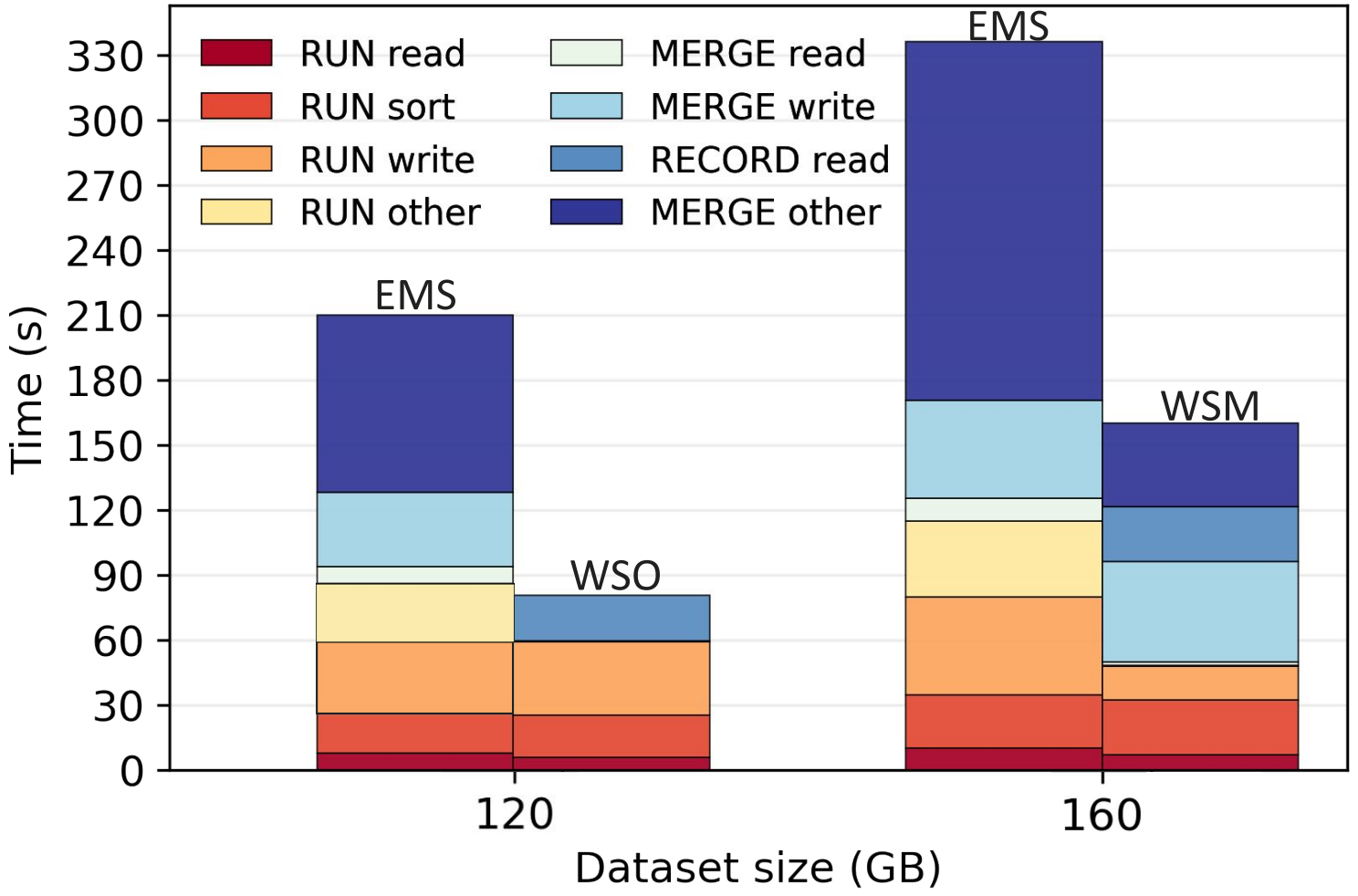
Sorting 400M records of 10B key and varying

Sortbenchmark

1. In memory sorting remains same

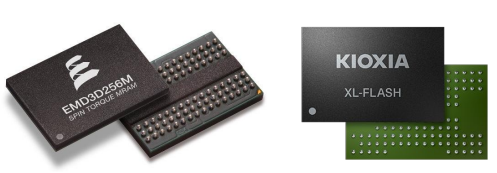
2. WSO write reduction: 100%
   WSM write reduction: 50%

3. Computational overheads reduced by being BRAID compliant

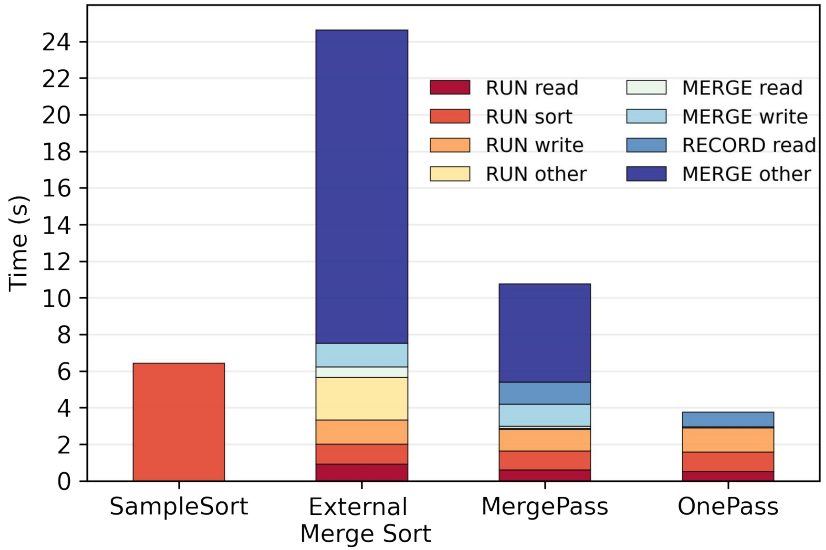


EMS = External merge sort
WSO = WiscSort OnePass
WSM = WiscSort MergePass

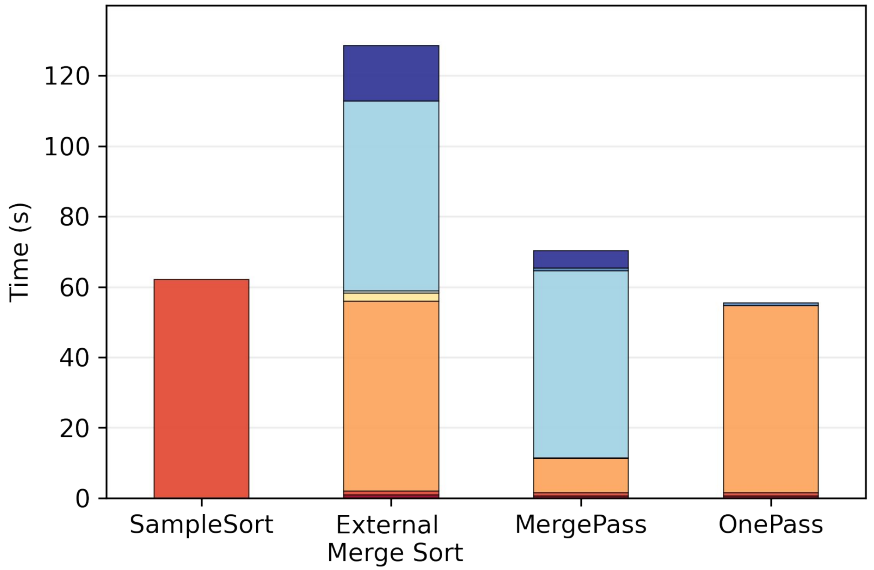
Future BRAID devices: CXL emulation



?



