FFCCD: Fence-Free Crash-Consistent Concurrent Defragmentation for Persistent Memory

Yuanchao Xu, Chencheng Ye, Yan Solihin, Xipeng Shen

NC STATE UNIVERSITY

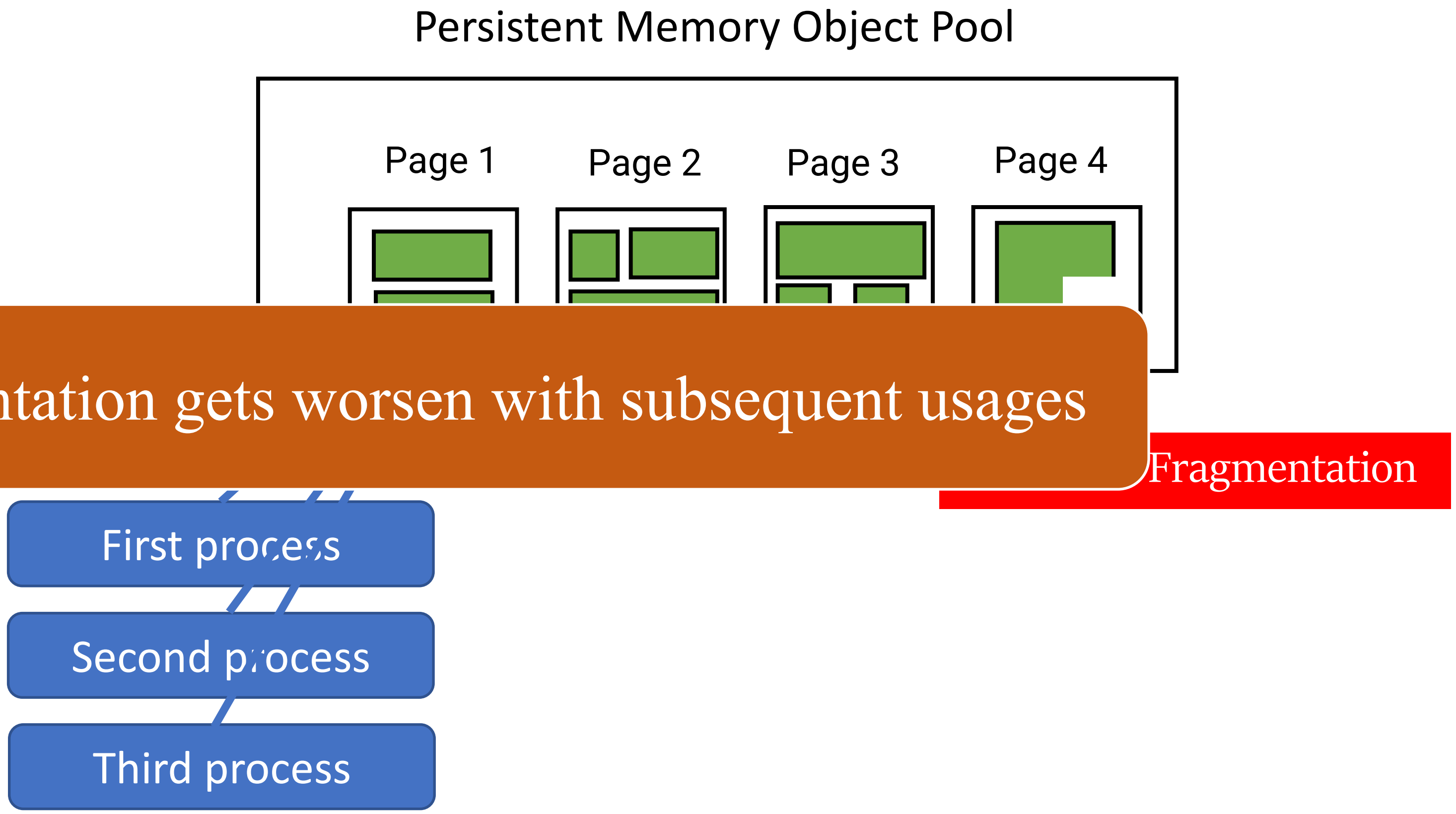
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

