Load-Site-Based Filtering of Transiently Hot Objects to Reduce the Effective Working Set

Naoki Nakanishi Takato Hideshima Tomoharu Ugawa

The University of Tokyo

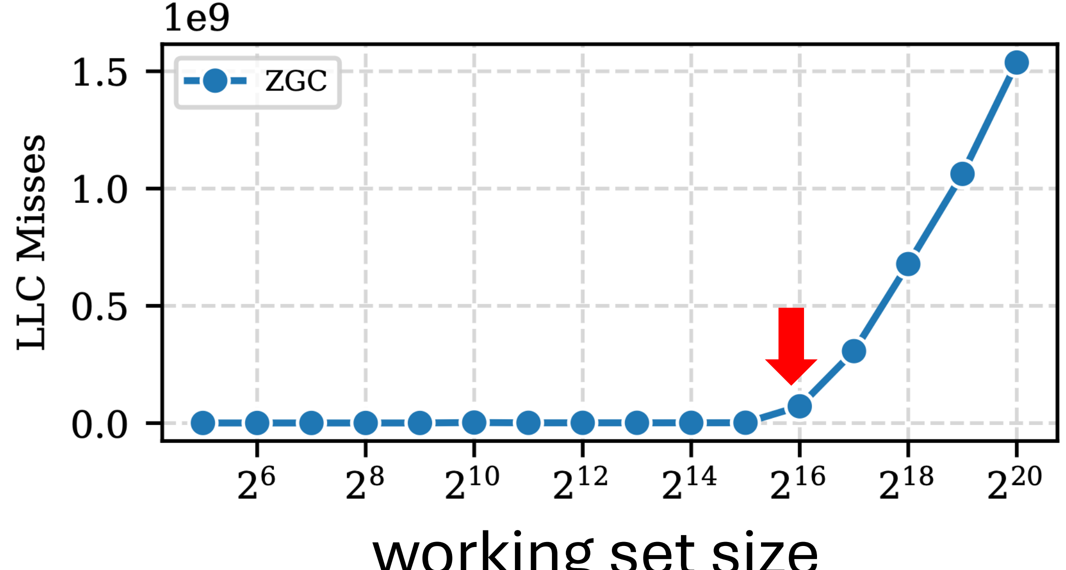
How to Reduce Working Set of Java Programs

Naoki Nakanishi Takato Hideshima Tomoharu Ugawa

The University of Tokyo

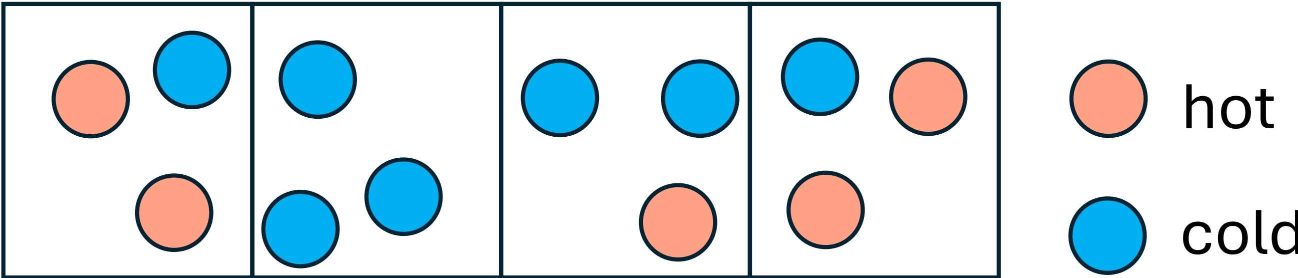
Cache Memory

- Modern CPUs are equipped with multi-level cache memory
- When working set exceeds the cache capacity, cache misses increases
  - Degrades performance



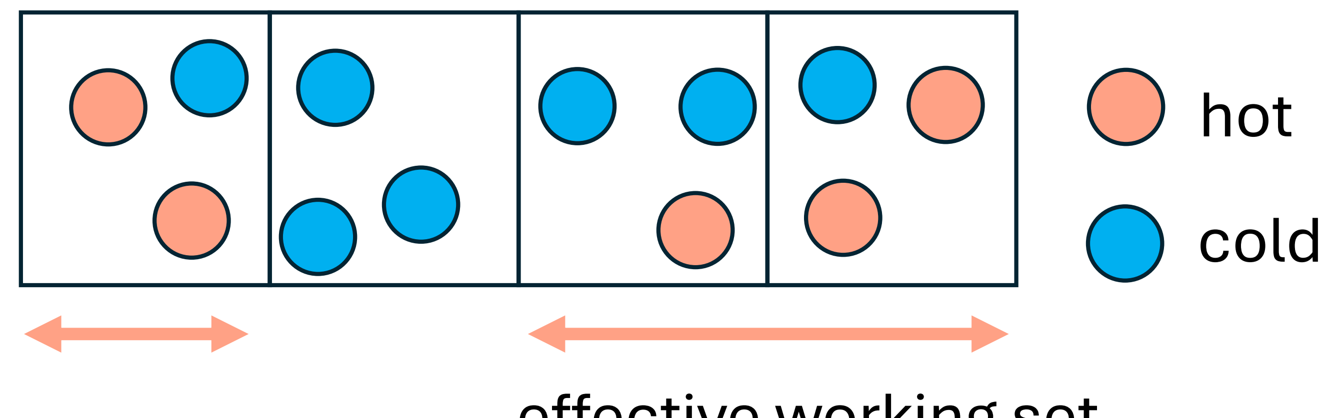
Effective Working Set

- Program frequently accesses a subset of objects
  - hot objects



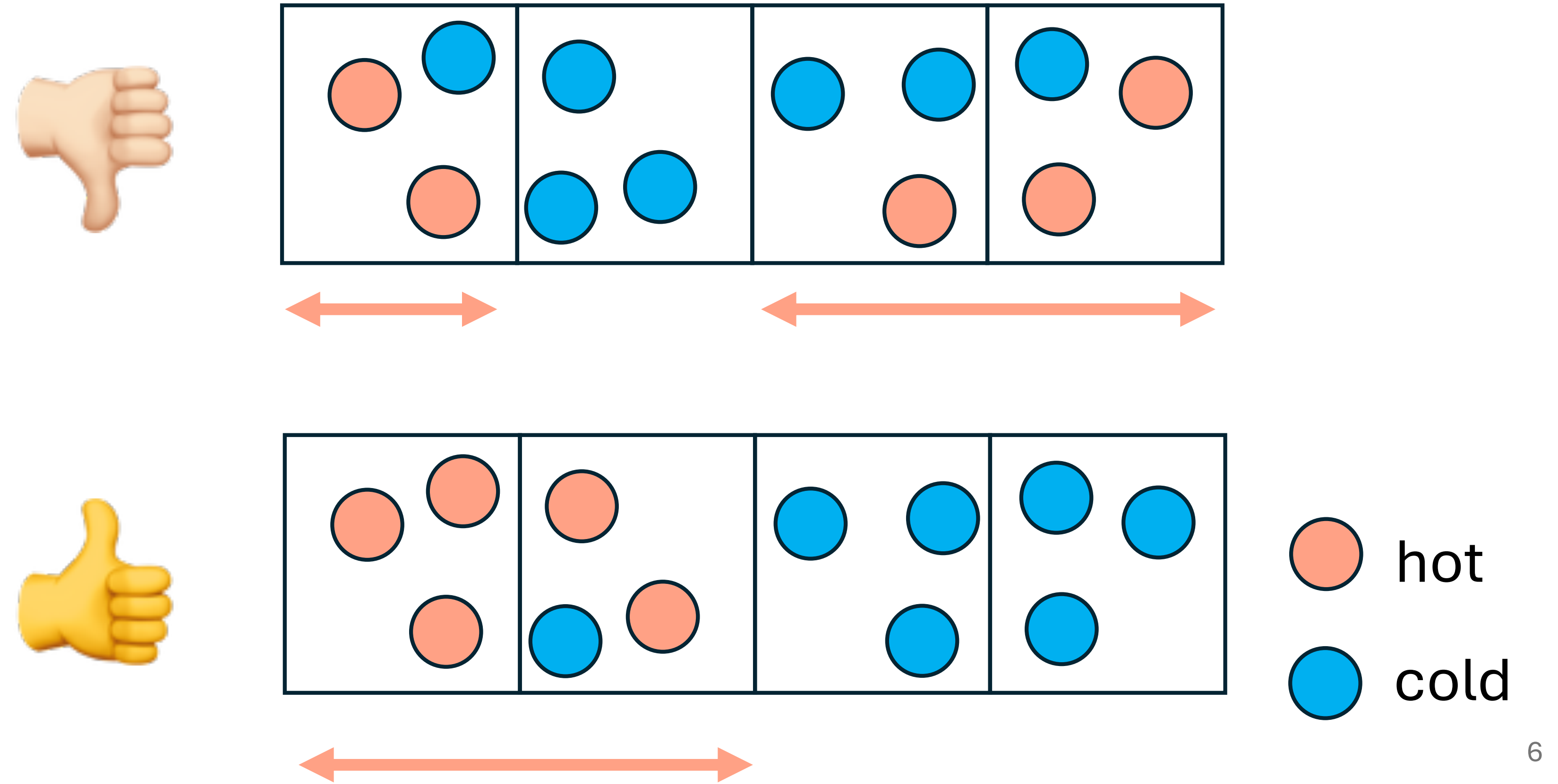
Effective Working Set

- Program frequently accesses a subset of objects
  - hot objects
- Memory blocks containing hot objects are cached
  - effective working set
- Cold objects in the same block occupy cache