***BRD*-Device.** Random read is equal to sequential read.



***BARD*-Device.** Writes 500 ns slower than reads.

External merge sort is BRAID complaint on a BD device (like HDD) and WiscSort is not!

Sorting 100M records of 10B key and 90B value.

BRAID for other devices

CXL-attached DRAM (Samsung CMM-D, Micron CZ120) - **BRD Device**

- 300ns latency. 64B accesses (B)
- Moderate forms of D: tail latencies spike under contention and bw plateaus
- KV separation (B), Thread-pool control (D), random reads (R).
- Interference aware scheduling is not beneficial.

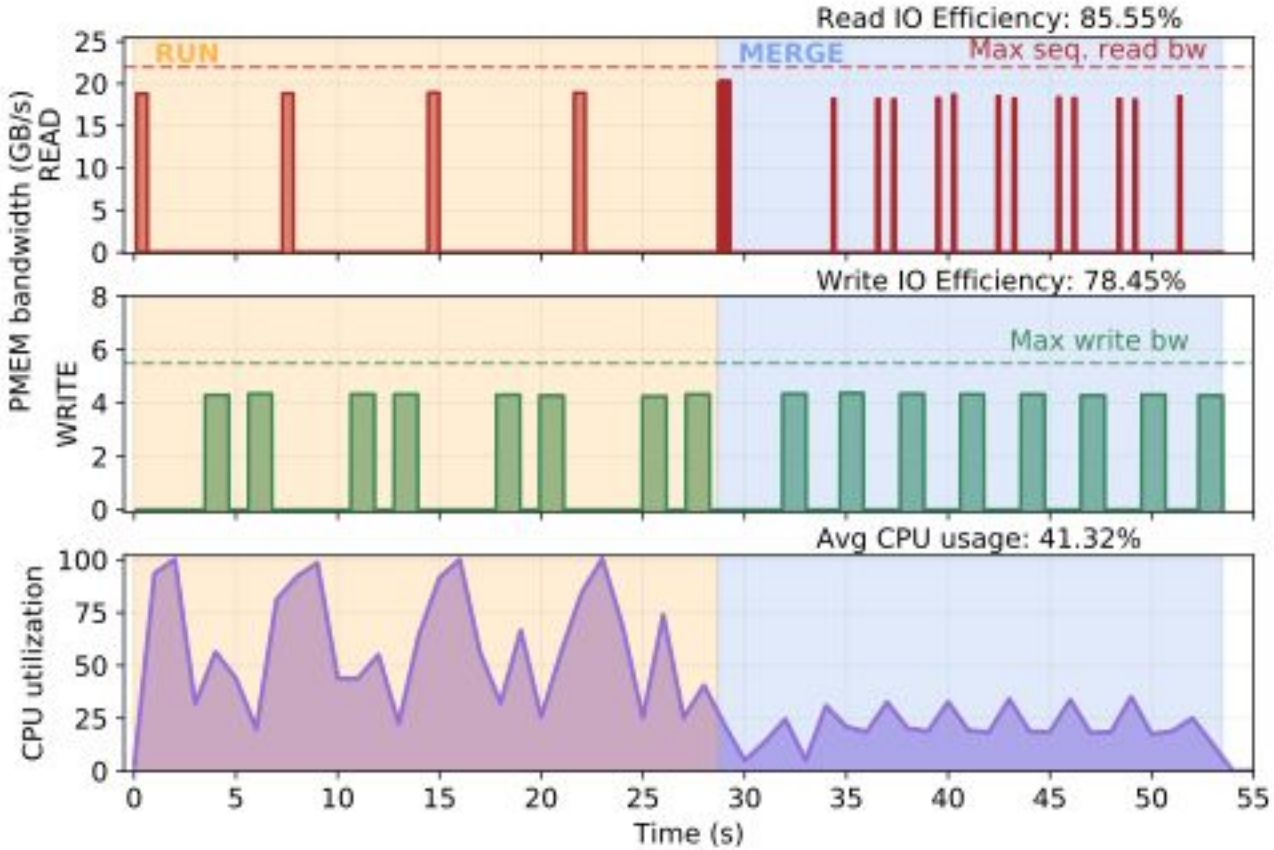
CXL DRAM +SSD Hybrid (Samsung CMM-H) - **BARD Device**

- CXL.mem (B)
- Asymmetry (cache misses fetch 4KB from NAND)
- More write reduction improves performance

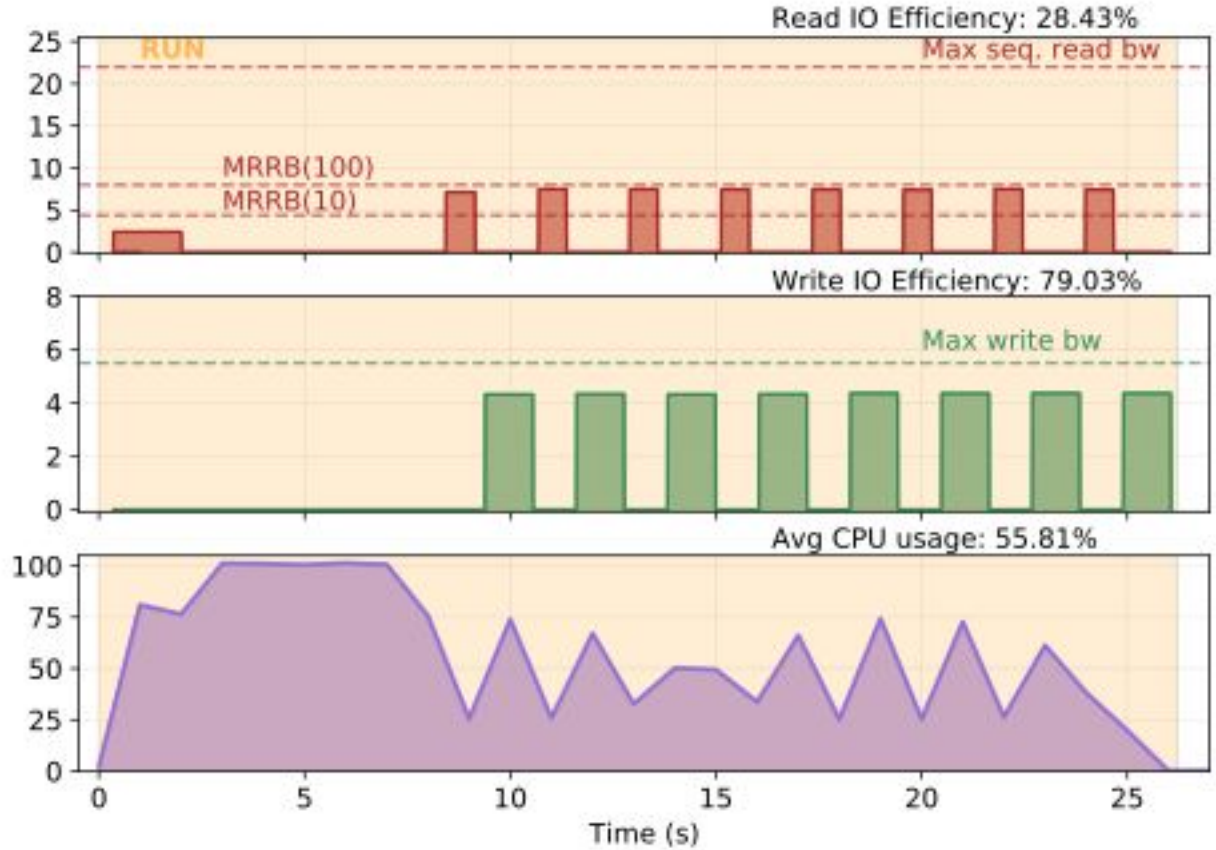
ULL SSDs (XL-FLASH, Samsung Z-SSD) - **AD** Device