UNIVERSITY OF CENTRAL FLORIDA

Persistent Fragmentation

Intel Persistent M

- Higher Density
- Byte-addressable Persistence
- DRAM-like Performance

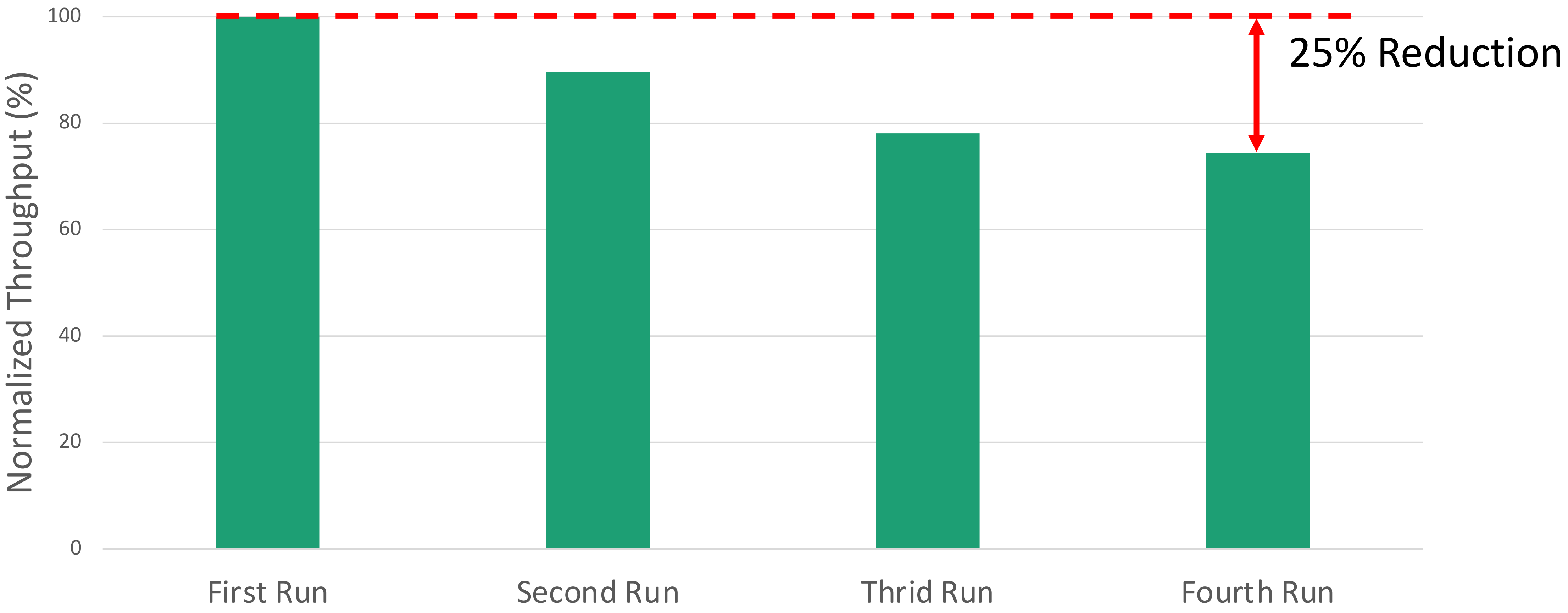


Persistent fragmentation gets worsen with subsequent usages

🟩 Live object

Performance Degradation from Persistent Fragmentation

Redis throughput for 2M insertions and 1M deletions

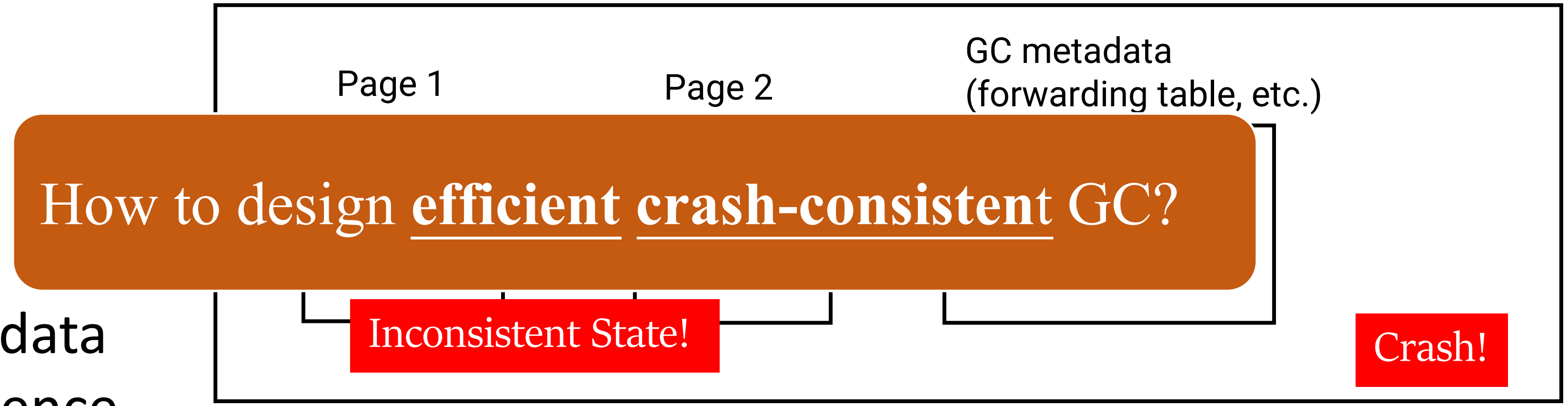


Garbage Collection (GC)

Garbage Collection Steps

1) Marking
2) Summary
3) Compaction
    - Relocation
    - Update metadata
    - Update Reference

Persistent Memory Object Pool



How to design **efficient** **crash-consistent** GC?

🟩 Live object → Pointer

Contributions

- First to analyze PM fragmentation systematically
  - Identify sfences as the key performance bottleneck
  - Analyze post-crash states to explore efficiency opportunities

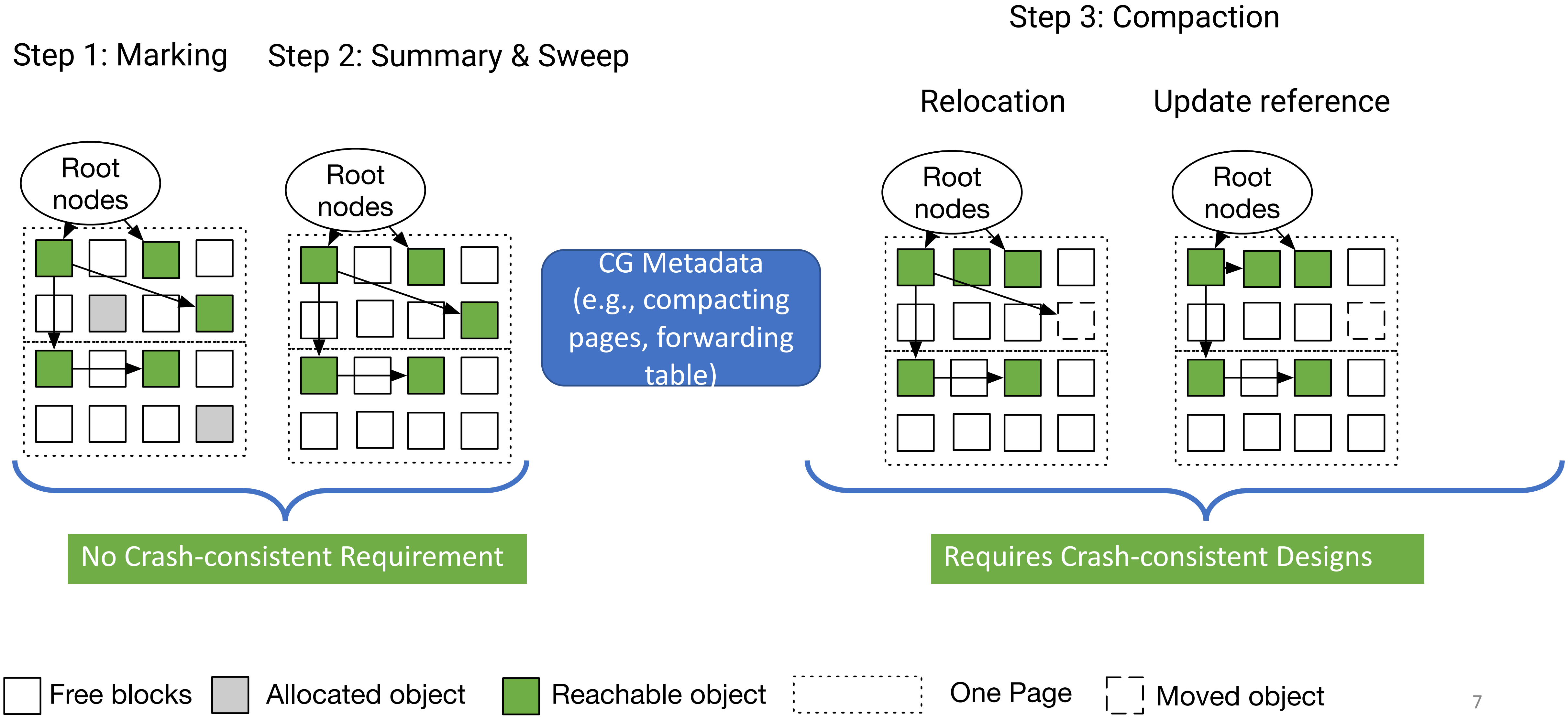
- Design multiple concurrent GC solutions
  - Pure software single-fence crash-consistent design (SFCCD)
  - Architectural support for fence-free crash-consistent design (FFCCD)

- We evaluate designs and show its effectiveness

Agenda

- Motivation
- Background
- Post-crash state exploration
- Architecture design
- Evaluation

Concurrent Garbage Collection



Step 1: Marking

Step 2: Summary & Sweep

Step 3: Compaction

Relocation

Update reference

CG Metadata (e.g., compacting pages, forwarding table)