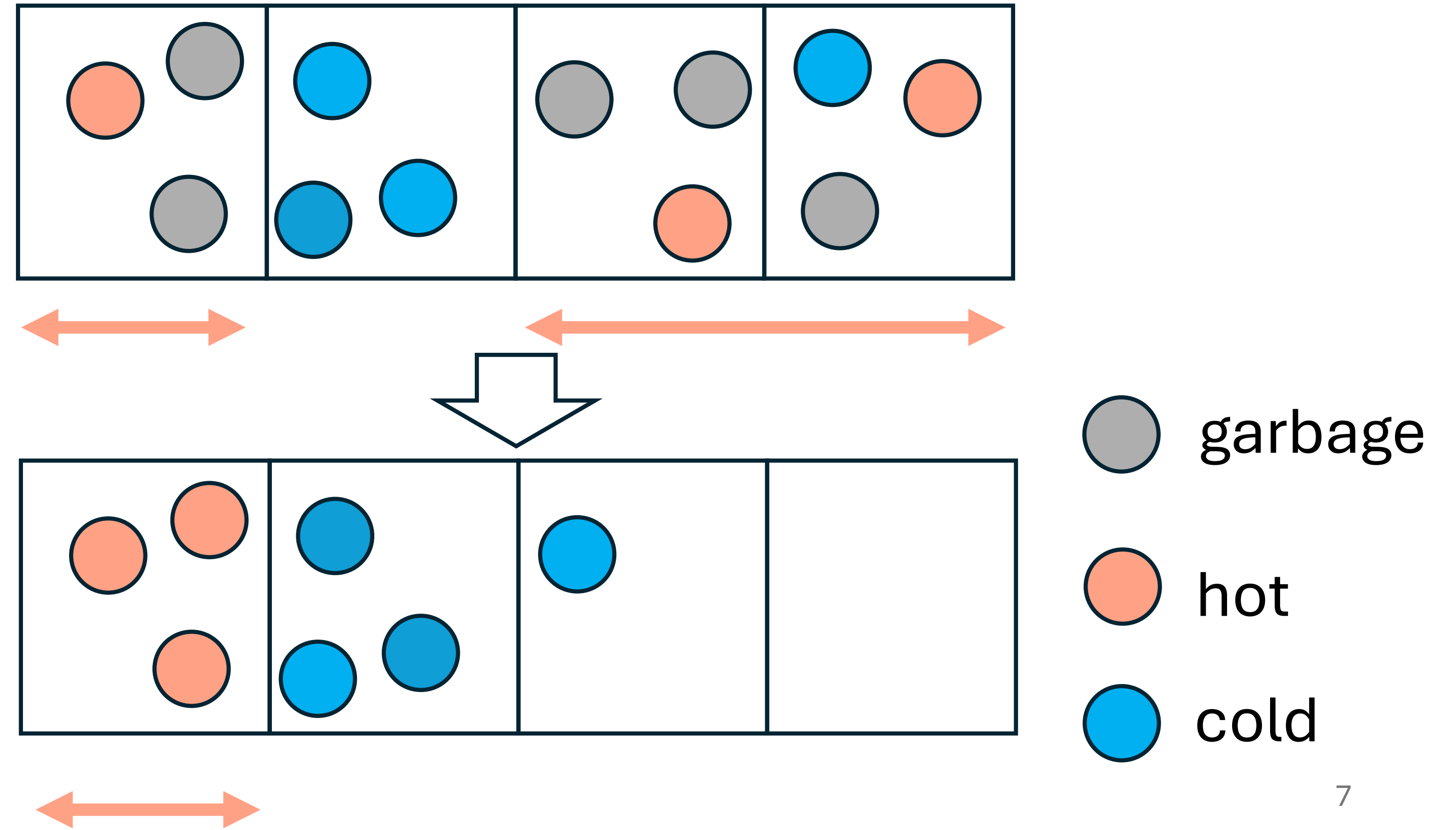
Cache Friendly Object Arrangement

- Placing hot objects close together reduces effective working set size



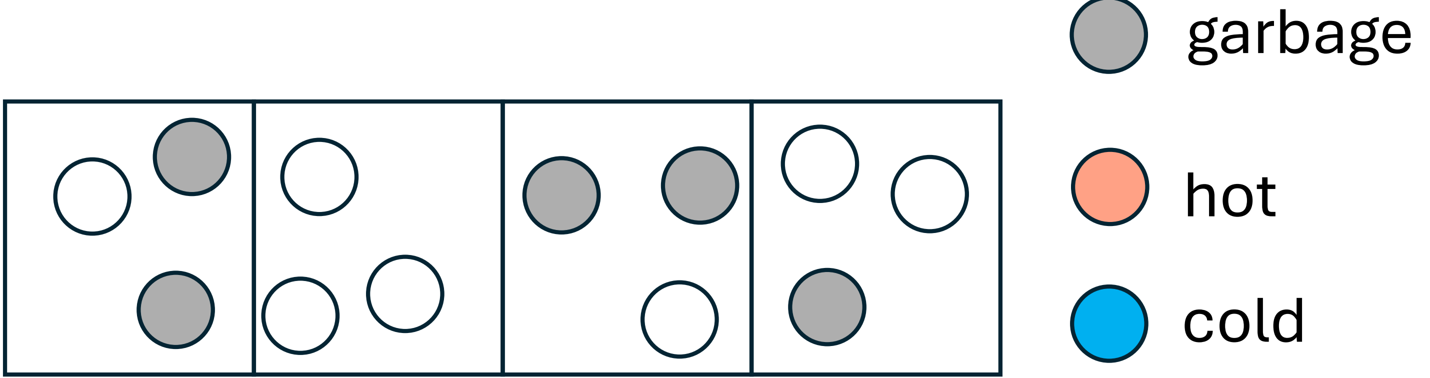
Motivation

- In managed languages, GC can rearrange objects
- Move hot objects close together



Challenge

- How to identify hot objects?
  - Accurate prediction of future object accesses
  - Low overhead

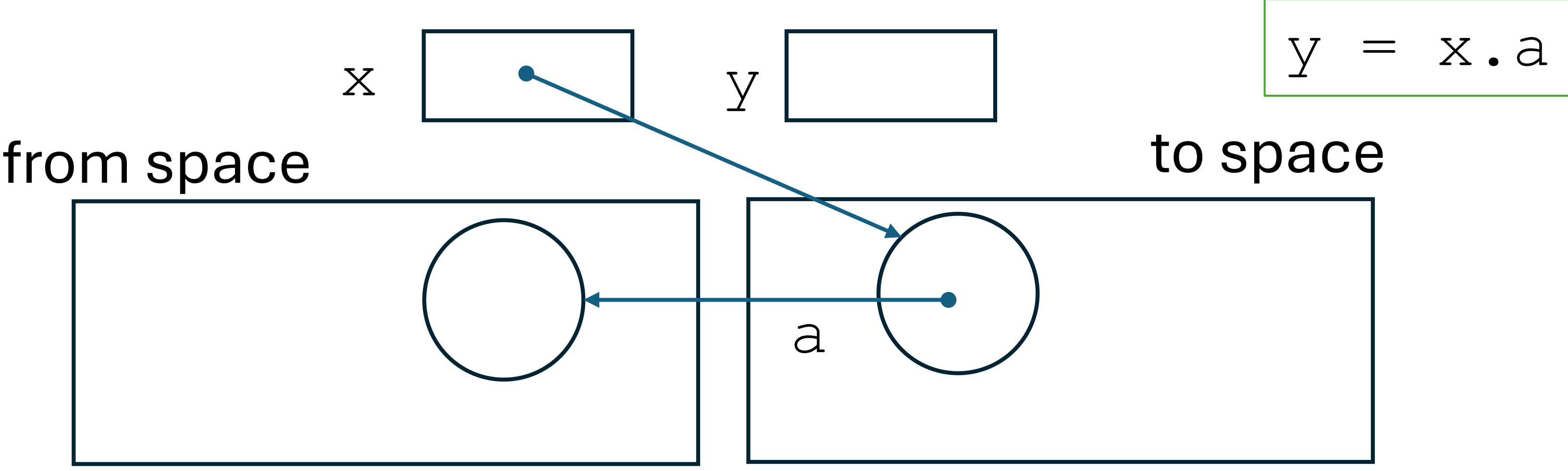


HCSGC [Yang+ 2020]

- GC built on top of ZGC
- Segregates hot objects from cold objects
- Leverages load reference barrier (LRB) to predict future accesses
- Predict hotness of an object when it is moved
  - no per-object housekeeping metadata (e.g., profiling counter)

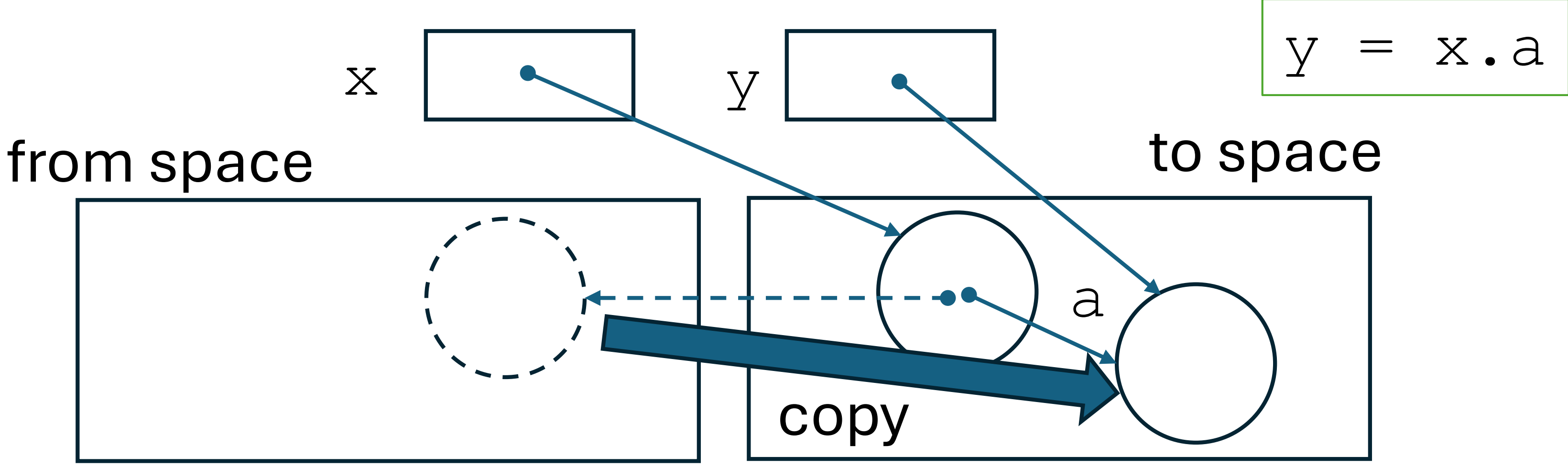
Load Reference Barrier

- Read barrier for concurrent copying
  - Invoked when mutator loads a reference



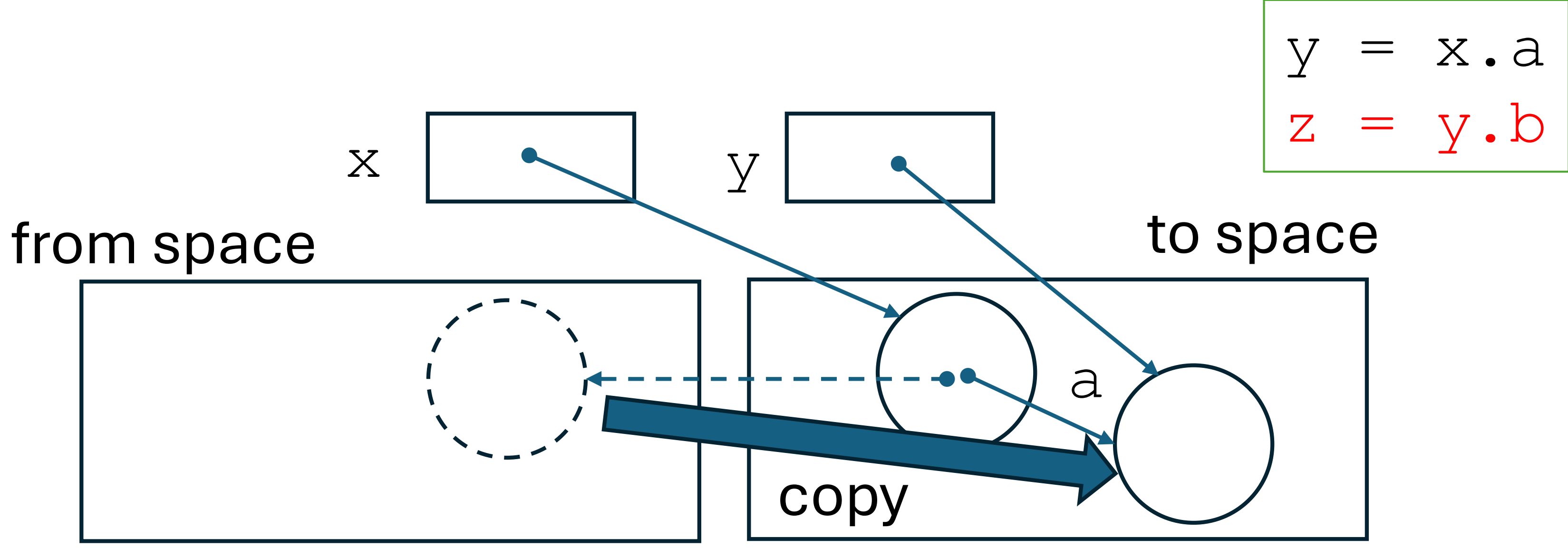
Load Reference Barrier

- Read barrier for concurrent copying
  - Invoked when mutator loads a reference
- LRB copies its referent if it has not been copied
  - Ensure all the references in the root set point to copied objects