- Asymmetry - reads faster than writes
- External merge sort is better

Resource utilization for sorting a 40GB file



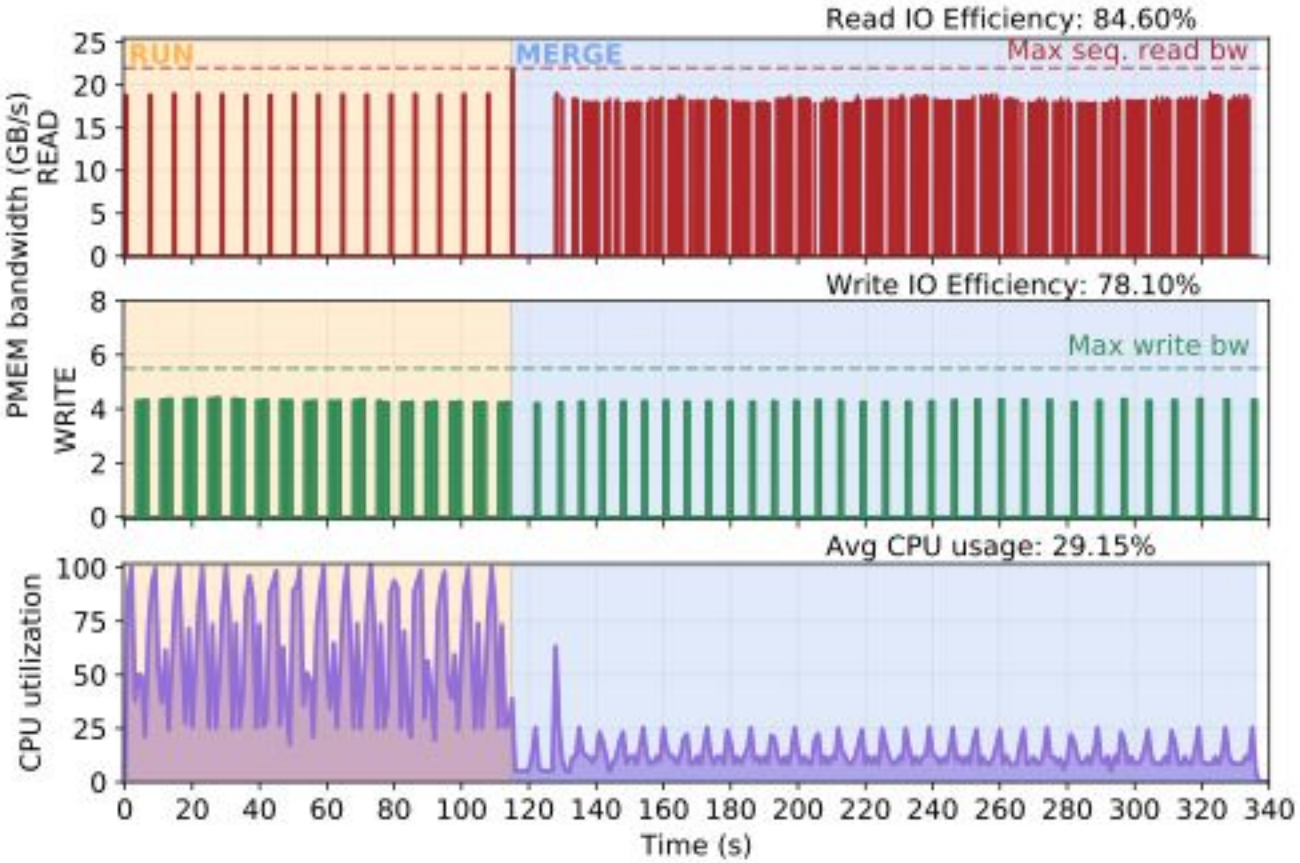
External merge sort. 10GB read buffer, 5 GB write buffer