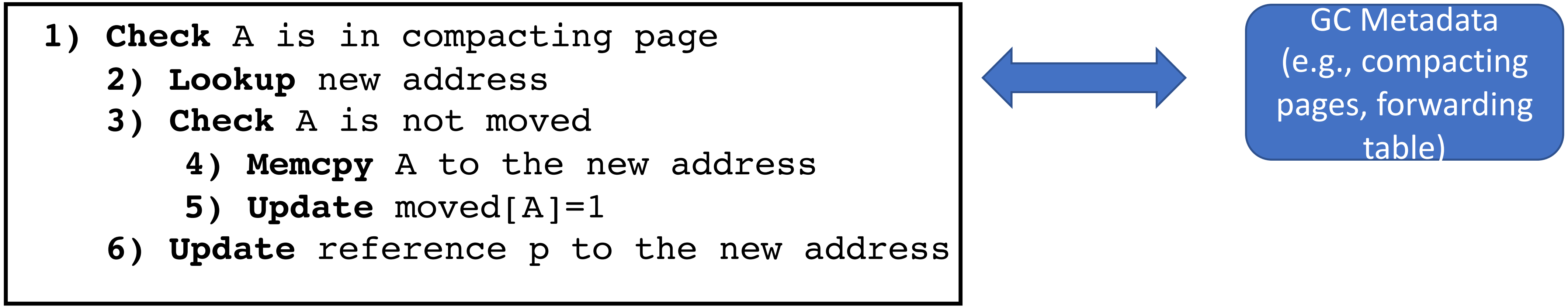
No Crash-consistent Requirement

Requires Crash-consistent Designs

☐ Free blocks ▨ Allocated object 🟩 Reachable object ⬚ One Page ⬚ Moved object

Concurrent Compaction: Read Barrier

```
p->first = input  //p points to object A
<read barrier>
```



```
1) Check A is in compacting page
    2) Lookup new address
    3) Check A is not moved
        4) Memcpy A to the new address
        5) Update moved[A]=1
    6) Update reference p to the new address
```

GC Metadata (e.g., compacting pages, forwarding table)

Crash-consistent Concurrent Compaction Baseline[1]

```
p->first = input  //p points to object A
<read barrier>
 1) Check A is in compacting page
    2) Lookup new address
    3) Check A is not moved
       4) Memcpy A to the new address
       sfence()
       5) Update moved[A]=1
       clwb(moved[A]);sfence()
    6) Update reference p to the new address
```

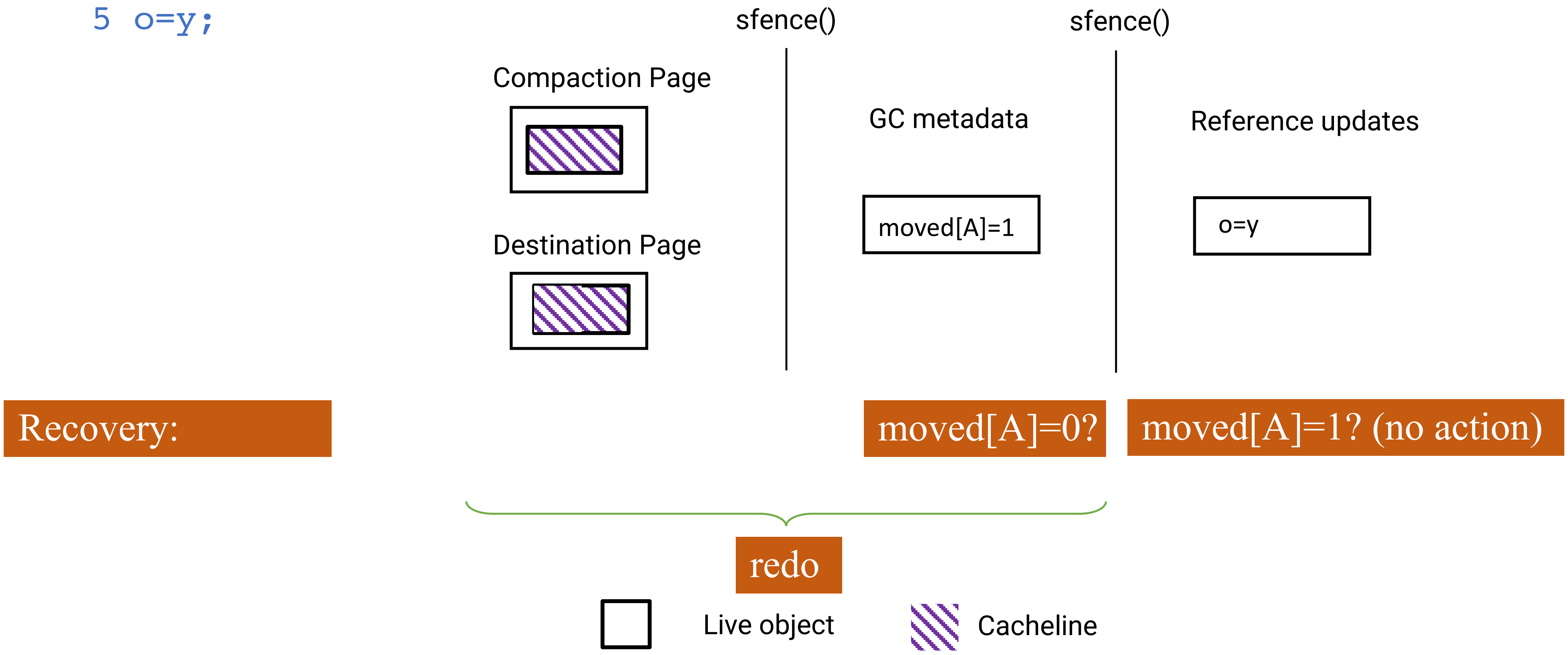
Persist GC metadata before compaction
(e.g., compacting pages, forwarding table)

Crash-consistency relocation incurs about 50% overhead!

[1] Mingyu Wu, et al. Espresso: brewing java for more non-volatility with non- volatile memory. ASPLOS 2018

Baseline Design

```
memcpy_nodrain(y,x,sizeof(A))
sfence();
moved[A]=1;
clwb(moved[A]);sfence();
o=y;
```



sfence()

sfence()

Compaction Page

Destination Page

GC metadata

moved[A]=1

Reference updates

o=y

Recovery:

moved[A]=0?

moved[A]=1? (no action)

redo

☐ Live object

▨ Cacheline

Can we remove the first sfence?

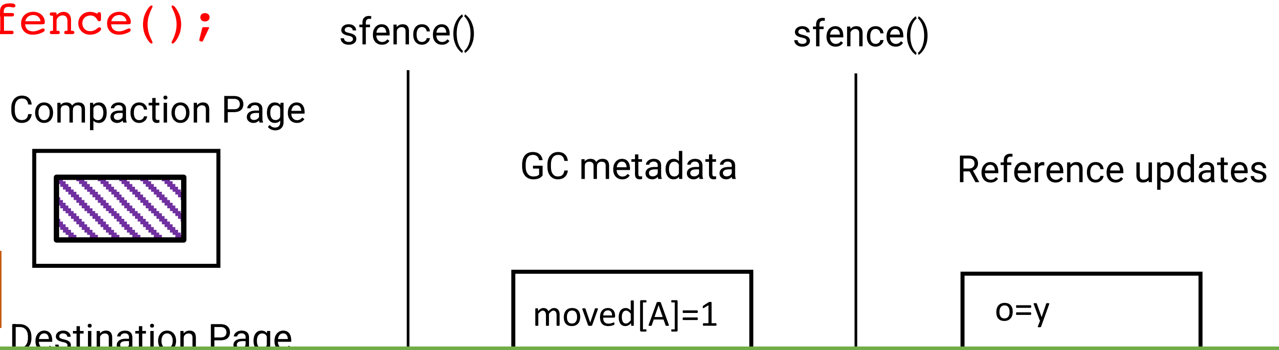
`memcpy_nodrain(y,x,sizeof(A))`

~~`sfence();`~~

`moved[A]=1;`

`clwb(moved[A]);sfence();`

`o=y;`



Post-crash states

Same?