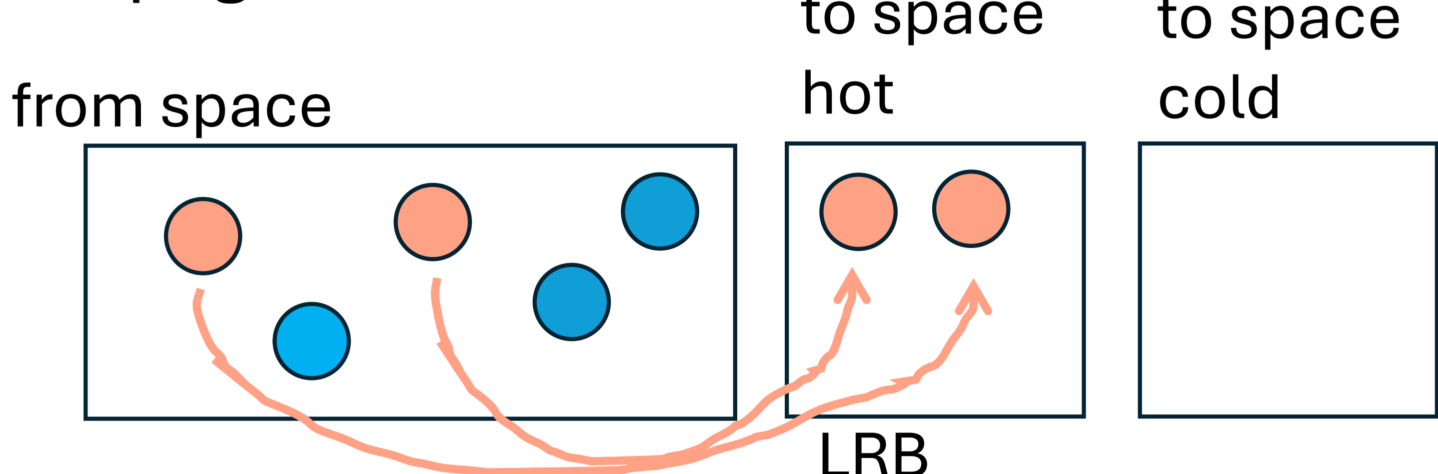
LRB's Implication

- LRB reflects mutator's behavior
- LRB tends to copy hot objects
  - Object whose ref is loaded is about to be accessed
  - Once accessed, it is likely to be accessed again soon



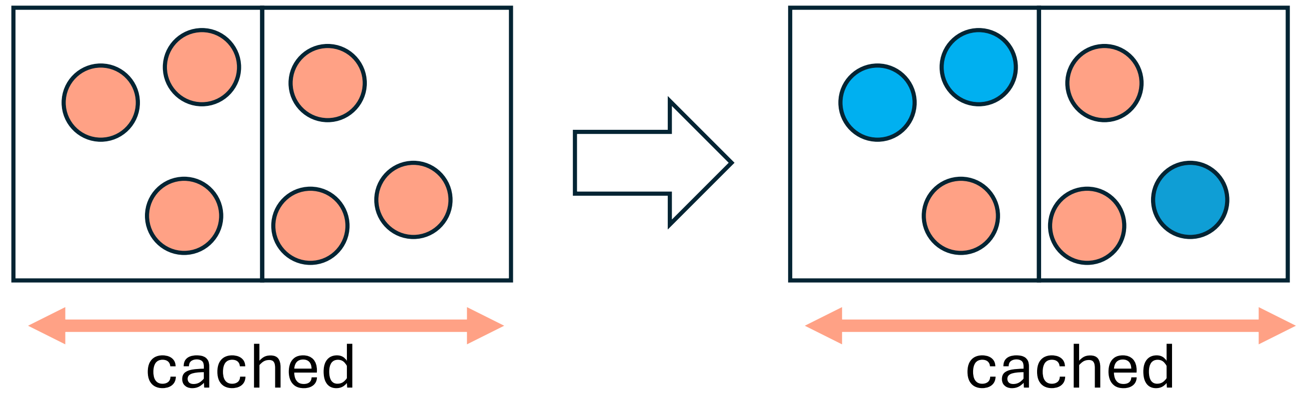
HCSGC

- Segregates objects copied by LRB from others
  - Hot pages
- Maximize opportunity for LRB to copy objects
  - Keeps LRB active even when GC is inactive
  - Treats all pages as from-space
- All objects whose refs are loaded are moved to hot pages



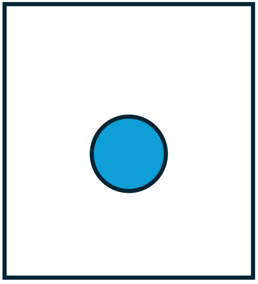
Possible Problem

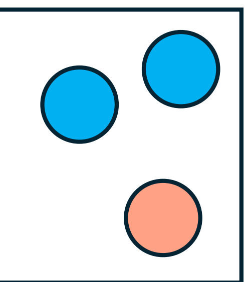
- Some objects copied by the LRB may become cold soon
- These objects may unnecessarily increase the effective working set



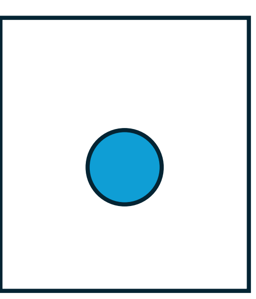
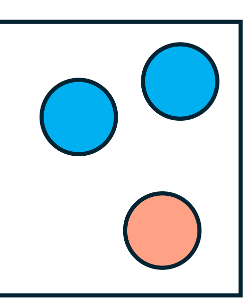
Transient Time

- Transient Time (TT) models the period a memory block stays in cache after access
  - Measured in the number of total accesses
- Memory block containing only objects that are not accessed will be evicted from cache after TT
- Hot page is likely to contain hot objects
  - objects are likely to stay in cache

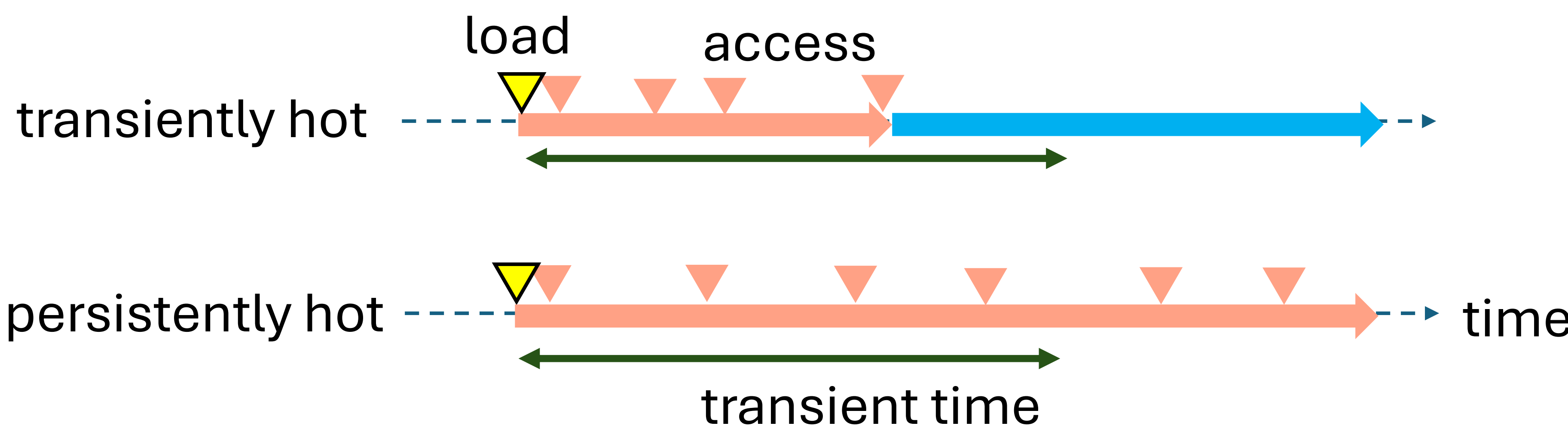




Examining HCSGC

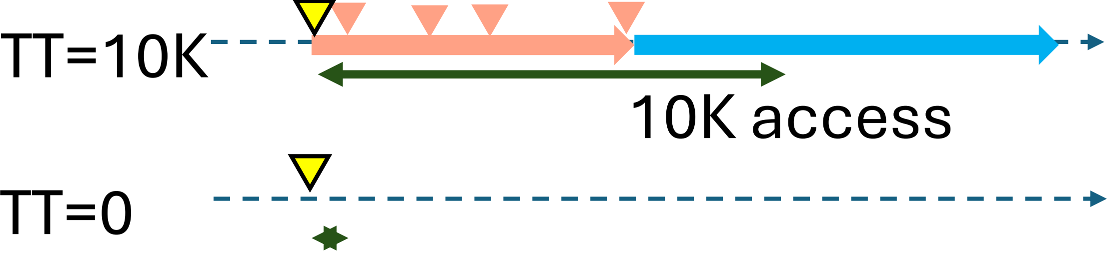



- Counted the number of objects that were not accessed again after TT
  - transiently hot



Finding

counted as “wasted” when

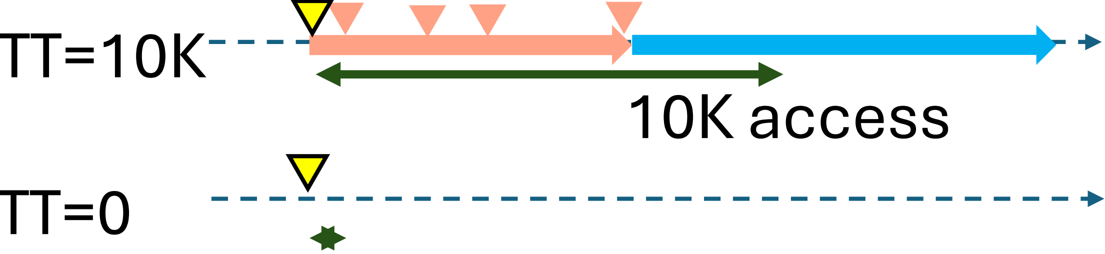


- Many objects were not accessed after TT
- Some objects were never accessed after ref loads

Ratio of **transiently hot**

counted as “wasted” when

Finding



- Many objects were not accessed after TT
- Some objects were never accessed after ref loads