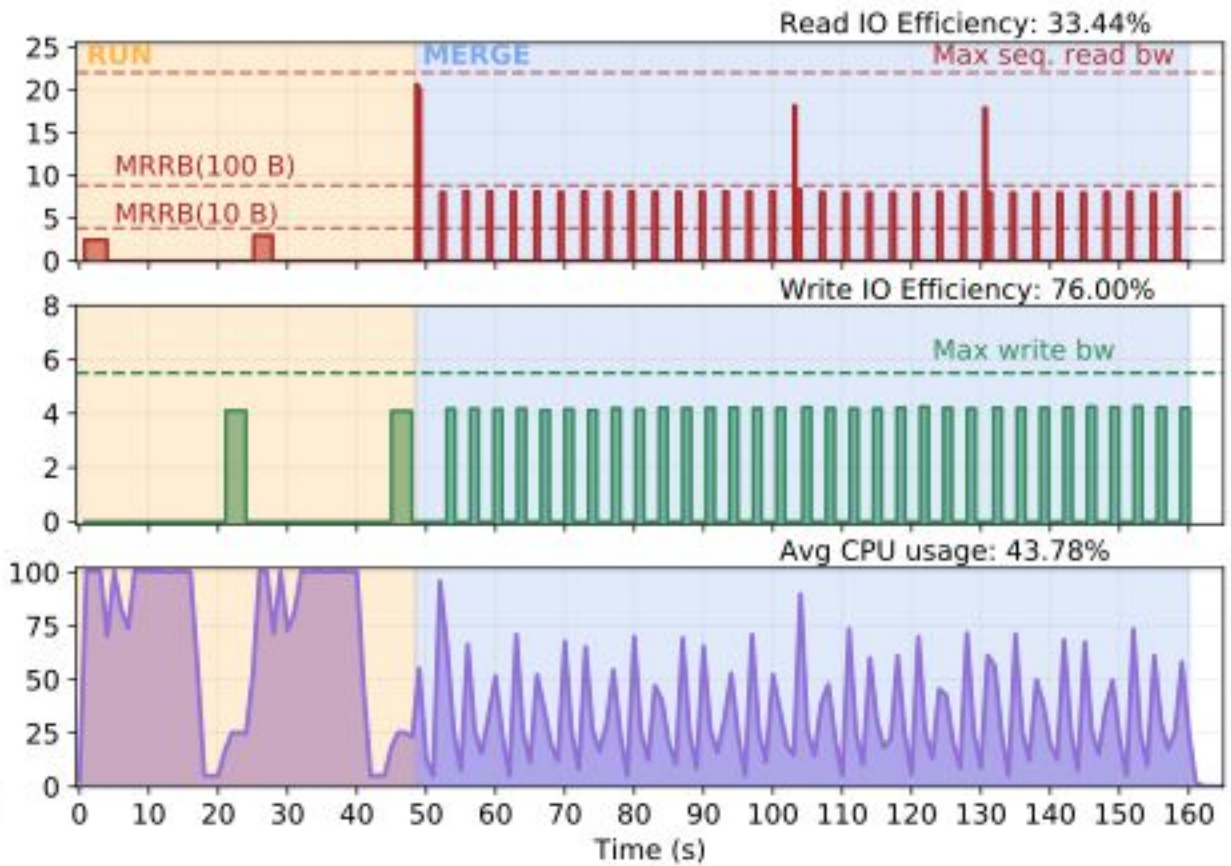
WiscSort OnePass. 5 GB write buffer

MRRB = Max Random-Read Bandwidth

Resource utilization for sorting a 160GB file



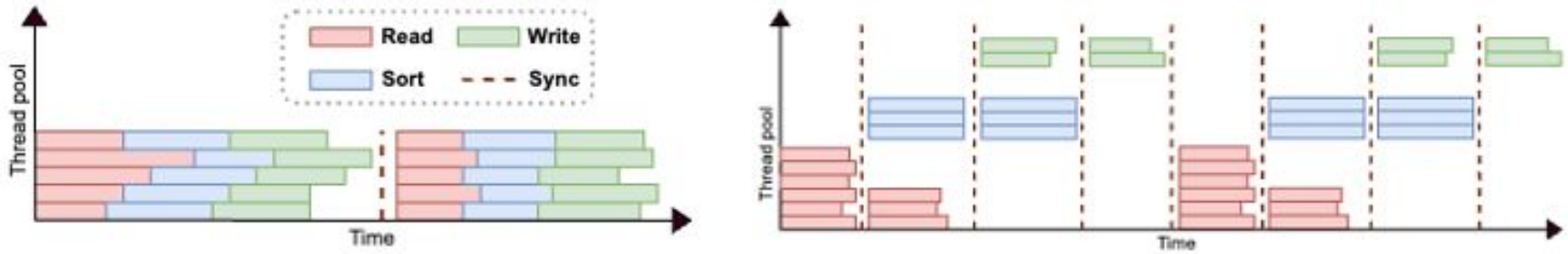
External merge sort. 10GB read buffer, 5 GB write buffer



WiscSort MergePass. 12 GB read buffer, 5 GB write buffer

MRRB = Max Random-Read Bandwidth

I/O and compute overlap overhead



(a) Synchronized uncontrolled pool

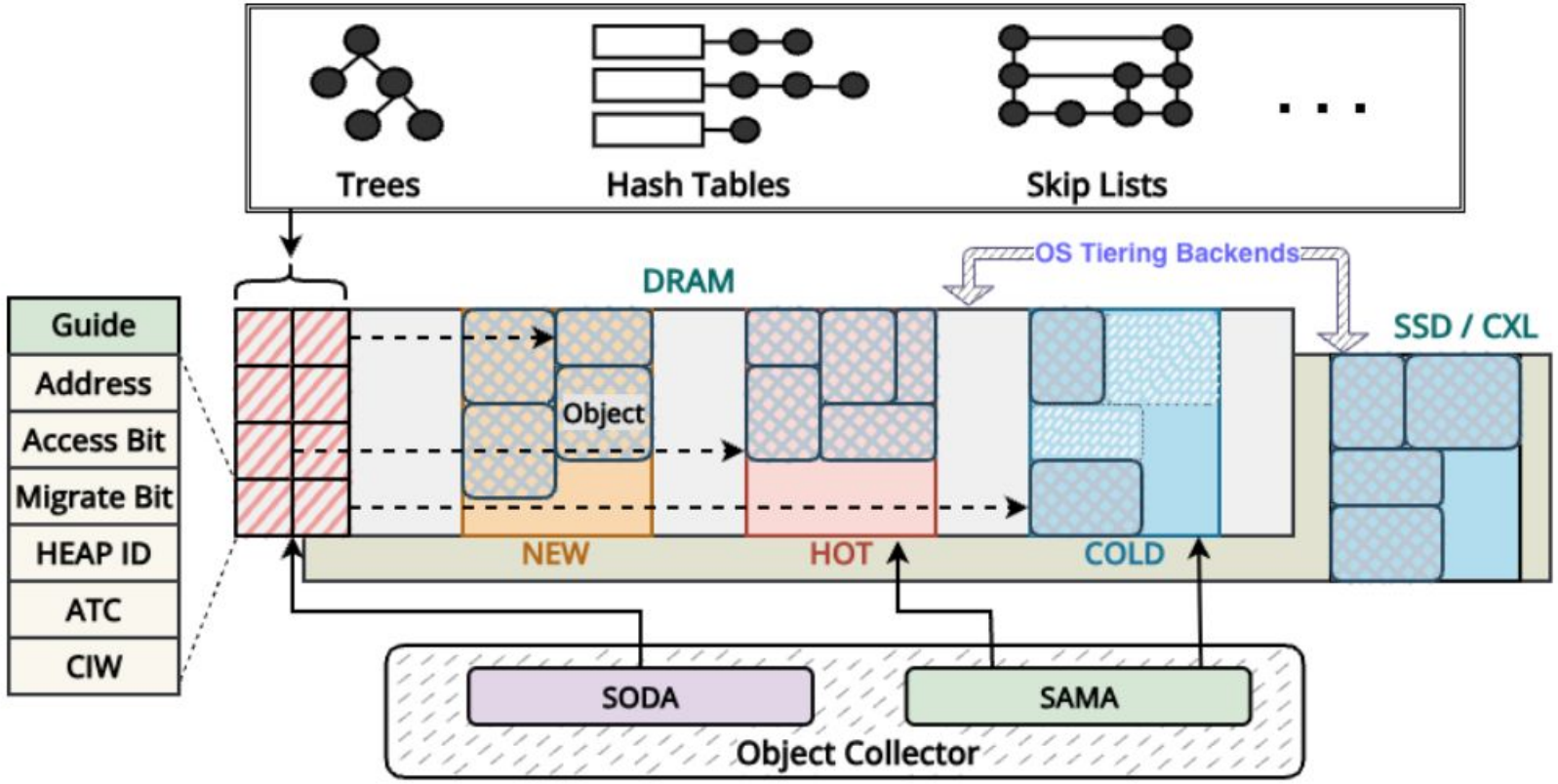
(b) Synchronized controlled pool

Figure 2: Overheads of overlapping I/O and compute. The figures exemplify why simple I/O and compute overlapping approaches may not be feasible. They suffer either from uncontrolled access between operation types to the device or require excessive locking and synchronization overhead while not achieving peak performance per operation type (E.g., read or sort).

OBASE Backup

OBASE Limitations

- **Lack of pointer stability:** Invalidates cached pointers like Abseil and STL structures
- **Unique Object Ownership:** When annotated users implicitly assert that it has exclusive access over the object. Similar to `std::unique_ptr` ownership model
- **Incompatibility with pointer arithmetic:** Unsuitable for data structures likes arrays and matrices
- **Language support restrictions:** Requires dereference overloading
- **Requirement for explicit annotations:** Users must specify which objects are managed



The Object Collector monitors access via Guides and SODA, migrates objects using a spatially-aware allocator, and presents a re-organized address space to the OS backend.