Single-fence solution

Recovery: compare values with memcpy source for objects without reference update

Recovery:

Comparision Results	Recovery steps
Same	set moved[A]=1
Paritally Same	memcpy others & set moved[A]=1
Differ	unset moved[A]=1

oved[A]=1? (no action)

eference is updated? (no action)

☐ Live object ▨ Cacheline

Can we remove the second sfence?

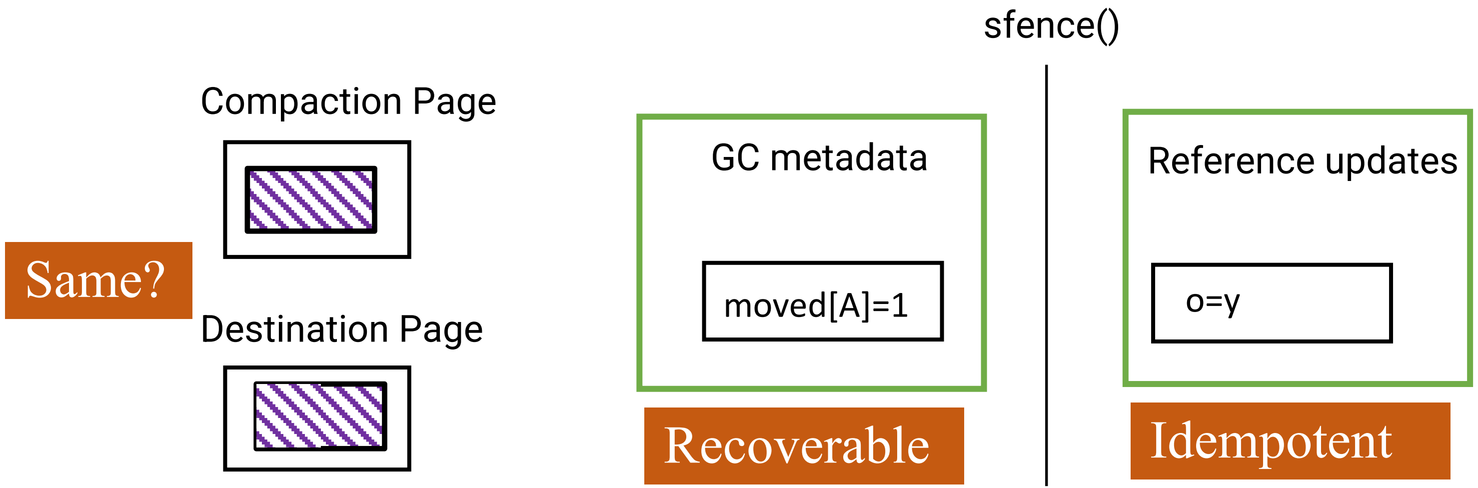
`memcpy_nodrain(y,x,sizeof(A))`

`moved[A]=1;`

~~`clwb(moved[A]);sfence();`~~

`o=y;`

Post-crash states



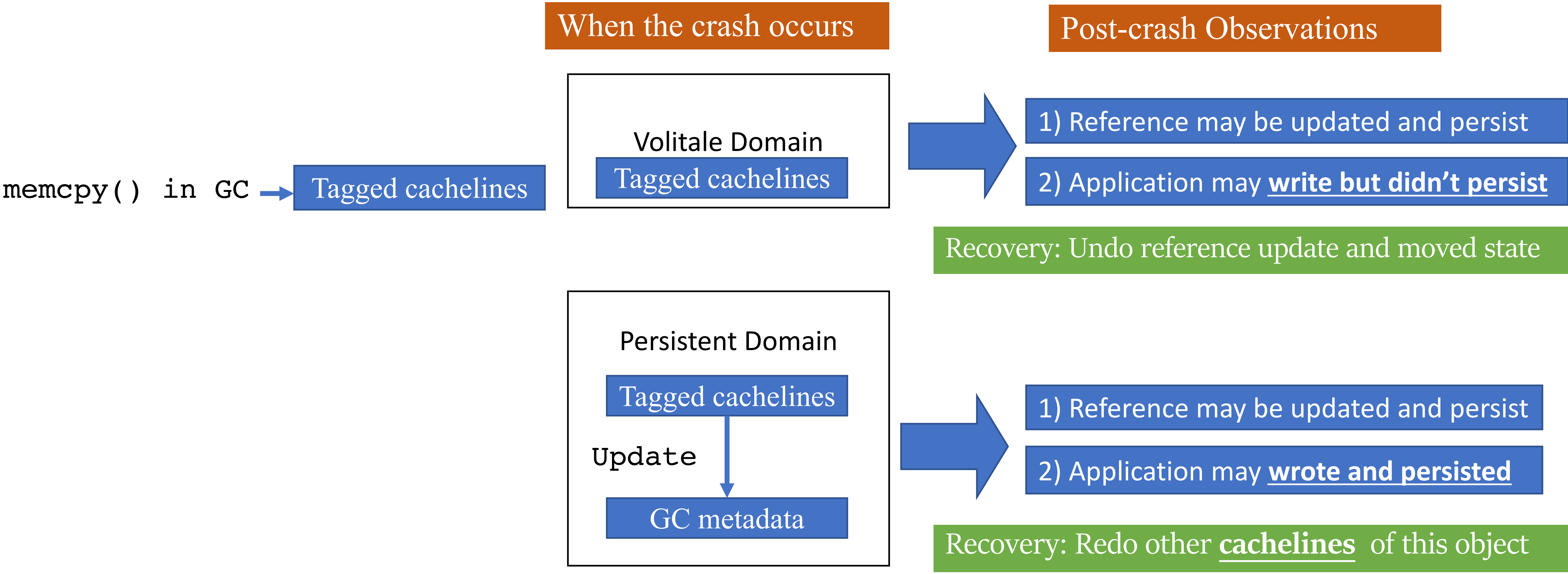
Recovery:

1) **Random**
2) Source
3) **New**

Destination Value for each cacheline	**Recovery steps**
Source	Set moved[A]=1
New	Set moved[A]=1
Random	Redo/undo