Ratio of **transiently hot**

TT=10K

100%

0%

xalan sunflow pmd lusearch-fix lusearch luindex jython h2 fop avrora

lower is better

TT=0

100%

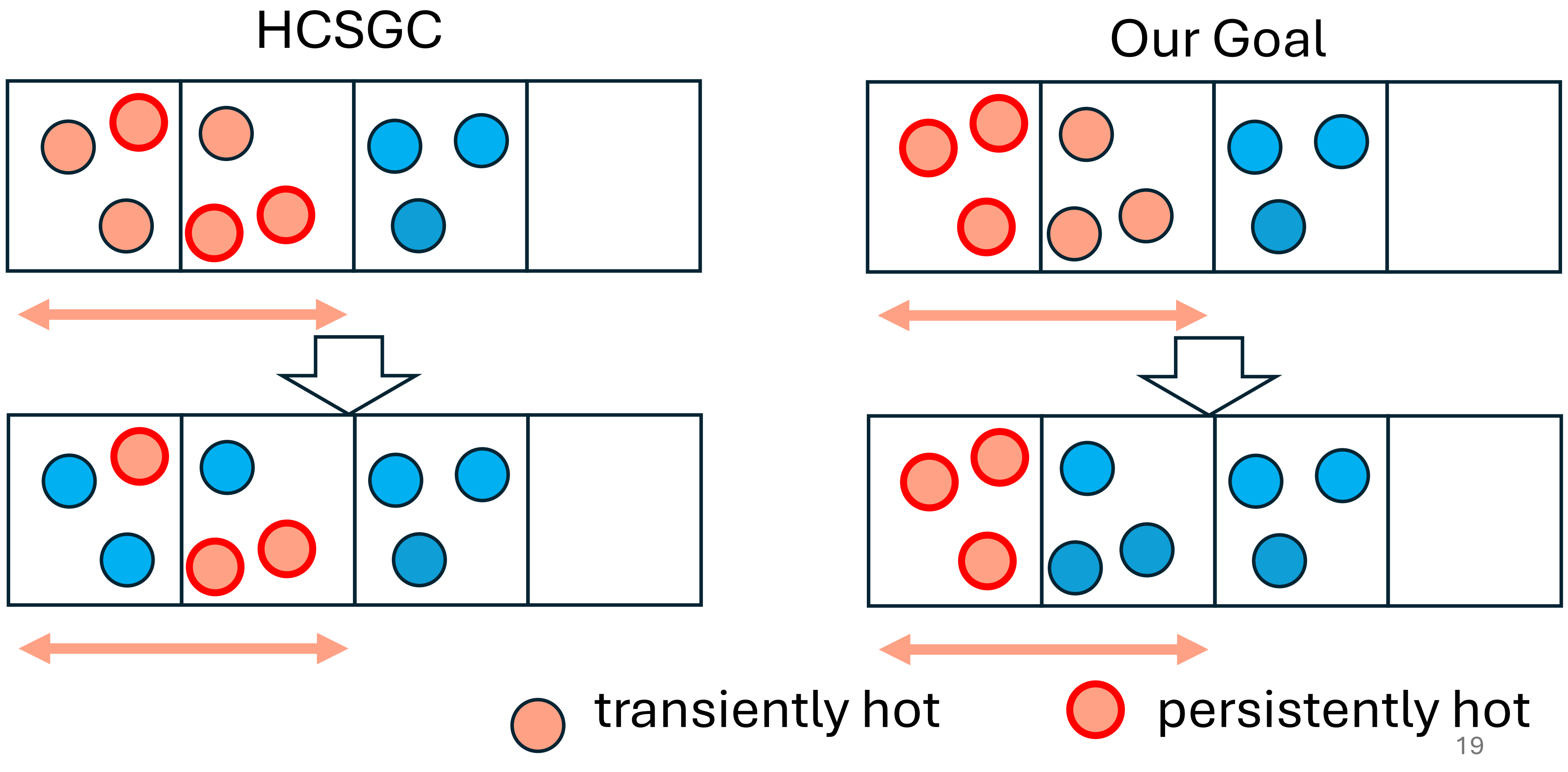
0%

```
if (x.a == null)
```

xalan sunflow pmd lusearch-fix lusearch luindex jython h2 fop avrora

Goal

Segregate transiently hot objects from persistently hot



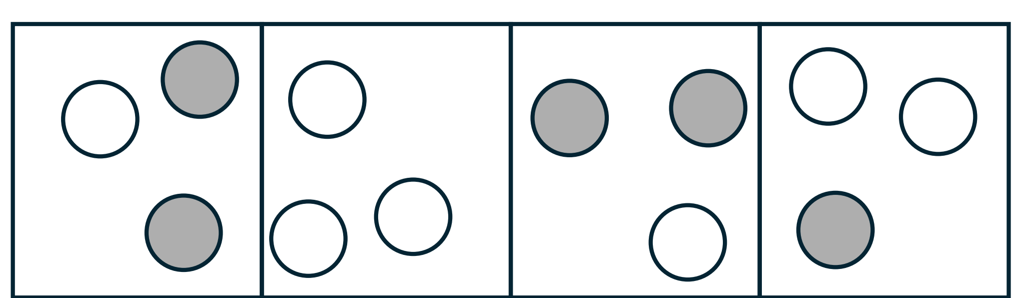
Our Challenge

- How to identify persistently hot objects?
  - Accurate prediction of future object accesses
  - Low overhead

persistently hot

hot

cold



Proposal: Load-site-based filtering (LSBF)

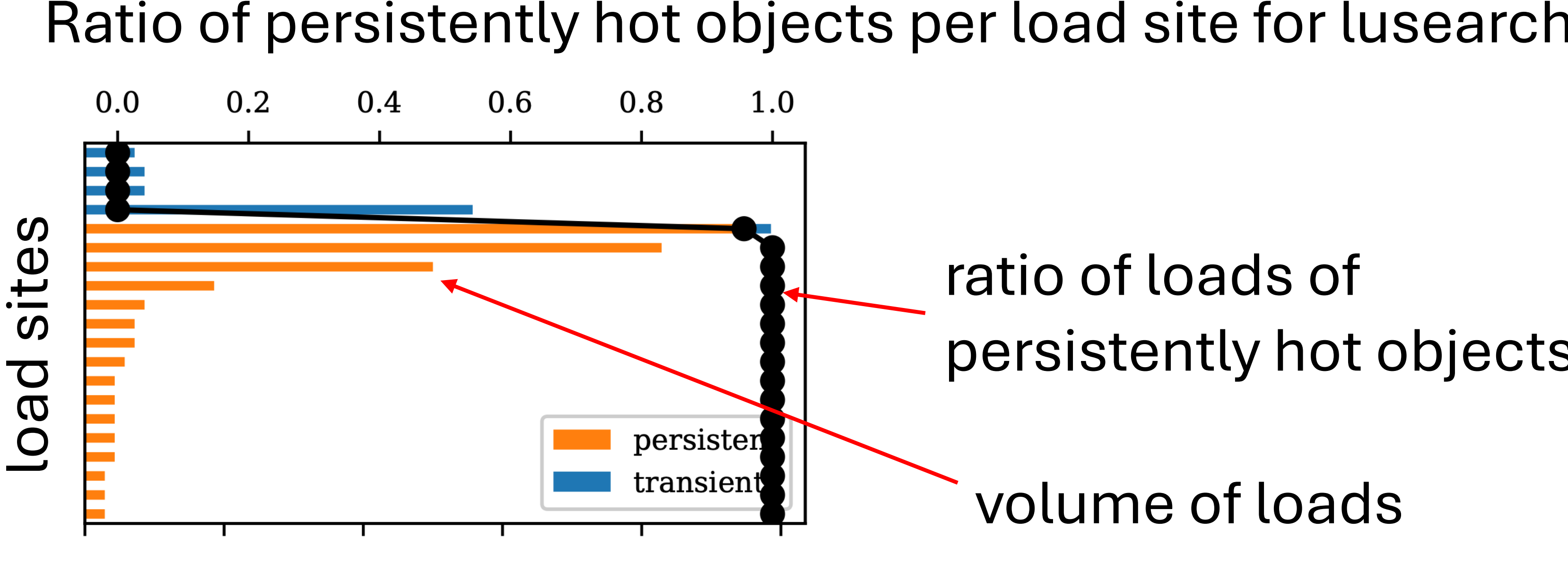
Use load-sites to predict durability of hotness

Filter transiently hot objects

- Inherit the merit of HCSGC:
  - Predict durability of hotness of object when it is moved
    - no per-object housekeeping metadata (e.g., profiling counter)
- Assumption: load-sites have correlation with durability of hotness