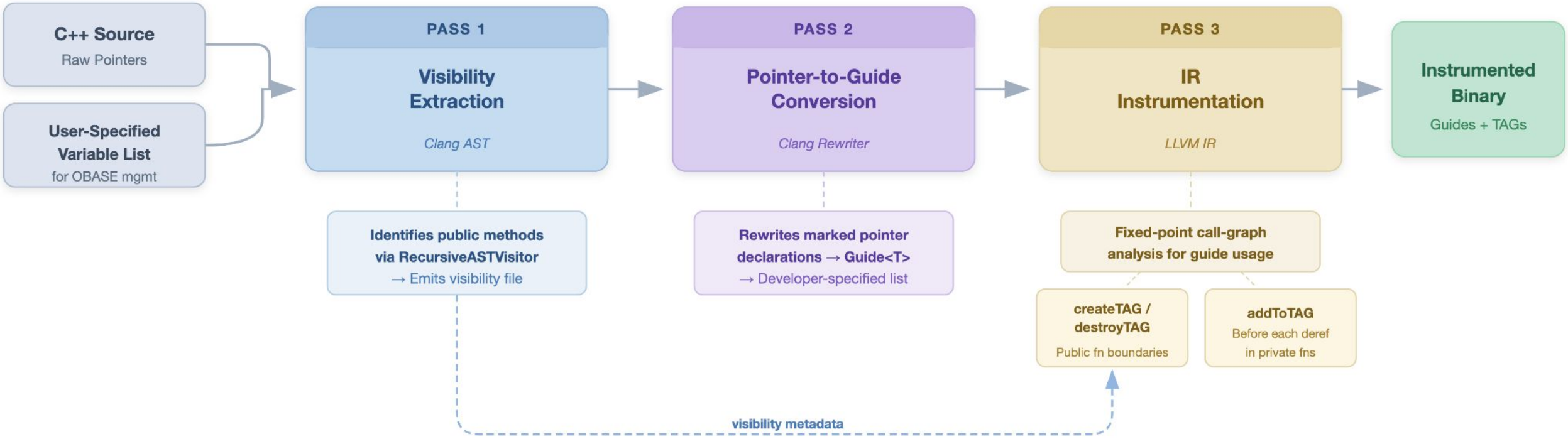
Guide Metadata Encoding

- 48 bits - Address
- 7 bits - Active Thread Count (128 concurrent threads)
- 5 bits - Consecutive Inactive Windows (32 windows): 1hr+ at 120s interval
- 2 bits - Heap ID (3 heaps)
- 2 bits - Access and Migration lock flags

Sparse Object Data Activity (SODA)

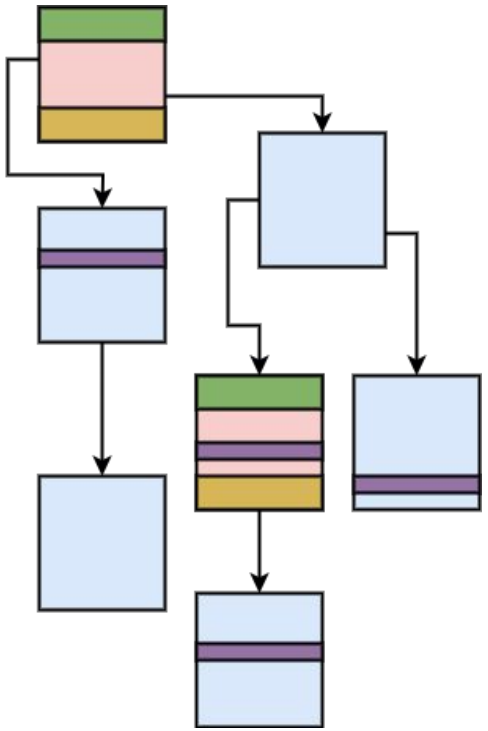
- SODA divides the address space into blocks and each block has a bunch of words. Each word is a 64-bit integer, and each bit represents 8 bytes of the address space.
- SODA tracks guide pointers rather than object addresses, it remains valid across relocations - the guide’s address does not change when object moves.
- With 128 KiB block size and 2^44 gbits, the lower bound is 40 MiB and the upper bound is 682 GiB.

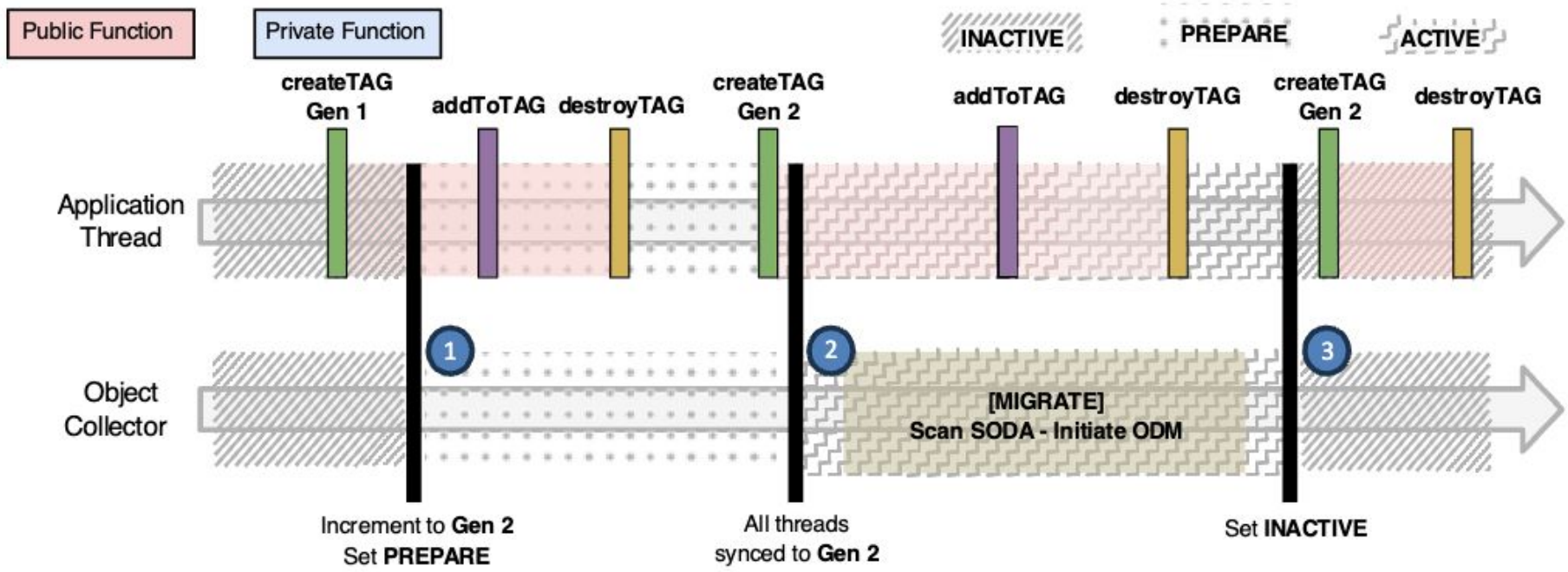
OBASE Compiler Passes for Guide Management



OBASE Compiler Passes for Guide Management

- **Pass 1 - Visibility Extraction (Clang Front-End)**
  - A `RecursiveASTVisitor` inspects C++ class definitions to identify public methods (e.g., get, set, delete) that serve as data-structure entry points.
  - Outputs a visibility file consumed by Pass 3, so TAGs are only created at public boundaries.
- **Pass 2 - Pointer-to-Guide Conversion (Clang Rewriter)**
  - Developers provide a list of pointer field names to convert; the pass rewrites their declarations and usages into `Guide<T>`.
  - Gives precise developer control over which objects OBASE manages while automating all syntactic changes.
- **Pass 3 - IR Instrumentation (LLVM IR)**
  - Performs a fixed-point analysis: first scans for direct guide usage, then propagates through the call graph to discover indirect usage.
  - Combines this with Pass 1's visibility data to classify every function, then inserts:
  - `createTAG` / `destroyTAG` at entry/exit of public functions that touch guides.
  - `addToTAG` before each guide dereference in private functions (no new TAG creation).