☐ Live object ▨ Cacheline

Idea: Track Relocated Cachelines



When the crash occurs

Post-crash Observations

`memcpy() in GC` → Tagged cachelines

Volitale Domain

Tagged cachelines

1) Reference may be updated and persist

2) Application may **write but didn’t persist**

Recovery: Undo reference update and moved state

Persistent Domain

Tagged cachelines

`Update`

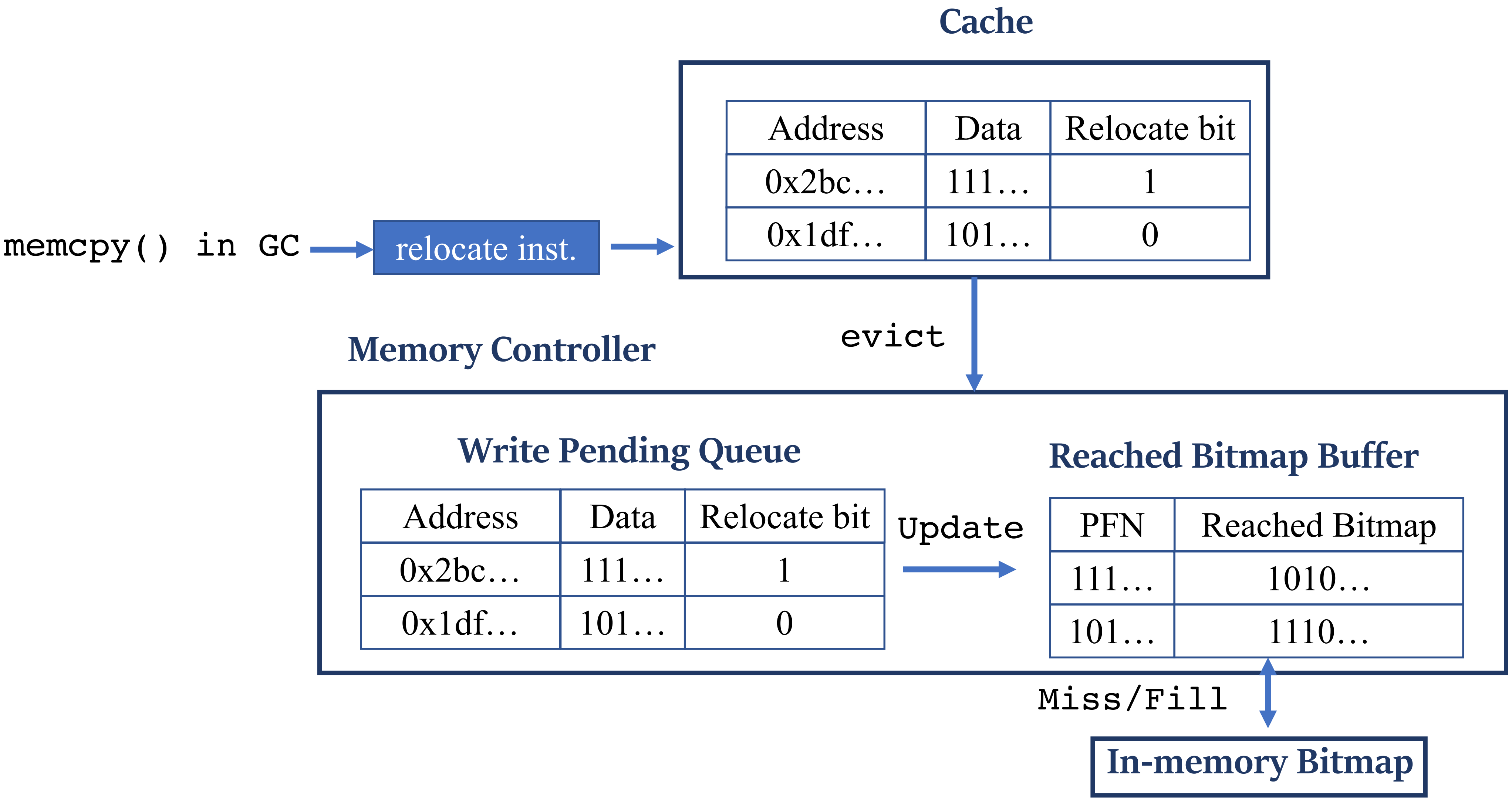
GC metadata

1) Reference may be updated and persist

2) Application may **wrote and persisted**

Recovery: Redo other **cachelines** of this object

Architecture support



Cache

Address	Data	Relocate bit
0x2bc…	111…	1
0x1df…	101…	0

`memcpy() in GC` → relocate inst. →

`evict`

Memory Controller

Write Pending Queue

Address	Data	Relocate bit
0x2bc…	111…	1
0x1df…	101…	0

`Update`

Reached Bitmap Buffer

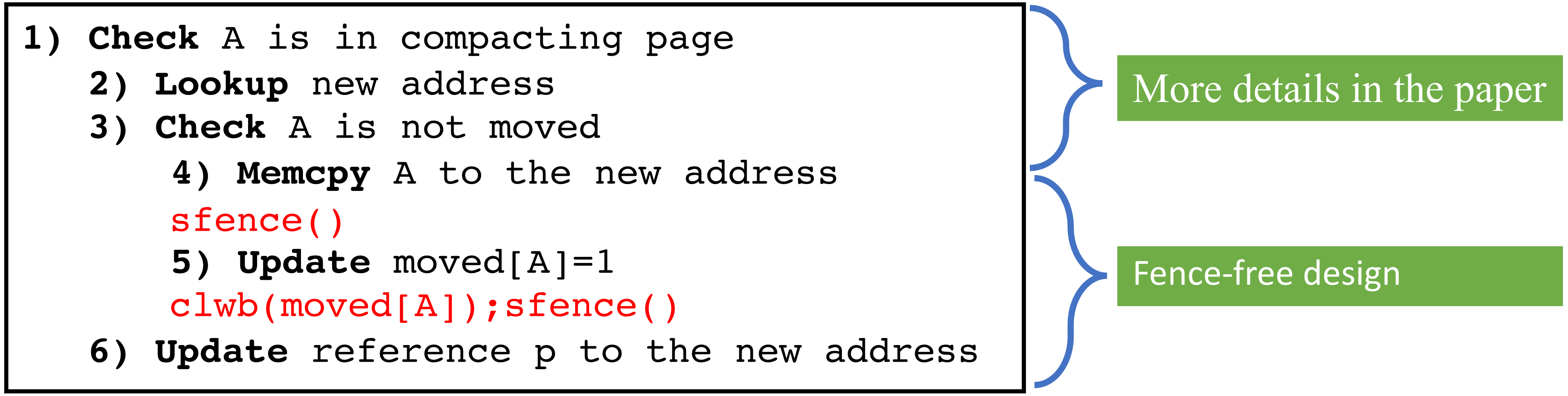
PFN	Reached Bitmap
111…	1010…
101…	1110…

`Miss/Fill`

In-memory Bitmap

Crash-consistent Concurrent Compaction Baseline

```
p->first = input  //p points to object A
<read barrier>
```