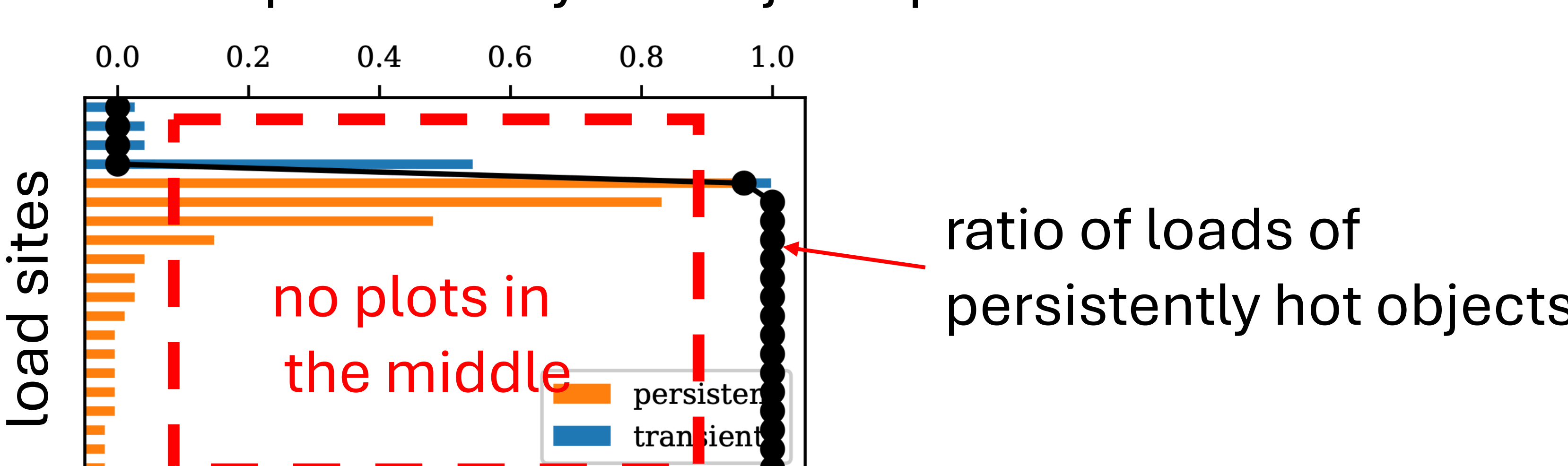
Examining Our Assumption

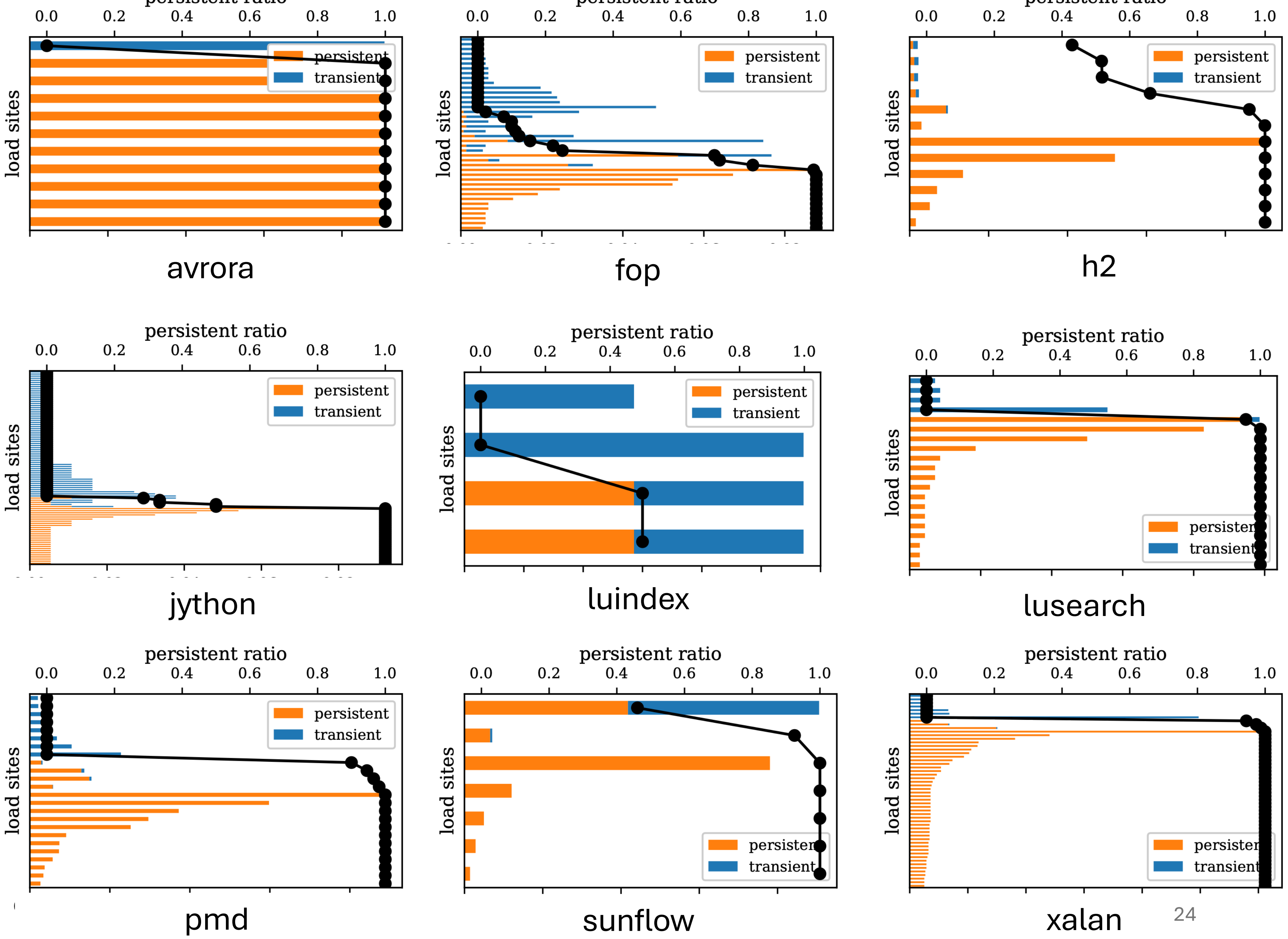
- Load sites distinguish durability of hotness



Examine Our Assumption

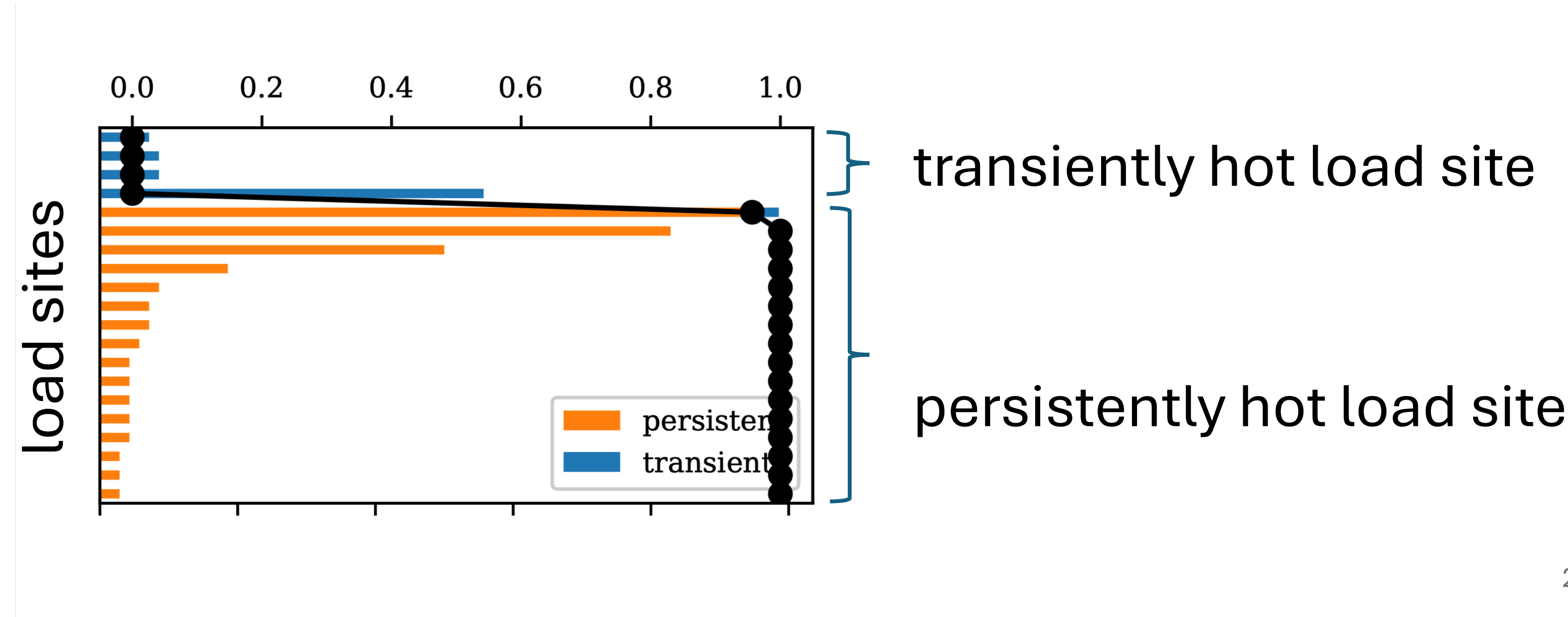
- Load sites distinguish durability of hotness




persistent ratio
load sites
persistent
transient
0.0
0.2
0.4
0.6
0.8
1.0
avrora
fop
h2
jython
luindex
lusearch
pmd
sunflow
xalan

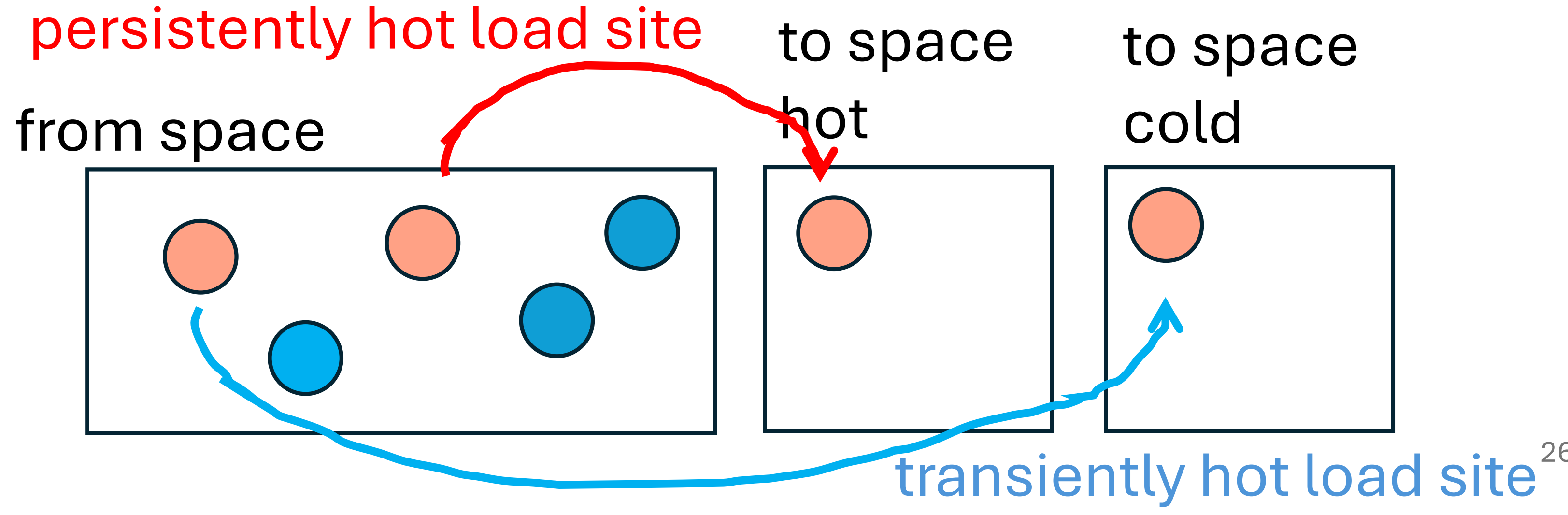
Classifying Load Sites

- Transiently hot load sites:
  99 % of loads refer to transiently hot objects
- Persistently hot load sites: others



Implementation

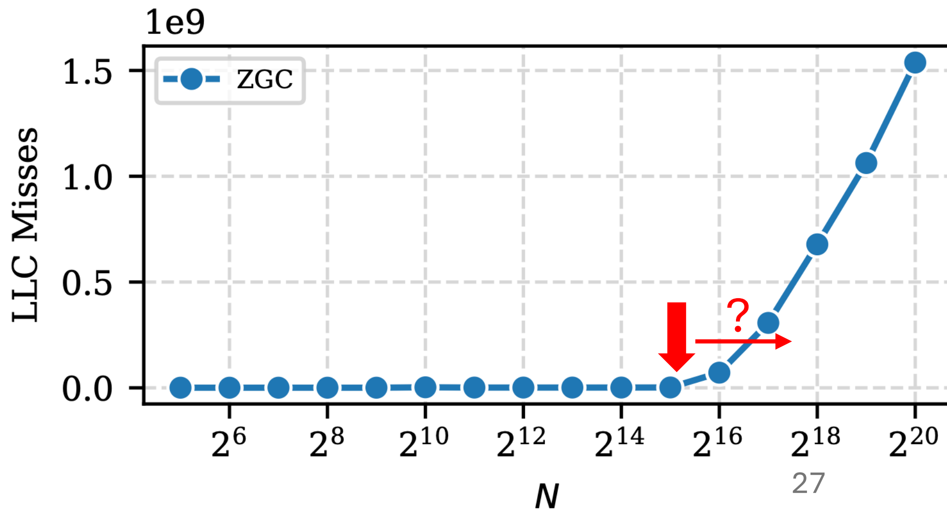
- JIT compiler compiles load sites differently
  - Persistently hot load sites copy objects to hot page
  - Transiently hot load sites copy objects to cold page
- Classify load sites by offline profiling



Evaluation

- Does LSBF reduce effective working set?
- Does LSBF improve program throughput?