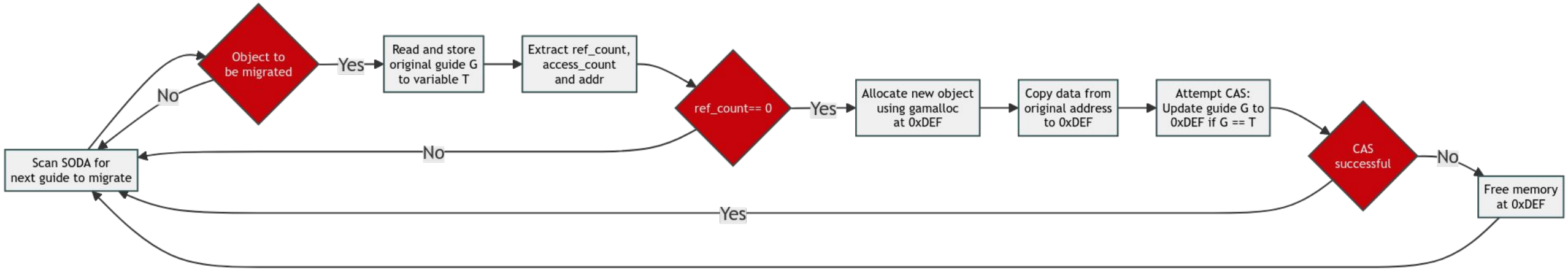




Start of Public Function	Guide Access	End of Public Function
createTAG: publicNestingLevel++ TAI[slot] = 1 threadLocalGen = GlobalGen	if (TAG_state != INACTIVE && addToTAG(Guide)): incrementATC(Guide)	destroyTAG: publicNestingLevel-- if (!publicNestingLevel): decrementAndClear(TAG) TAI[slot] = 0

The left side shows the instrumented call graph, code is inserted in three scenarios as shown. Public functions create and destroy Thread-local Active Scope Guards (TAGs), maintaining nesting levels and registering the thread, in the Thread Activity Index (TAI). Guides increment active reference counts only if added to the TAG. Reference counts are decremented only when the outermost public function exits. Active Thread Count (ATC) is enabled only during PREPARE and ACTIVE states

Optimistic Data Migration - ACTIVE



Algorithm 1 Optimistic Data Migration (ODM)

```
1:  procedure MigrateObject(ptr, targetHeap)
2:      guide ← AtomicLoad(ptr)                          ▷ Load object guide
3:      if RefCount(guide) > 0 then
4:          return false                                 ▷ Object in use
5:      end if
6:      srcAddr ← ExtractAddr(guide)
7:      srcHeap ← ExtractHeapType(guide)                 ▷ Get current heap type
8:      SetMigrationLock(ptr)                            ▷ Mark as being migrated
9:      size ← GetSize(srcAddr)
10:     dstAddr ← samalloc(size,targetHeap)
11:     if dstAddr = null then
12:         ClearMigrationLock(ptr)
13:         return false                                 ▷ Allocation failed
14:     end if
15:     Copy(dstAddr,srcAddr,size)                       ▷ Copy data
16:     newGuide ← CreateGuide(guide,dstAddr,targetHeap)
17:     ClearMigrationLock(newGuide)
18:     success ← AtomicCAS(ptr,guide,newGuide)
19:     if success then
20:         Free(srcAddr,srcHeap)                        ▷ Release old memory
21:         return true
22:     else
23:         Free(dstAddr,targetHeap)                     ▷ Rollback
24:         ClearMigrationLock(ptr)
25:         return false                                 ▷ Migration failed, guide changed
26:     end if
27: end procedure
```

Time	Actor	Action	ATC	Lock	Outcome
t_0	OC	read guide	0	0	eligible
t_0'	OC	CAS: set lock	0	1	copying begins
t_1	Thread	dereference	1	0	lock cleared
t_2	OC	CAS: commit	*mismatch*		**aborted**
If no thread intervenes between t_0' and t_2:					
t_2'	OC	CAS: commit	0	0	**success**

Table 5.1: **Race between migration and concurrent access.** A dereference at t_1 modifies the guide, causing the OC's commit CAS to fail. When no thread intervenes, both CAS operations succeed and the object moves.

Tiering background

AutoNUMA - *Reactive, recency-based, no demotion*

- Default Linux NUMA balancing; periodically unmaps a sample of pages and uses the resulting minor faults as a hotness signal.
- A single access is enough to trigger promotion - works well when pages are clearly hot or cold, but over-migrates under transient or shifting access patterns.
- Has no proactive demotion path; pages only leave the fast tier when general memory pressure forces standard reclamation.

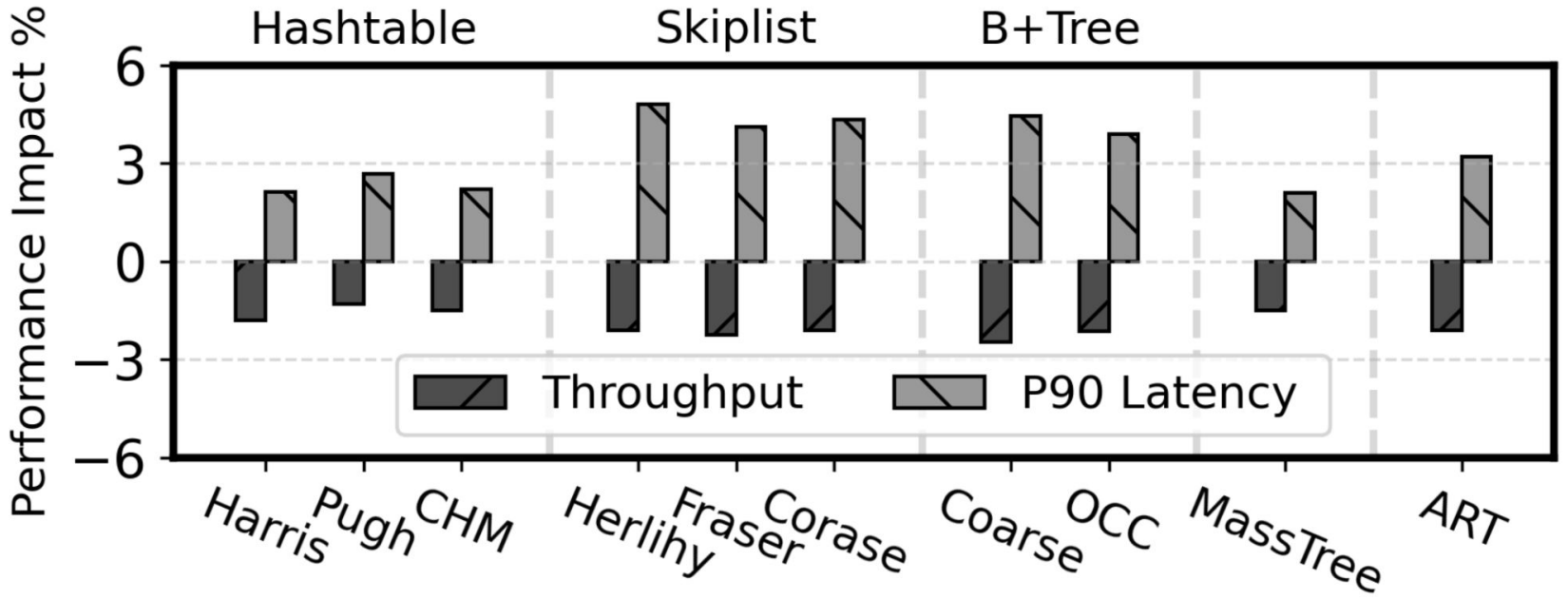
TPP (Transparent Page Placement) - *Production-tuned, hysteresis filtering*