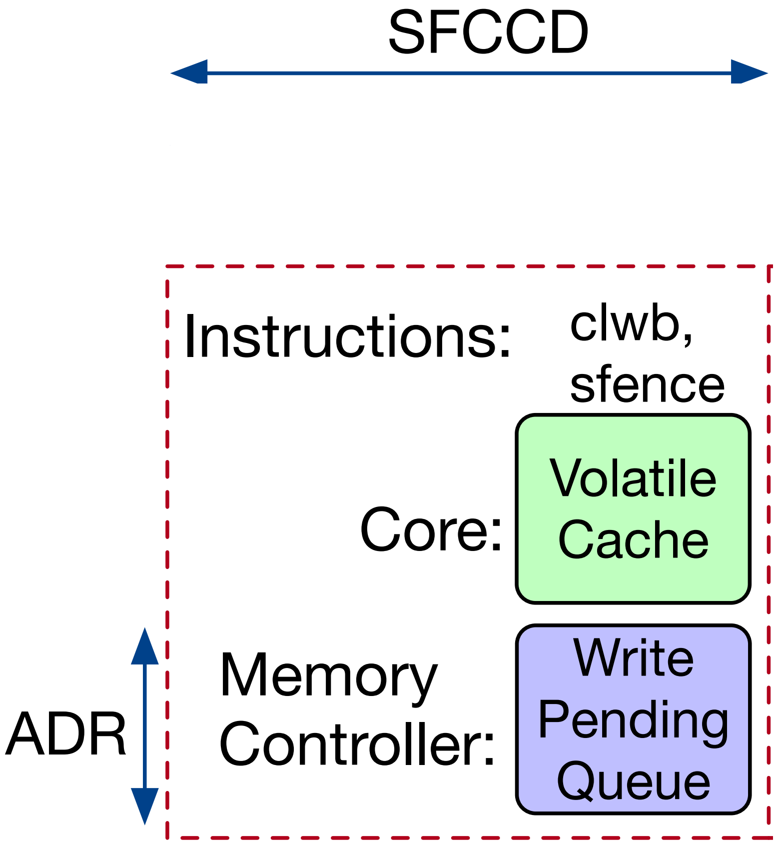


```
1) Check A is in compacting page
   2) Lookup new address
   3) Check A is not moved
      4) Memcpy A to the new address
      sfence()
      5) Update moved[A]=1
      clwb(moved[A]);sfence()
   6) Update reference p to the new address
```