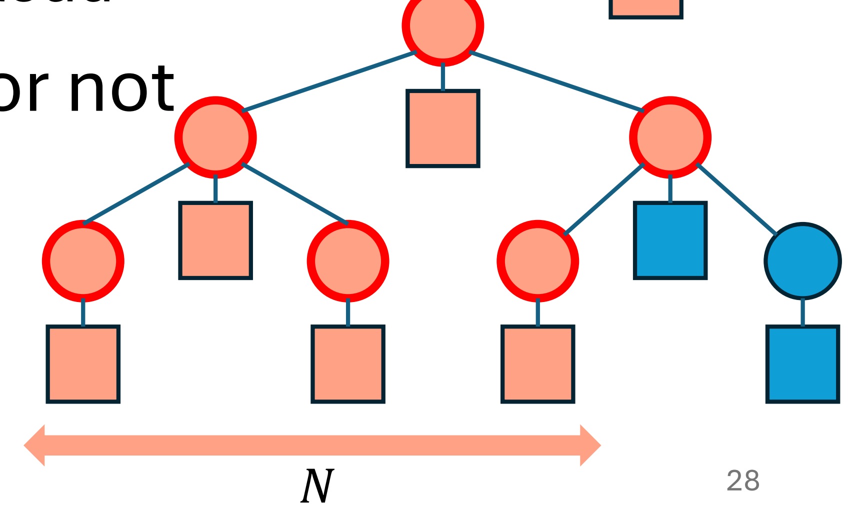
- Implemented LSBF in OpenJDK 14
- Measure cache misses for different workload sizes



Workload: hsckiller

- Interleaves loads of ref to persistently and transiently hot objects

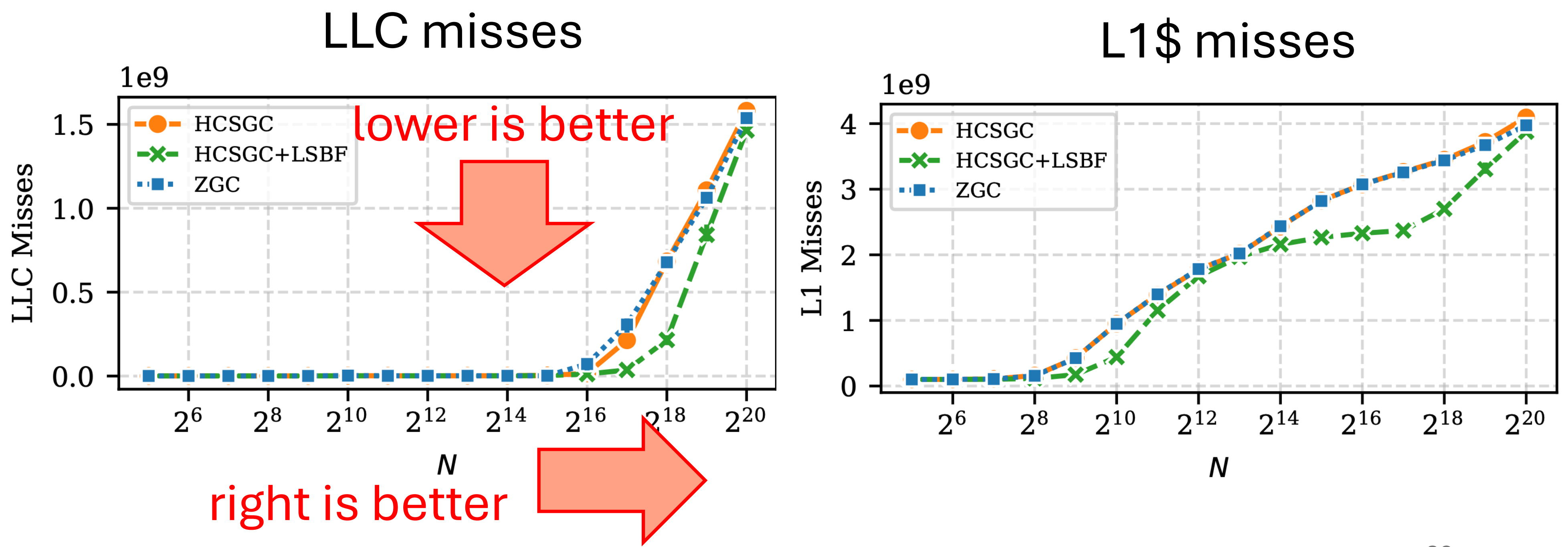
- Large binary tree
- Searches for keys in a specific range: $[0, N]$
  - N controls the size of workload
- Checks if `value` is `null` or not
  - Does not access values
  - Values are transiently hot



LLC: 8.3 MB shared
L1$: 32 KB per core

Cache misses

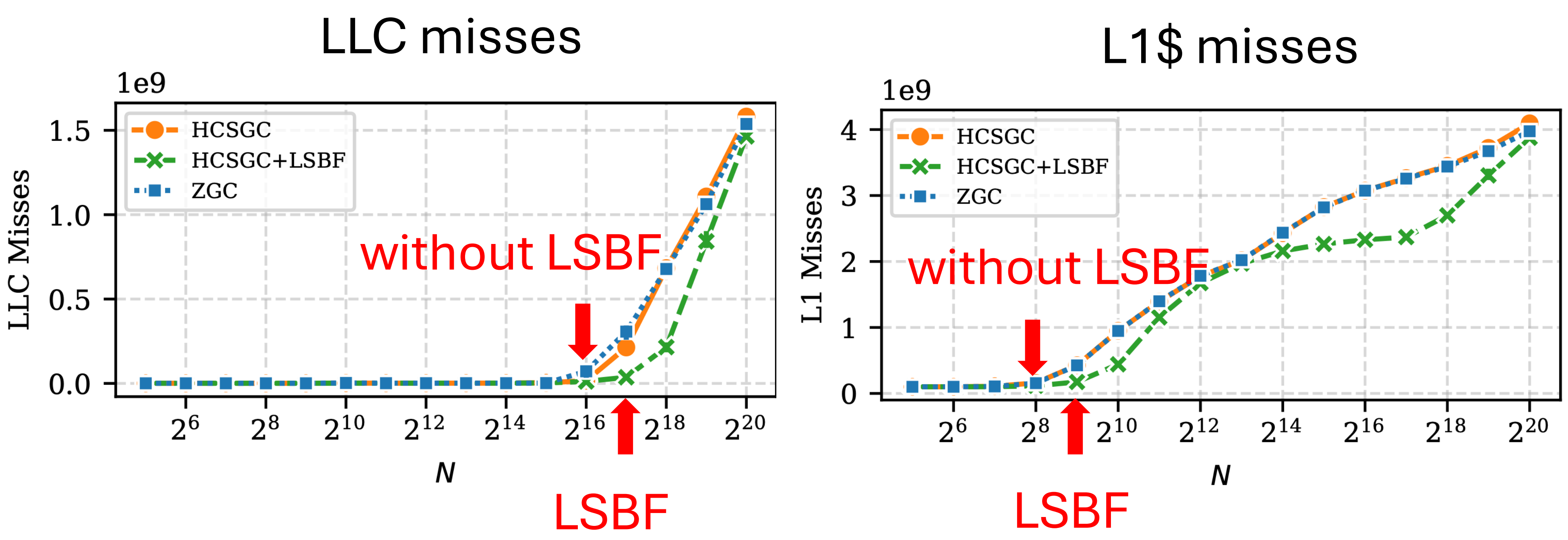
- With LSBF, a larger workload could be contained in cache



LLC: 8.3 MB shared
L1$: 32 KB per core

Cache misses

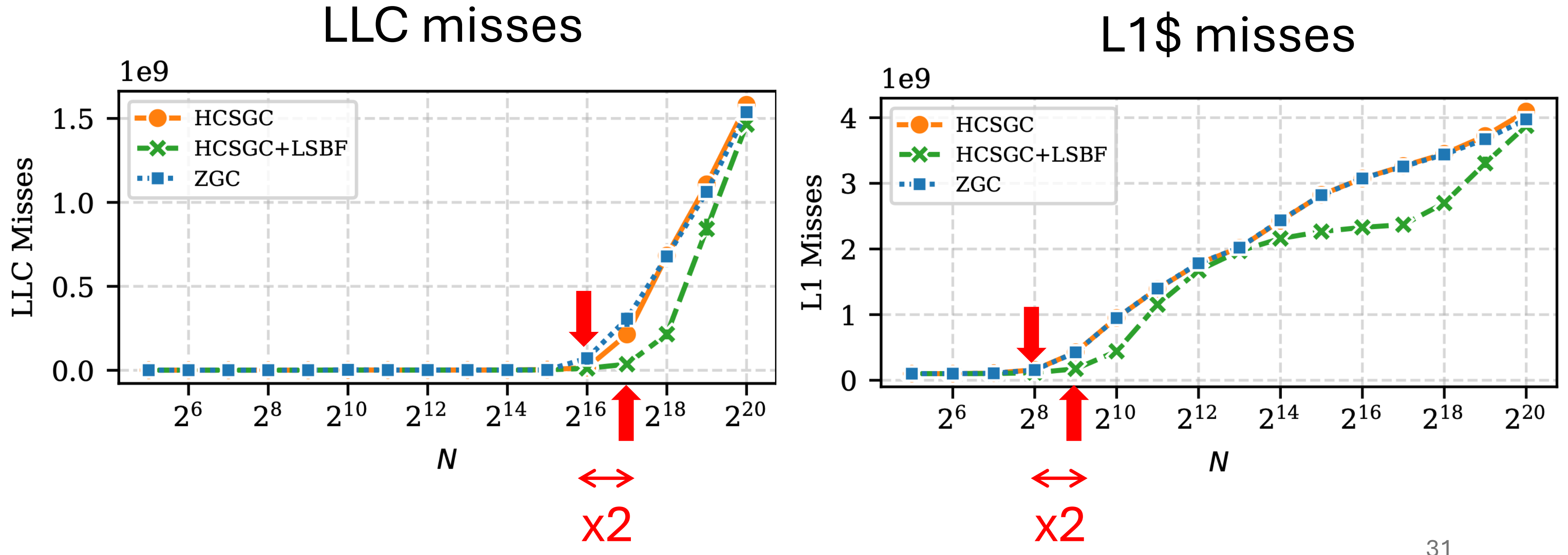
- With LSBF, a larger workload could be contained in cache



LLC: 8.3 MB shared
L1$: 32 KB per core

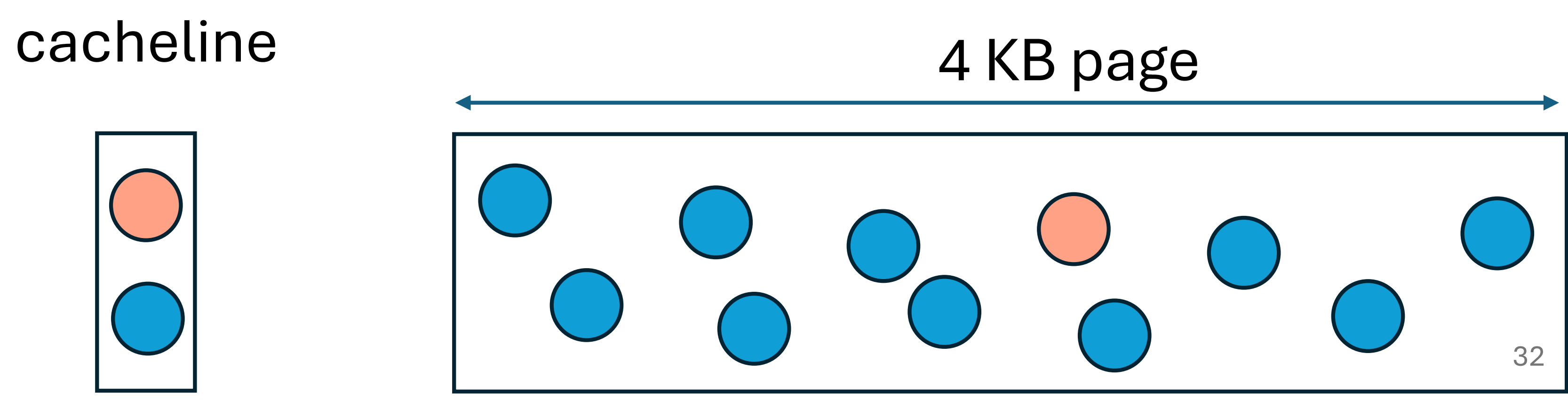
Cache misses

- With LSBF, a larger workload could be contained in cache
    - double size of workload could be contained