- Demotes cold pages via migration, preserving byte-addressable access on the slow tier.
- Decouples allocation from reclamation with separate watermarks, keeping headroom in the fast tier for promotions without stalling allocators.
- Adds a hysteresis check on top of NUMA hint faults: a page must be in the active LRU list *and* fault again to be promoted, filtering out one-off accesses that trip up AutoNUMA.

Memtis - *Adaptive threshold, hardware-sampled*

- Uses Intel PEBS (Processor Event-Based Sampling) instead of page faults, giving low-overhead access counts at sub-page granularity.
- Maintains a histogram of access counts in exponential bins and dynamically computes a hot threshold so the fast tier is always filled with exactly the hottest pages - no static "promote after N accesses" cutoff.
- Can split huge pages and promote only the hot 4 KiB subpages, reducing fast-tier waste from mixed-temperature huge pages.

Performance overhead

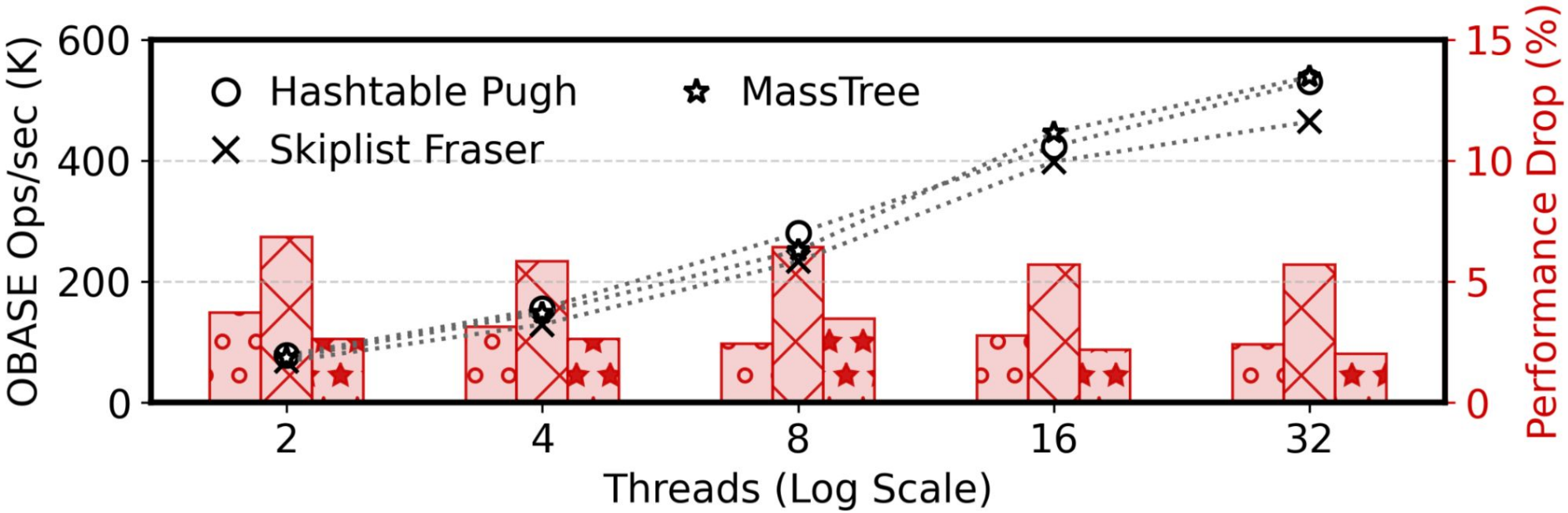


Throughput and p90 latency overhead across data structures

The overhead has two main sources:
(1) A tagged-pointer read-modify-write on each guide dereference
(4-5 ns, comparable to an L1 cache hit)
(2) TAG/ATC bookkeeping during ACTIVE migration epochs.

The Object Collector runs in a dedicated thread and consumes less than 5% of CPU time.

Scalability

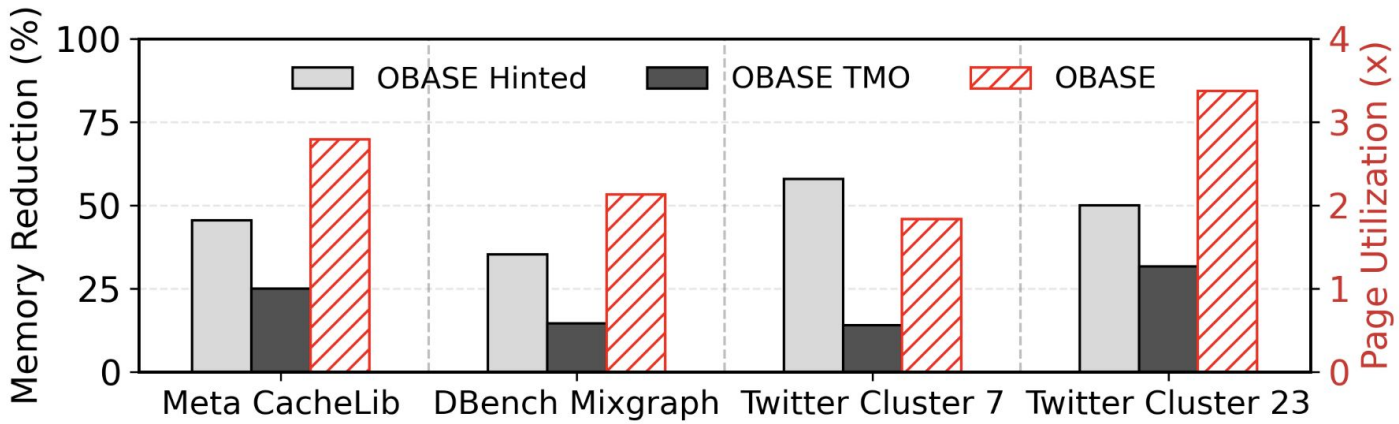


Bars show absolute throughput; markers show overhead relative to baseline.
Overhead remains bounded at 1–8% with no upward trend.

Guide metadata updates target per-object state (no cross-thread contention), the test-and-set optimization skips redundant writes for hot objects, TAGs are thread-local, and ATC increments occur only during brief ACTIVE epochs.