More details in the paper

Fence-free design

Design Summary



SFCCD

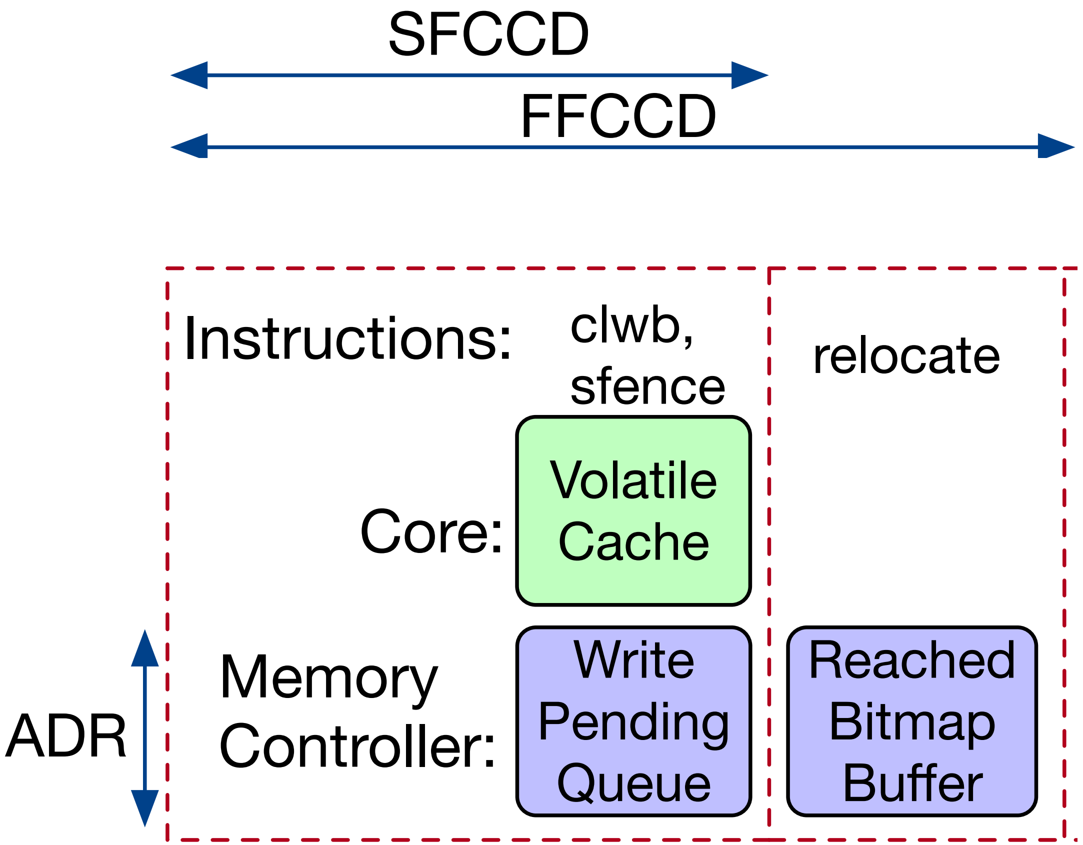
Instructions: clwb, sfence

Core: Volatile Cache

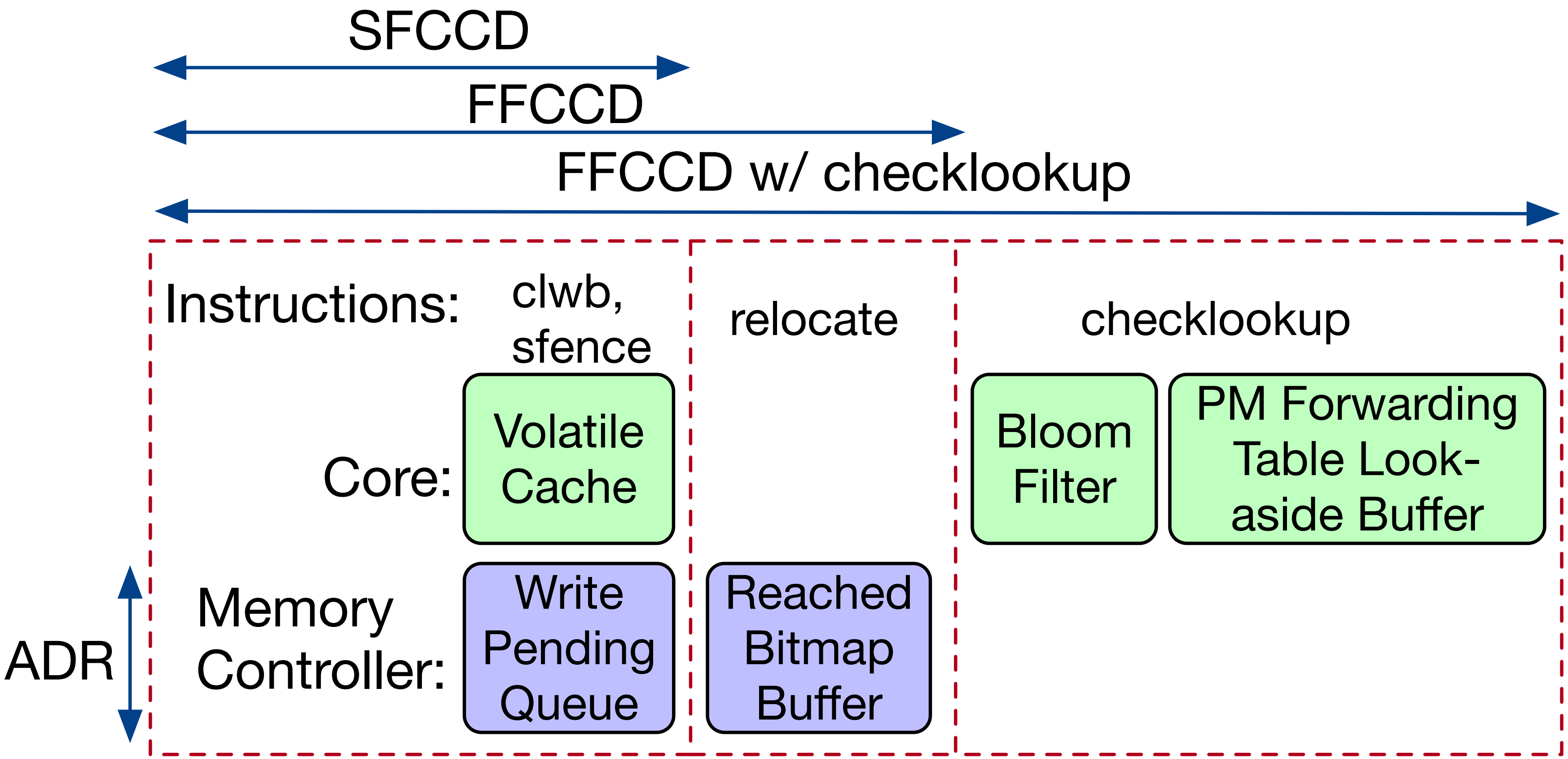
ADR

Memory Controller: Write Pending Queue

Design Summary



Design Summary



Evaluation

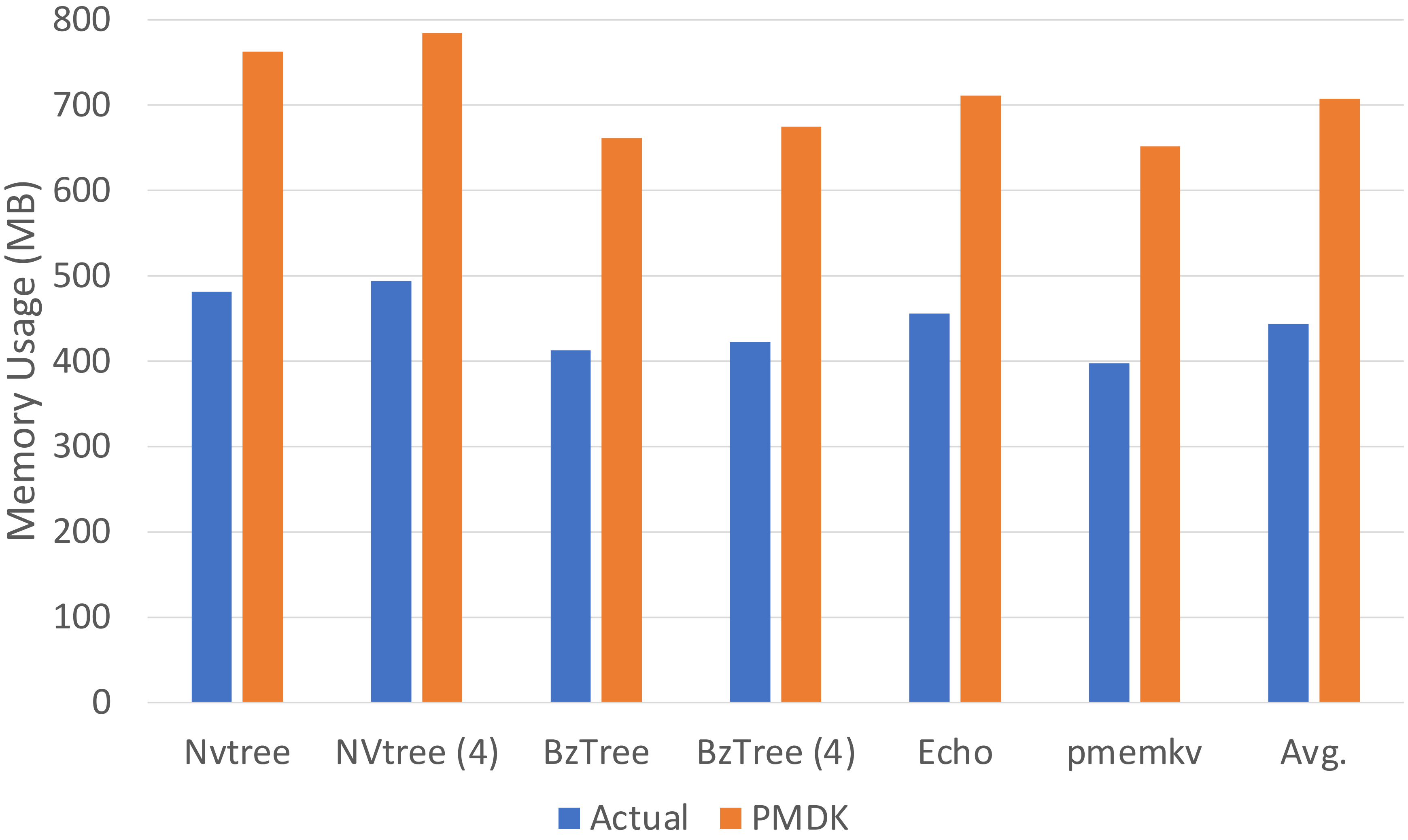
Benchmarks:

- PM data structures
- PM key-value stores

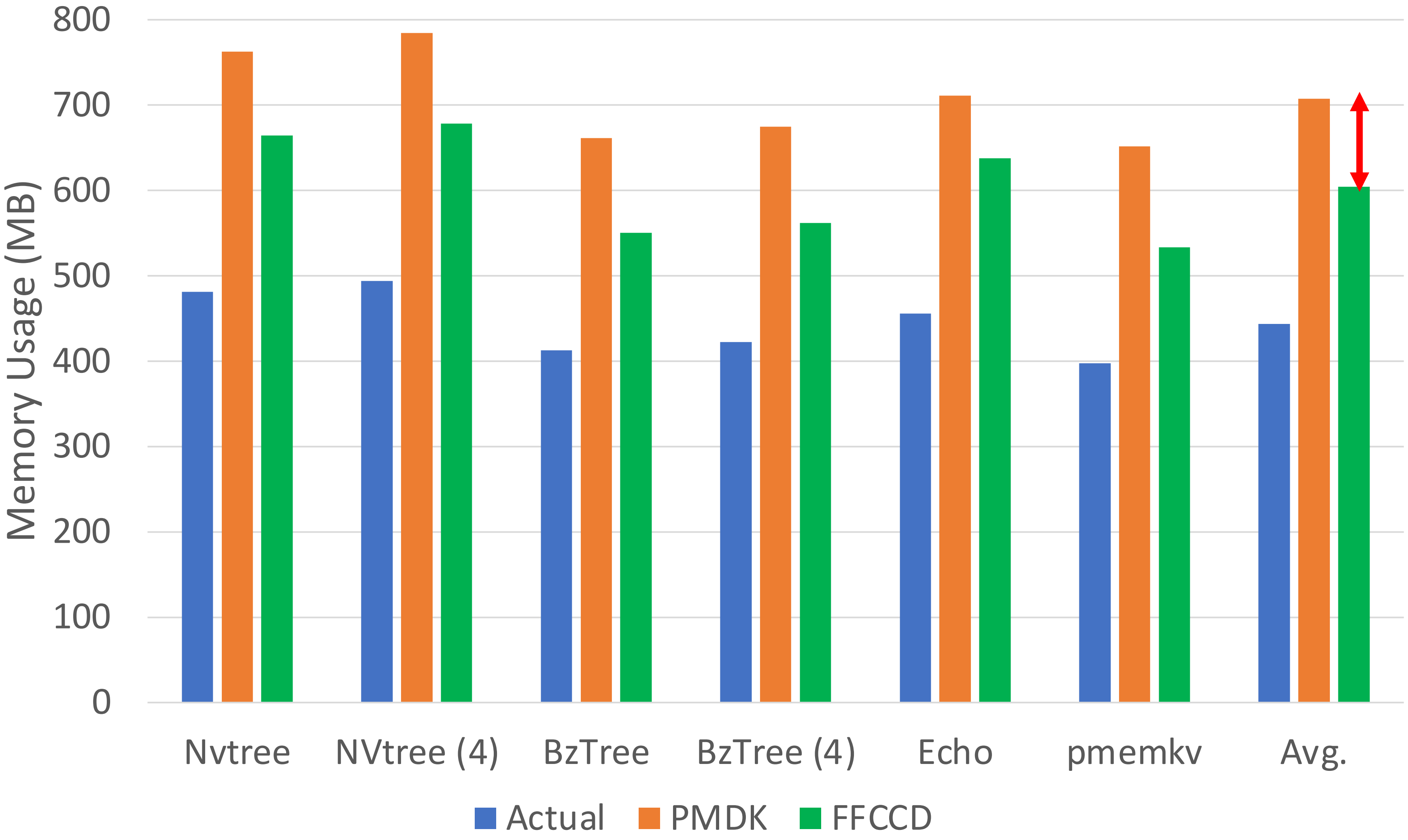
Workload:

1) Initialize data with 5M insertions
2) Delete 4M nodes
3) Insert 4M nodes
4) Delete 4M nodes

Defragmentation

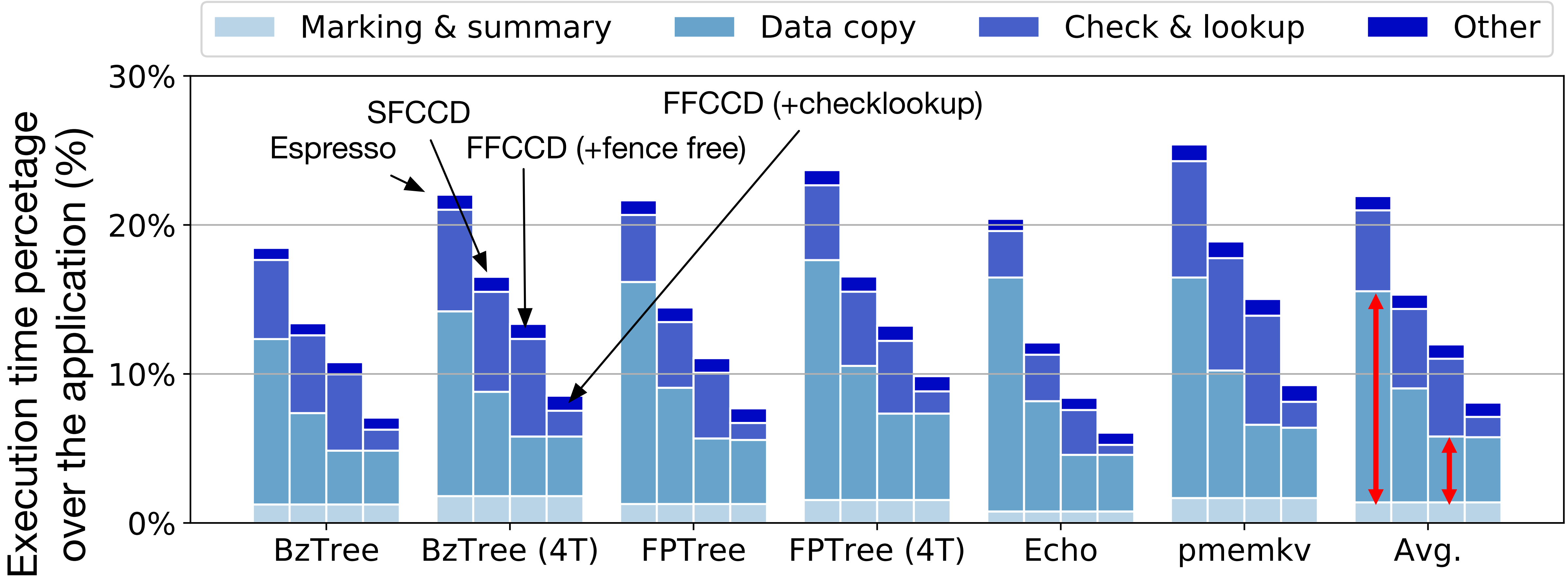


Defragmentation



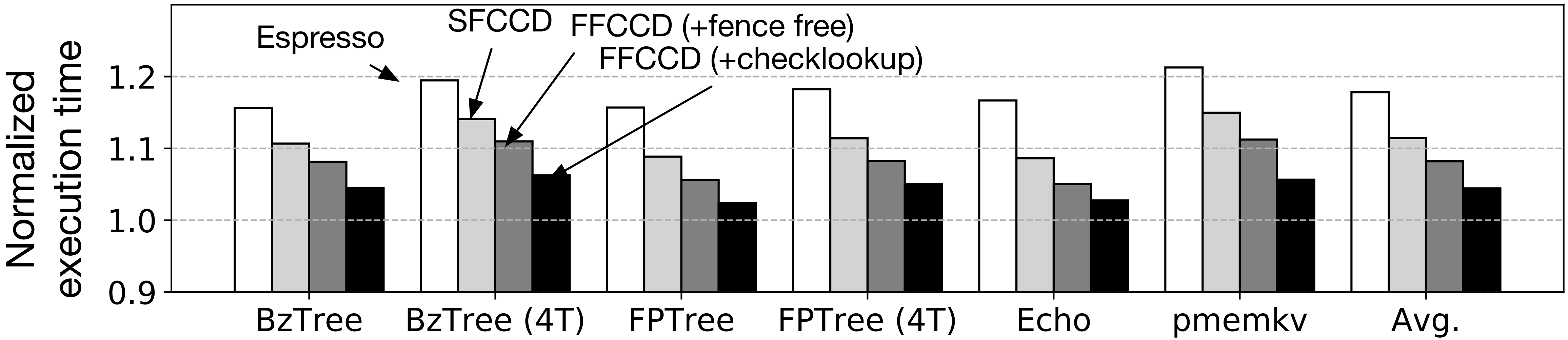
40% Reduction

GC Overhead Breakdown



FFCCD reduces 62% overhead in crash-consistent data copy

Total Execution Time with GC



FFCCD incurs 4.1% overhead than non-defragmentation performance

Conclusions

- We identify sfences as the key challenge to design efficient crash-consistent GC

- We design multiple solutions
  - SFCCD, software-only solution to remove 1 sfence
  - FFCCD, architectural-supp [illegible] ove 2 sfences

Thank you!
Q&A

- FFCCD provides 28–73% fragmentation reduction

- FFCCD incurs 4.1% overhead
  - Improve application performance due to better locality
  - Low overhead from GC