TLB

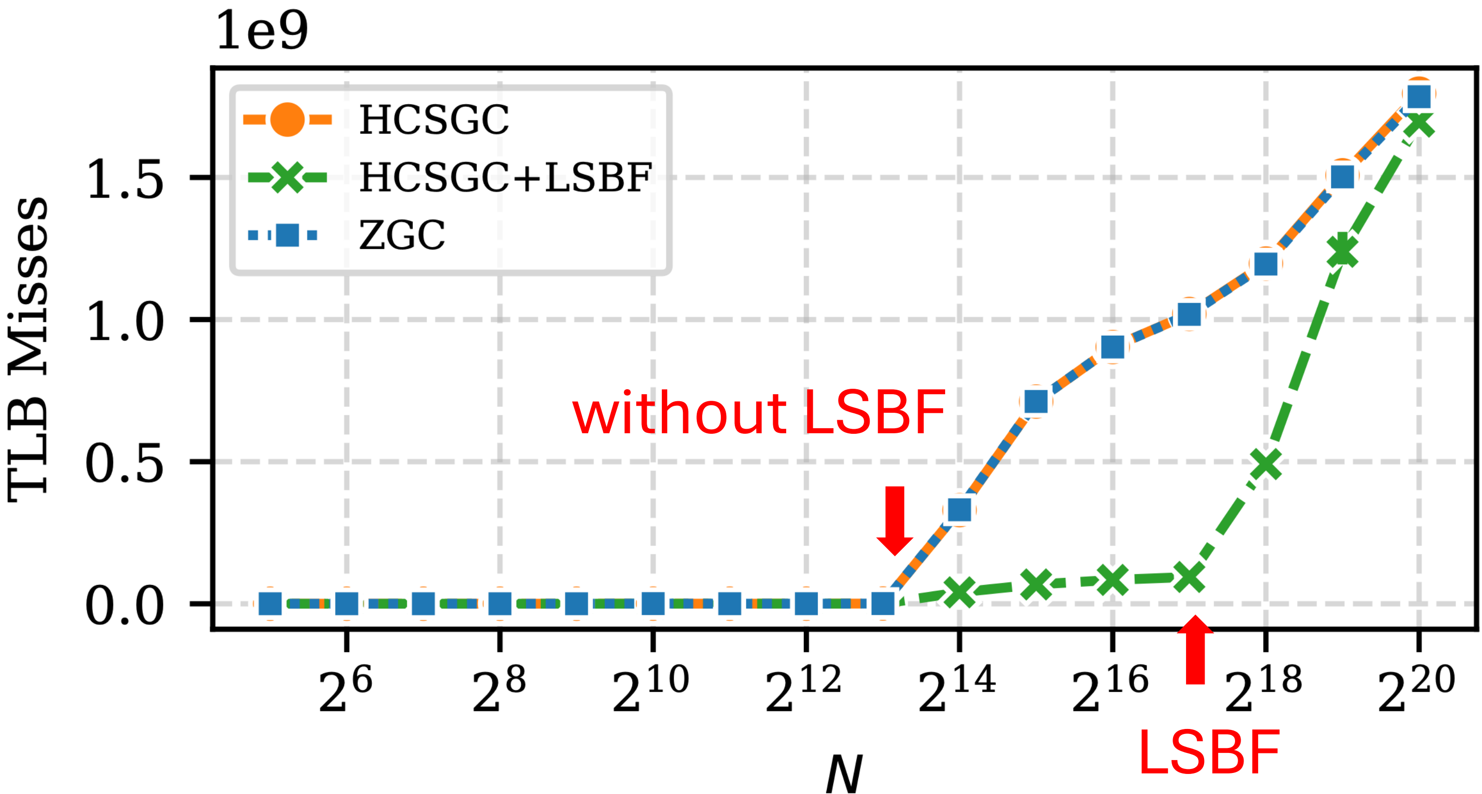
- Cache of virtual-physical address translations
  - Address translation accesses page table in memory
  - TLB caches the translation results
- A TLB entry covers more objects than a cache line
  - One cacheline covers 64 bytes
  - One TLB entry covers a 4-KB page



TLB miss

1536 TLB entries
covering ~ 6 MB

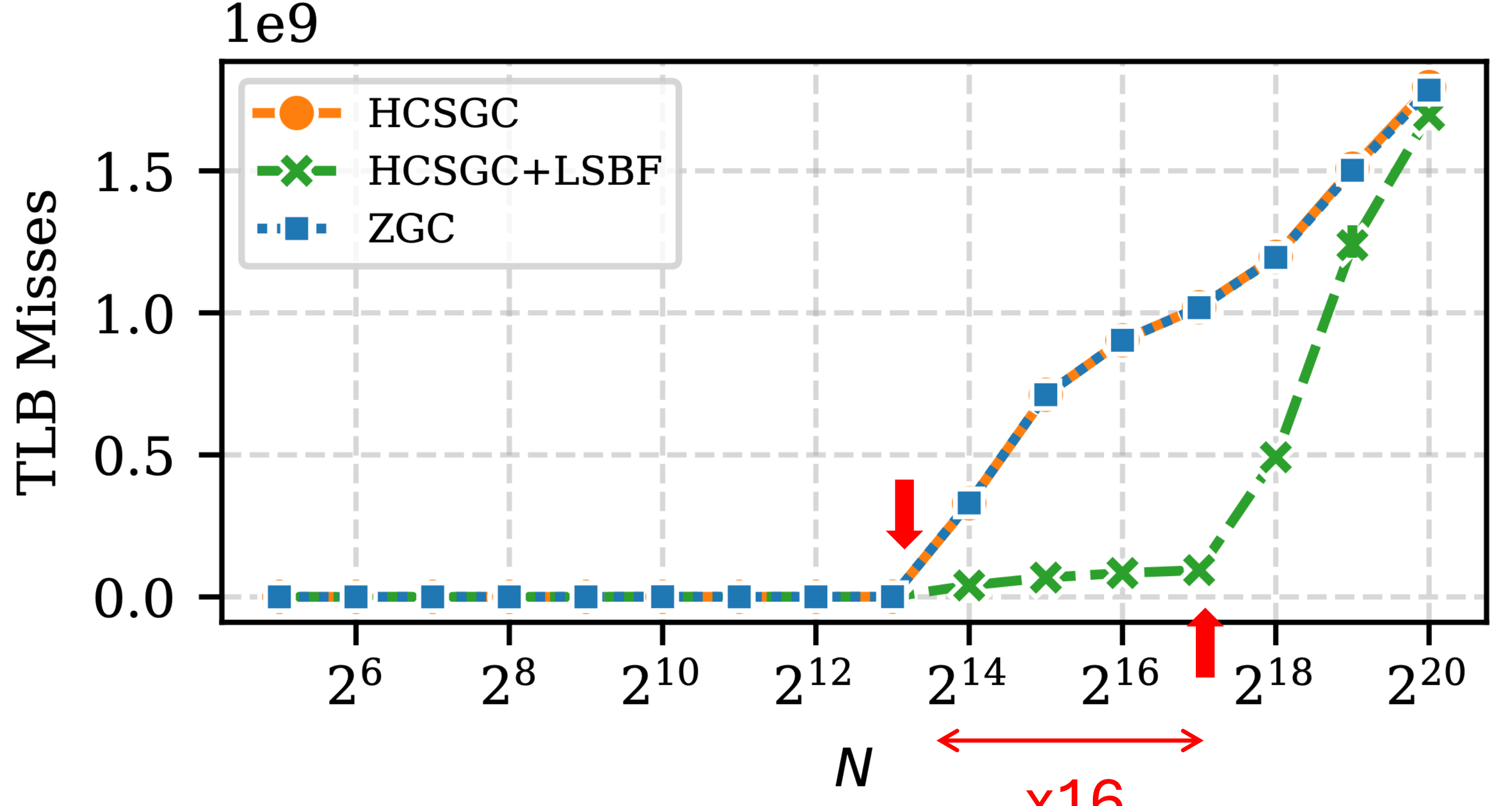
- Similar tendency to cache misses



TLB miss

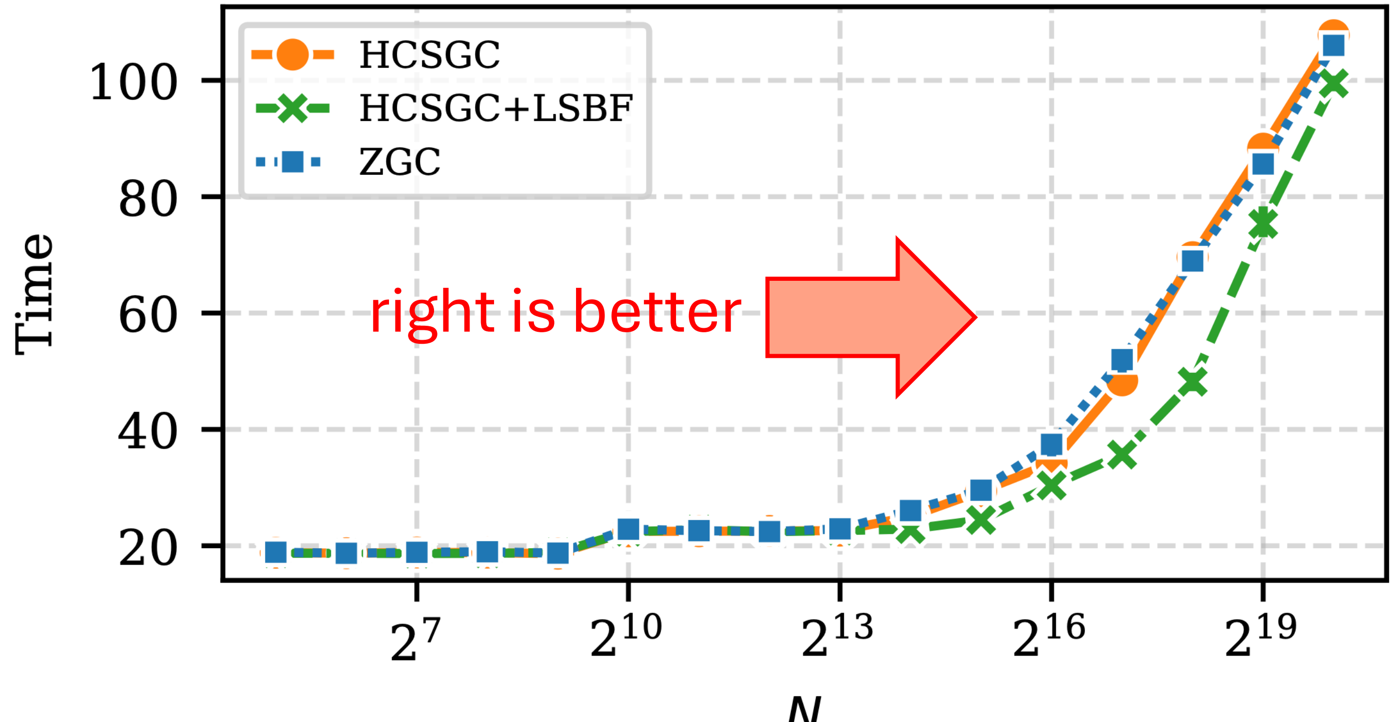
1536 TLB entries
covering ~ 6 MB

- Similar tendency to cache misses
- Difference is larger



Execution time

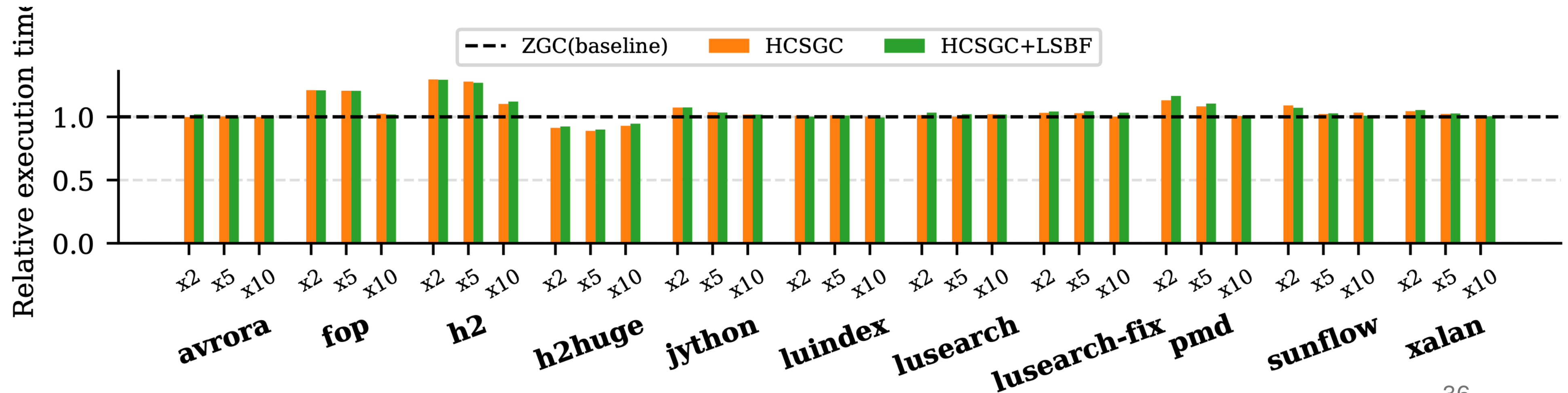
- Execution time reflects the cache misses
- With LSBF, the curve is shifted to the right



DaCapo (execution time)

No improvements

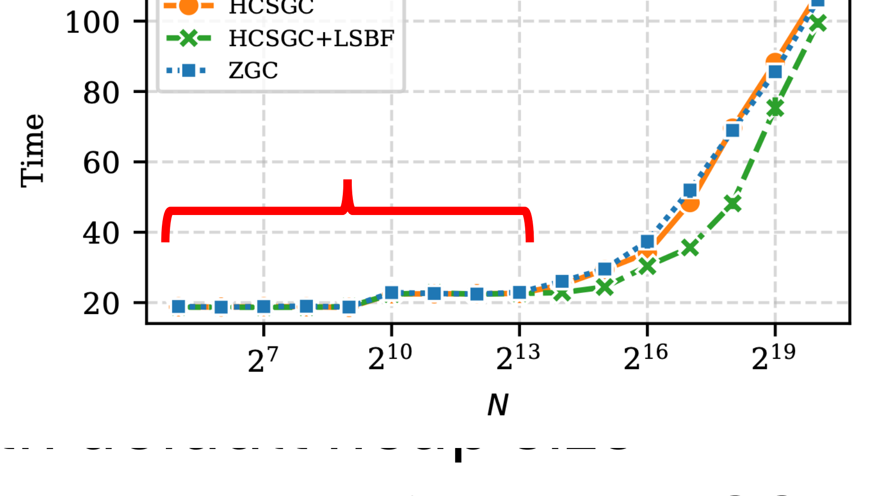
- Workload was too small
- GC occurred frequently with default heap size
  - Objects that became cold are segregated in the next GC

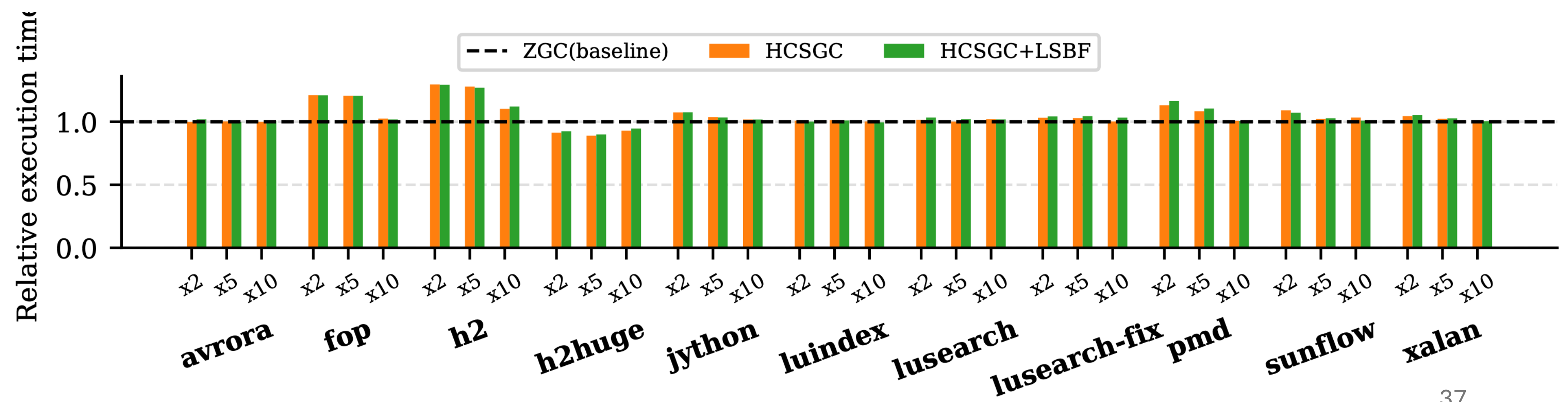


DaCapo (executior

No improvements

- Workload was too small
- GC occurred frequently wi
  - Objects that became cold are segregated in the next GC

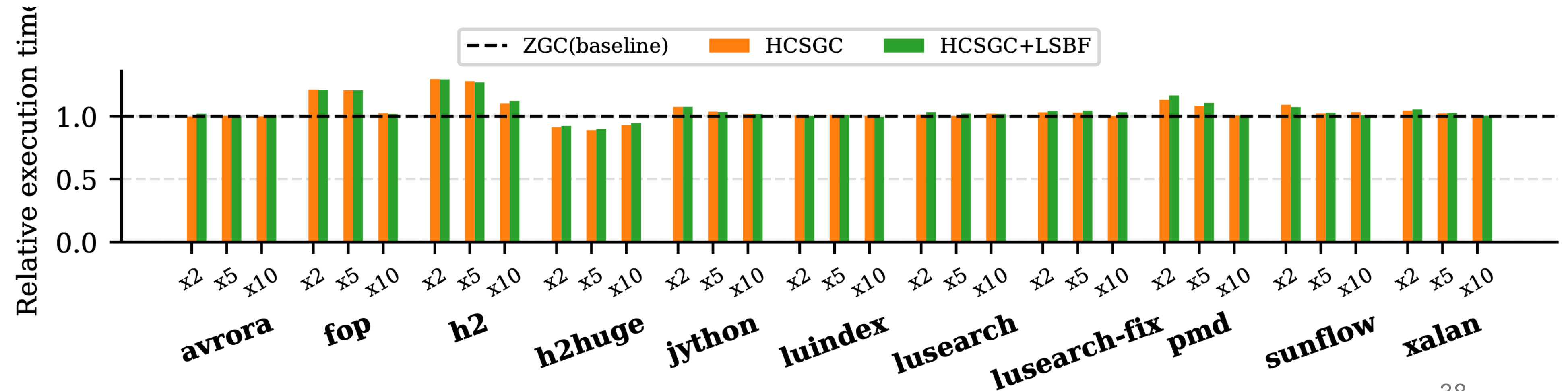




DaCapo (execution time)

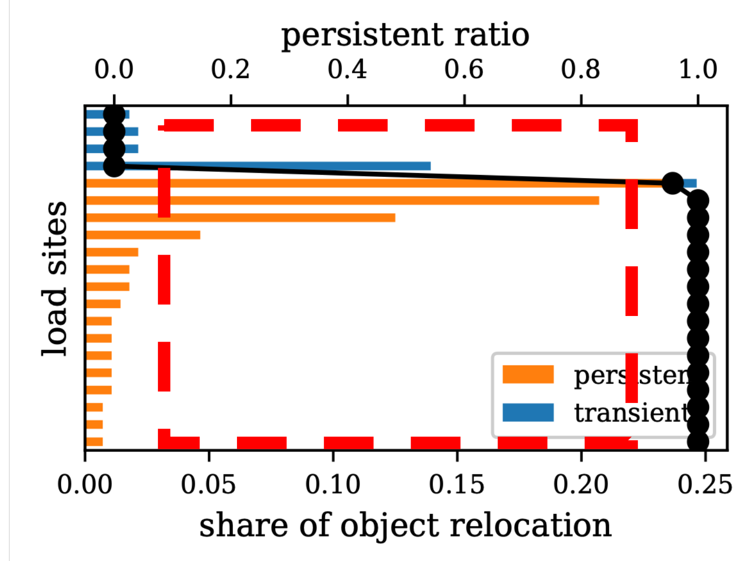
No improvements

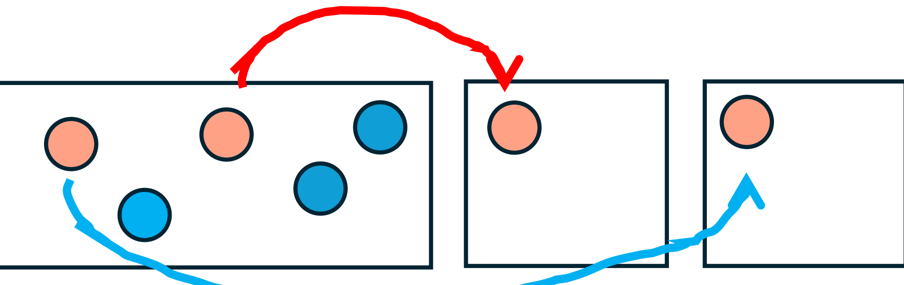
- Workload was too small
- GC occurred frequently with default heap size
  - Objects that became cold are segregated in the next GC



Conclusion

- Objects whose refs are loaded are not necessarily accessed frequently
- Load site distinguishes transiently hot objects
- We proposed LSBF
- There is a program where LSBF can reduce cache misses





TLB miss