Production trace results

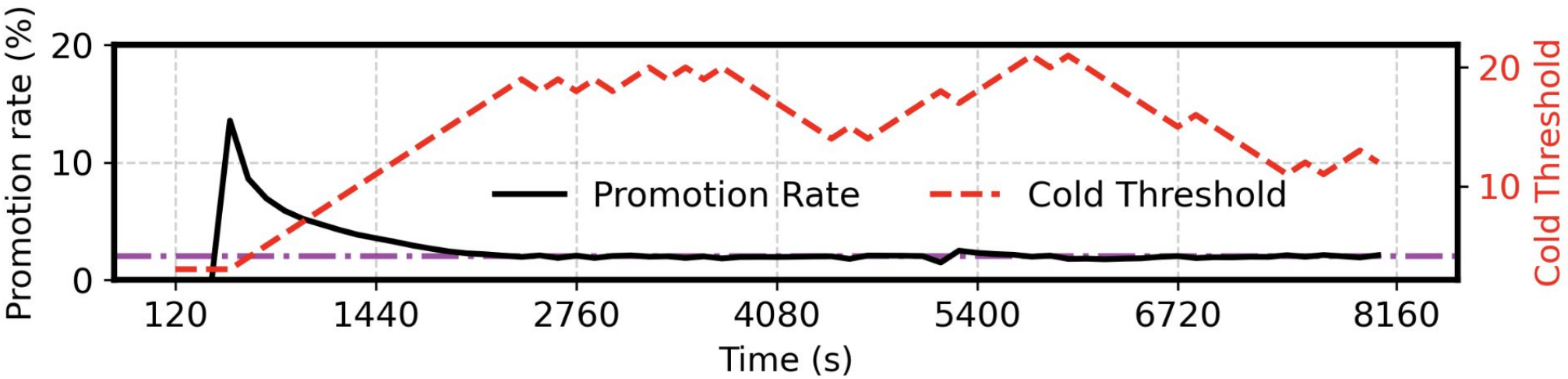
- **Meta CacheLib:** Read-heavy (83% GET) with gradually shifting popular keys.
- **DBench Mixgraph:** Models Meta's ZippyDB with key-range locality across 30 prefixes. Read-heavy (85% GET, 14% PUT, 1% SEEK).
- **Twitter Cluster 7:** High skew (α=1.07) with small, concentrated working set of reads and writes.
- **Twitter Cluster 23:** Write-heavy (31% SET, 30% INCR) with low skew (α=0.274) and deletes.



OBASE Hinted reduces RSS by 36–58% compared to no reclamation

Adding OBASE to TMO provides 15–30% additional savings

Cold threshold adaptation (Meta CacheLib)



Additive Increase Additive Decrease of Cold Threshold

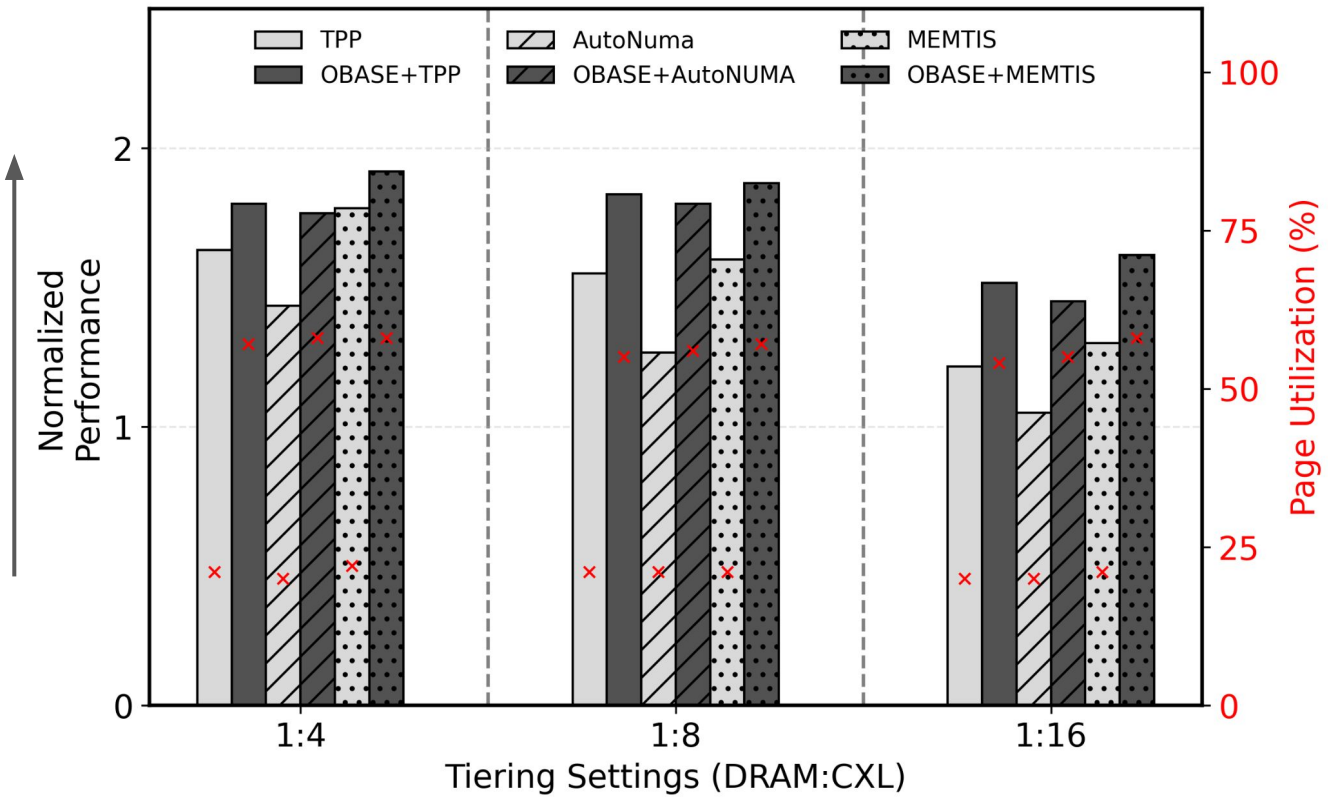
PR = (unique COLD pages accessed / working set size) × (60 / scan interval)
Target Promotion Rate = 1% (based on CXL tiering deployments)

Backend Validation (Tiering)

YCSB-B with ~67 GiB footprint (50M - 30B key and 1KiB value)

- Tiering backends: TPP, AutoNuma, MEMTIS
- DRAM:CXL ratio across three configurations: 1:4 (14.8 GiB DRAM), 1:8 (7.4 GiB DRAM), and 1:16 (3.9 GiB DRAM)
- Without OBASE hotset spans 16.3 GiB pages but with OBASE it spans 6.33 GiB

OBASE at ratio 1:X performs comparably to baseline at 1:(X/2)



Related Work

- **Object-Level Management:**
  - AIFM and MIRA operate at object granularity but focus on far-memory over RDMA, requiring direct hardware access that limits production adoption
  - Alaska uses handle-based indirection to reduce RSS through heap compaction, addressing fragmentation reactively without object hotness classification
- **Runtime vs. Allocation-Time Placement:** Allocation time hinting approaches fail to capture objects transitioning between hot and cold states or distinguish between objects from the same allocation site with different access patterns
- **Page-Level Optimizations:** TPP, HeMEM, TMTS, MEMTIS, etc


WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON