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

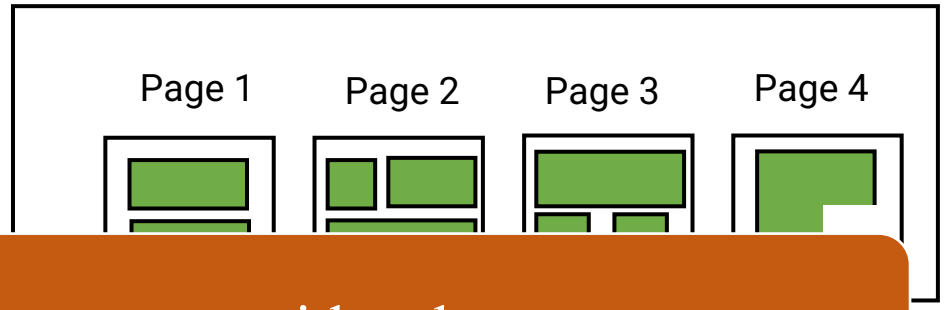
Yuanchao Xu, Chencheng Ye, Yan Solihin, Xipeng Shen

Persistent Fragmentation

Intel Persistent M

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

Persistent Memory Object Pool



Persistent fragmentation gets worsen with subsequent usages

Fragmentation

First process

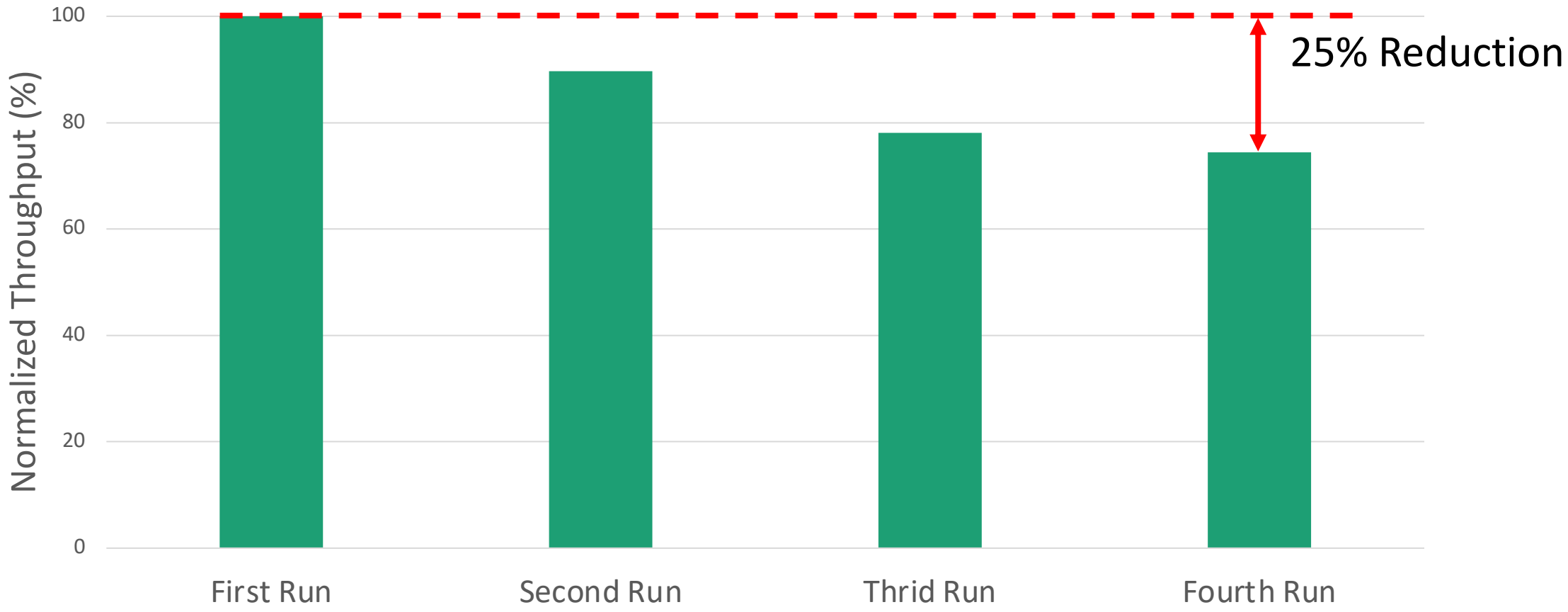
Second process

Third process

Live object

Performance Degradation from Persistent Fragmentation

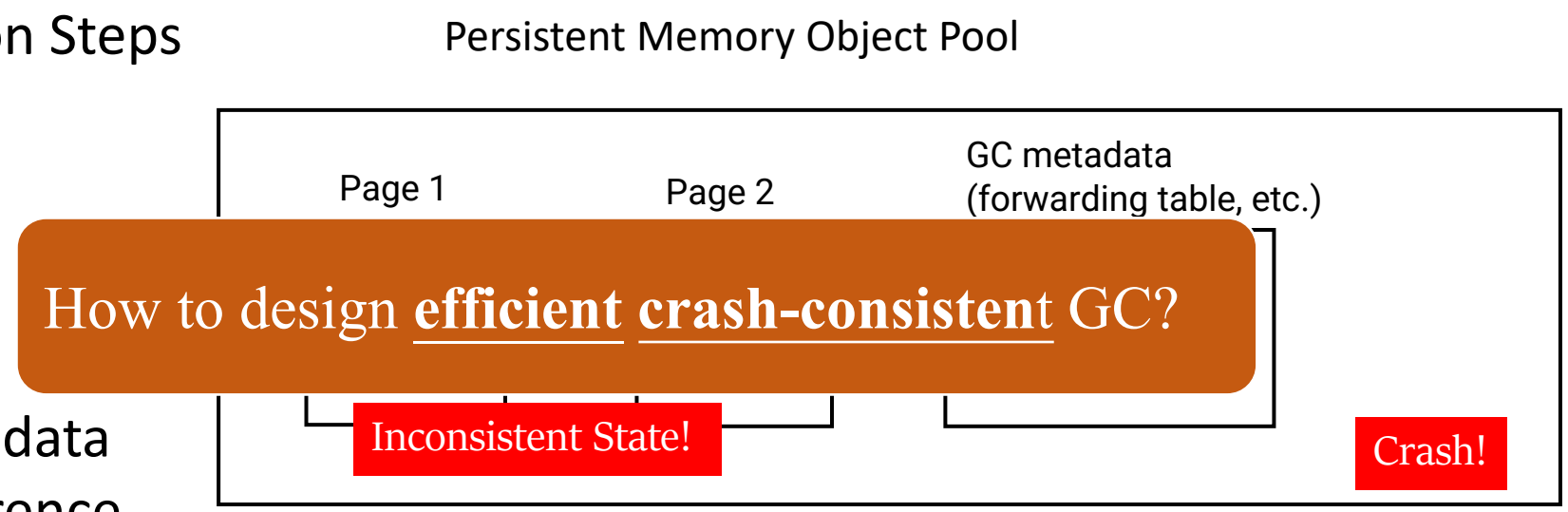
Redis throughput for 2M insertions and 1M deletions



Garbage Collection (GC)

Garbage Collection Steps

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



Live object → Pointer

Contributions

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

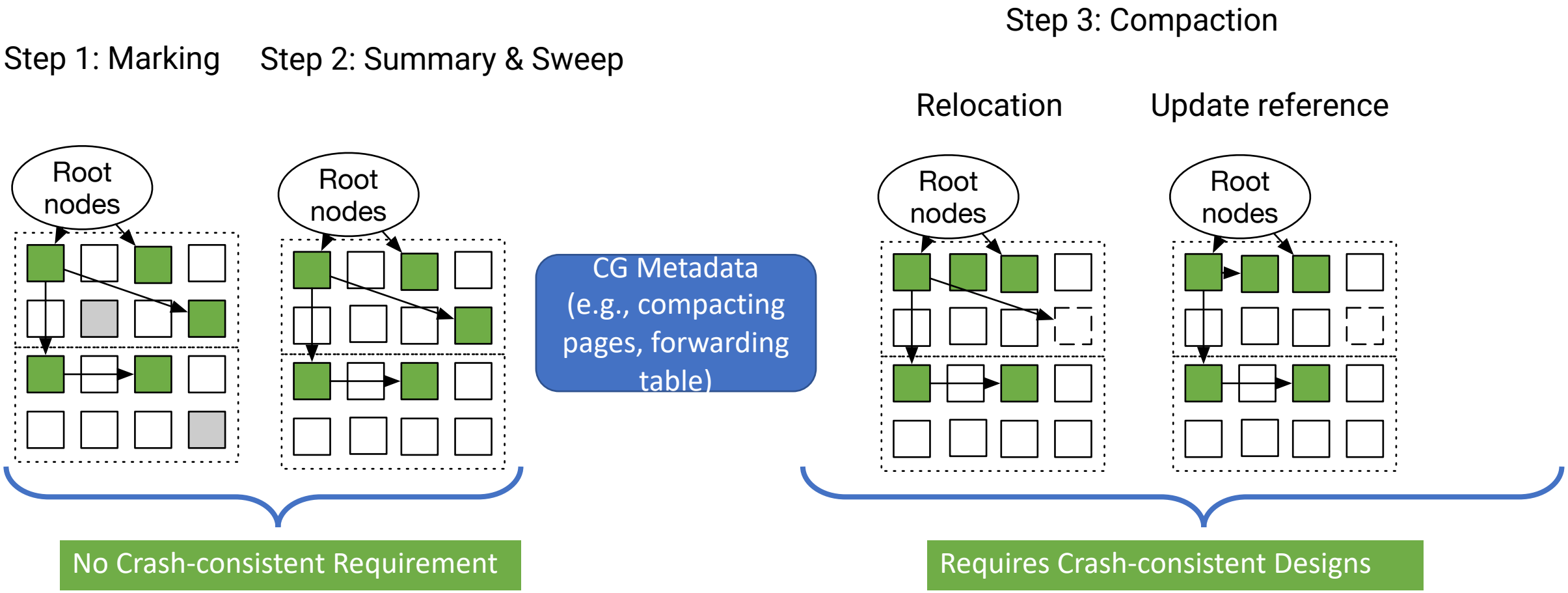
- Design multiple concurrent GC solutions
  - Pure software single-fence crash-consistent design (SFCCD)
  - Architectural support for fence-free crash-consistent design (FFCCD)

- We evaluate designs and show its effectiveness

Agenda

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

Concurrent Garbage Collection



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
```

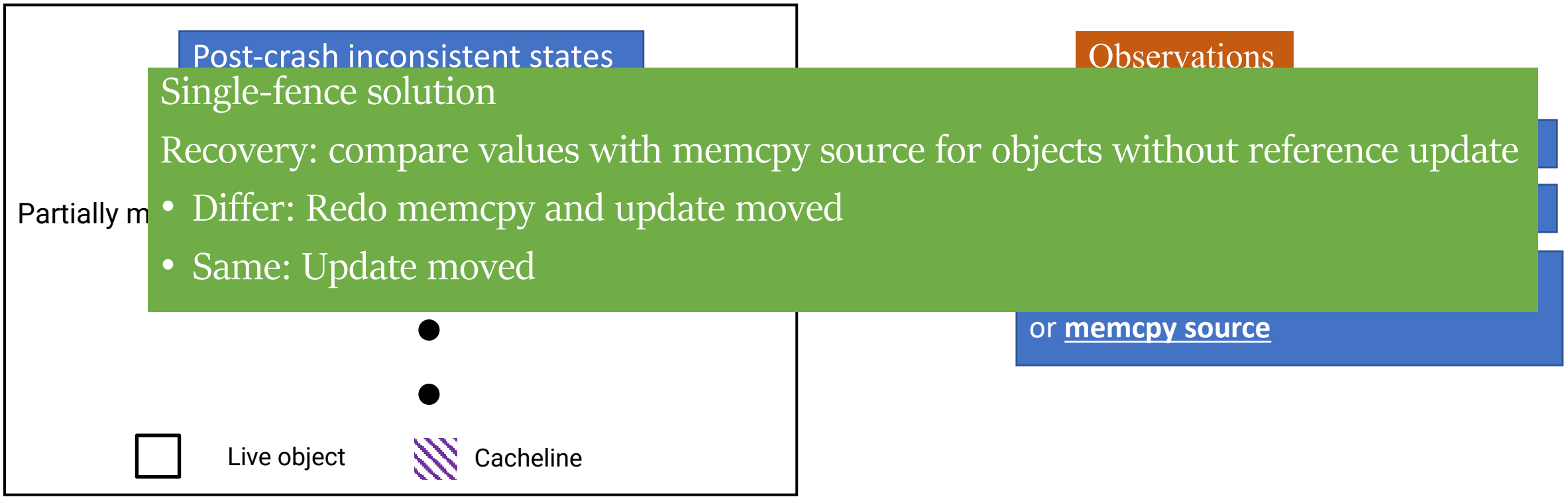
Persist GC metadata before compaction
(e.g., compacting pages, forwarding table)

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

Can we remove the first sfence?

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



Can we remove the second sfence?

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

Numerous possible post-crash inconsistent states

Observations

1) memcpy source data is clean

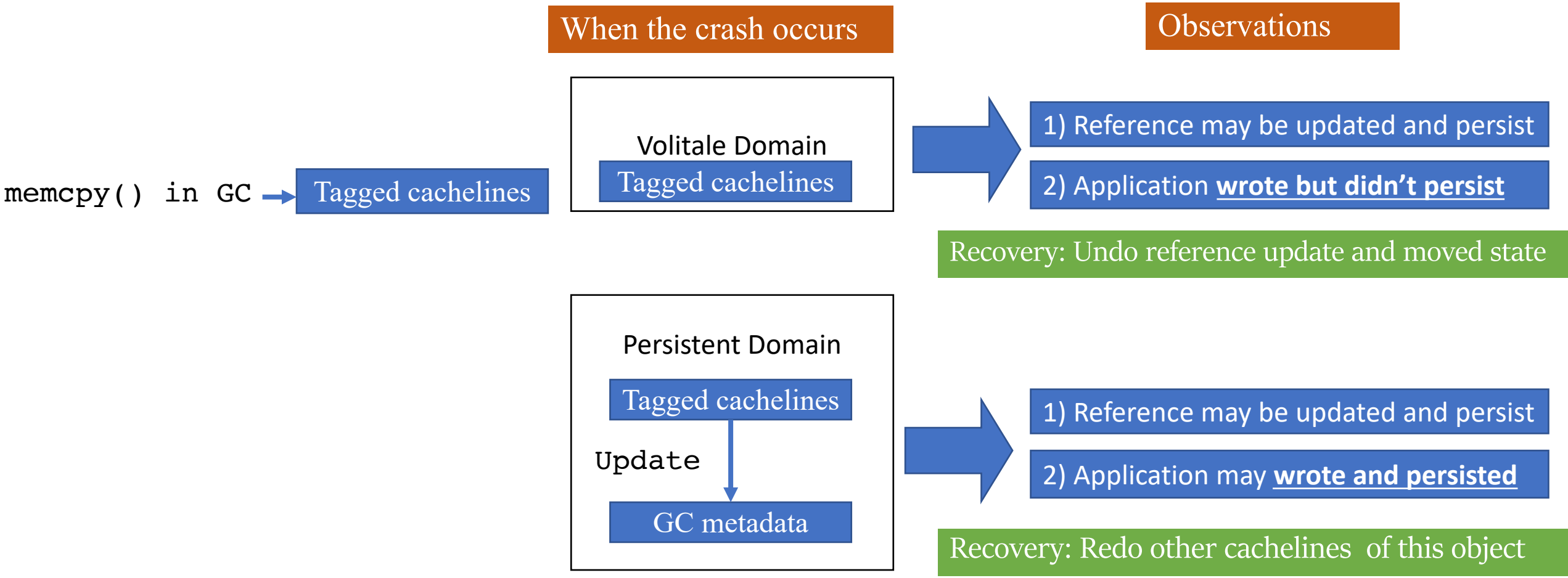
2) References **may be updated**

3) Memcpy destination value is **original**, **memcpy source,** or **the new value overwritten by application**

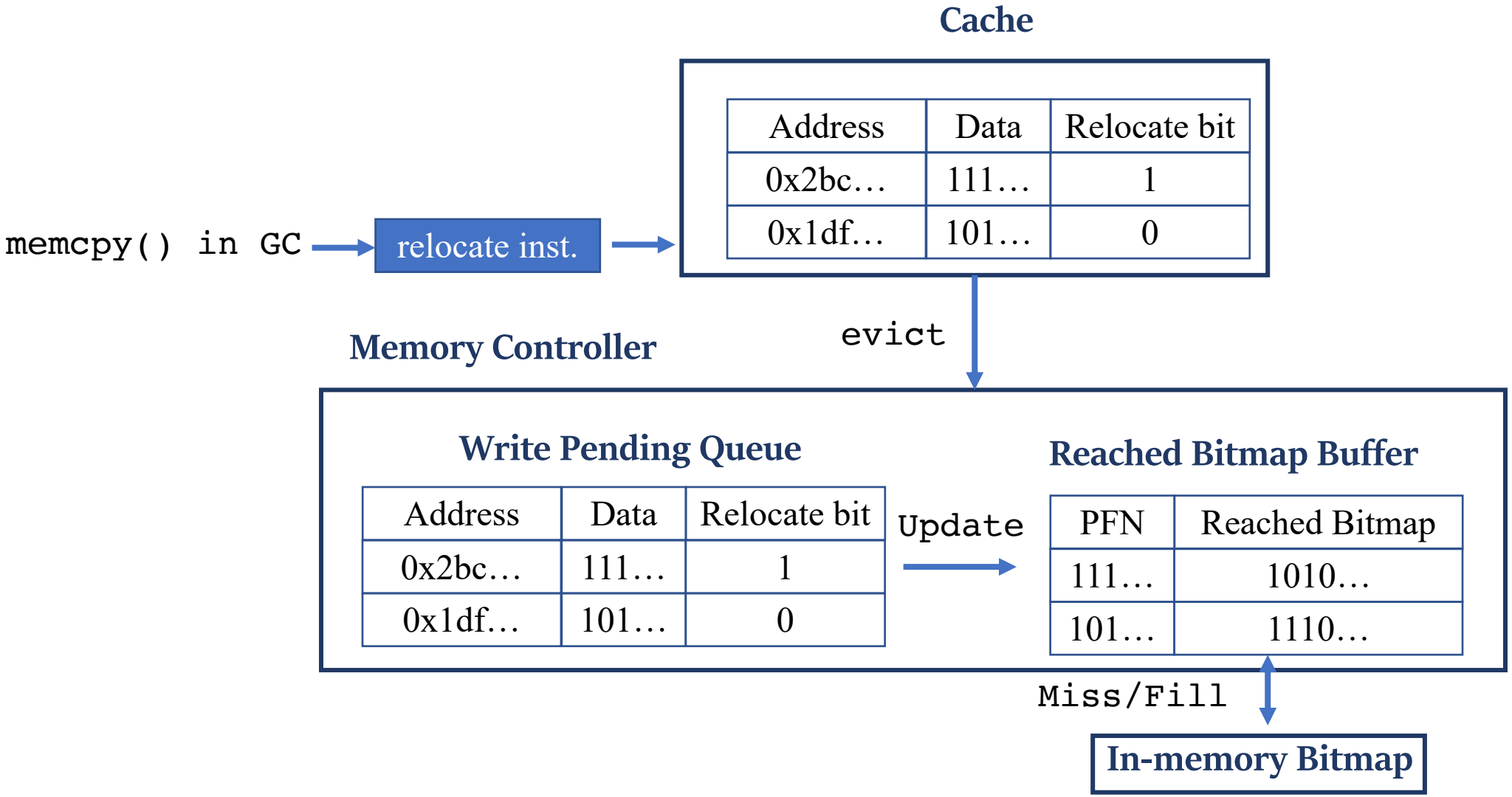
If we can distinguish the original value from the new value

- New value: Redo other cachelines of this object
- Original value: Undo side effects

Idea: Track Relocated Cachelines



Architecture support



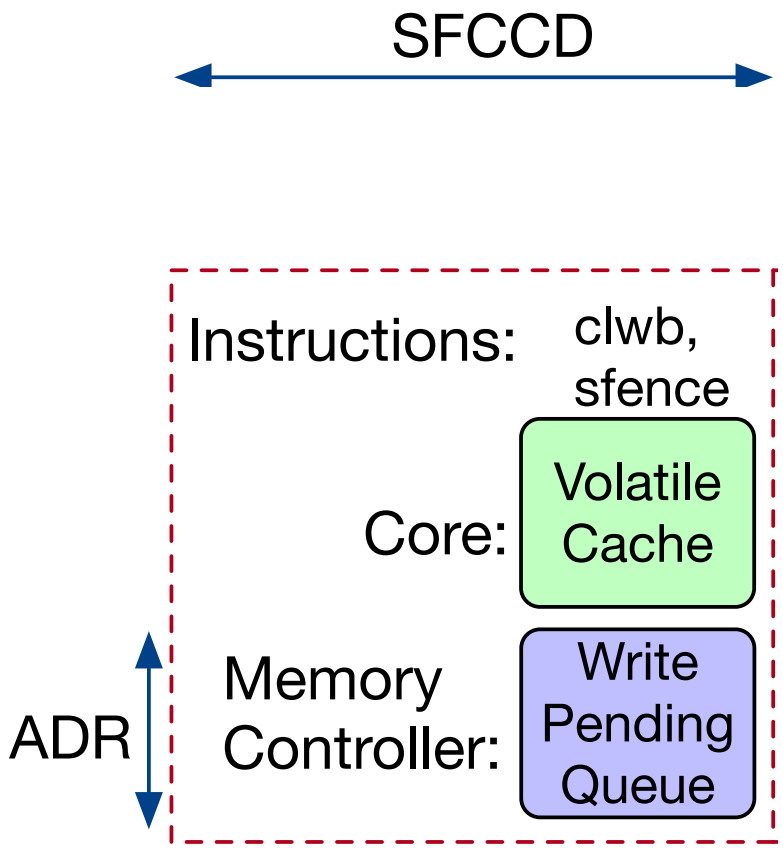
Crash-consistent Concurrent Compaction Baseline

```
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
```

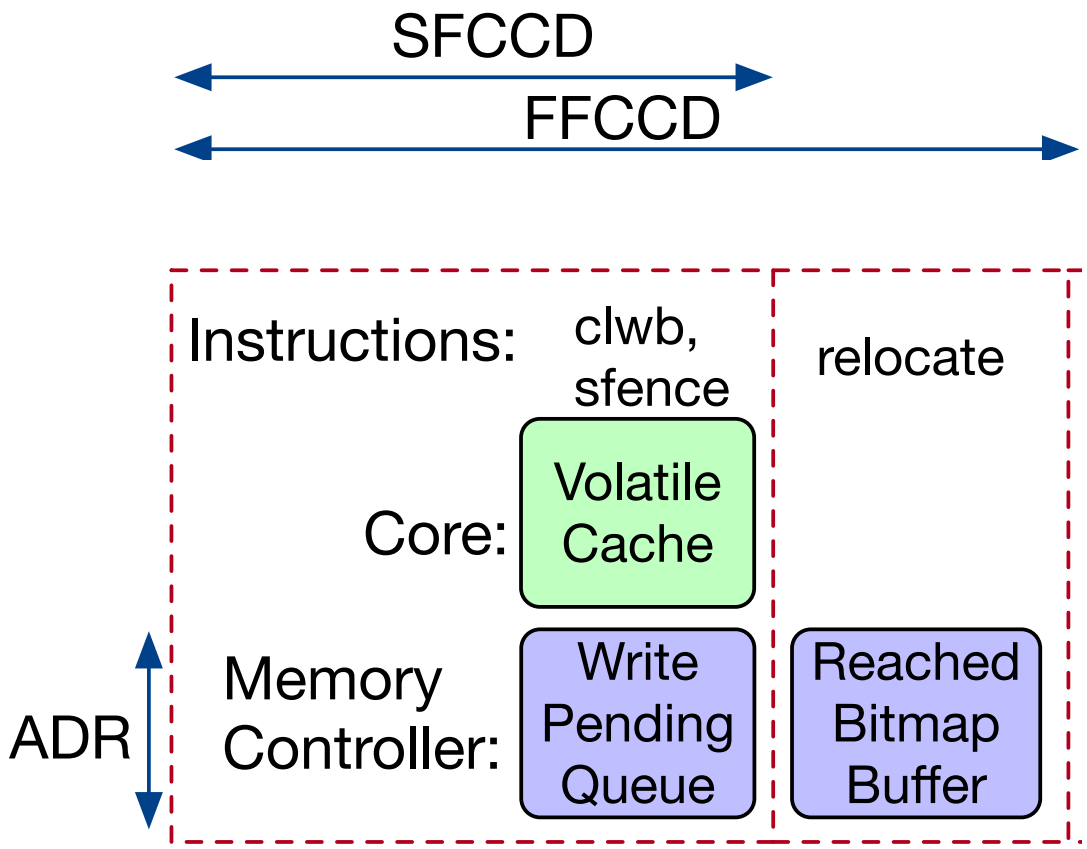
More details in the paper

Fence-free design

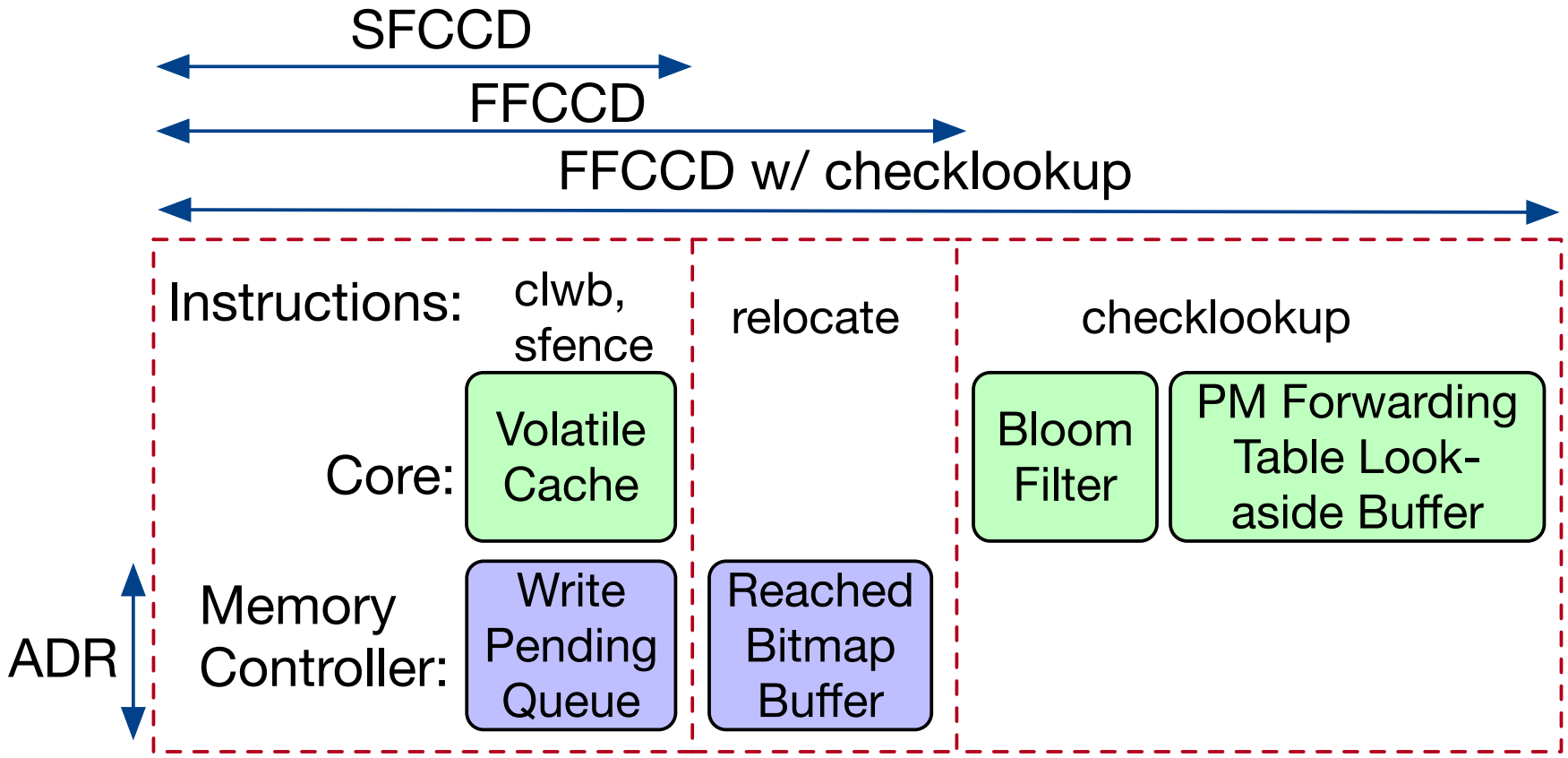
Design Summary



Design Summary



Design Summary



Evaluation

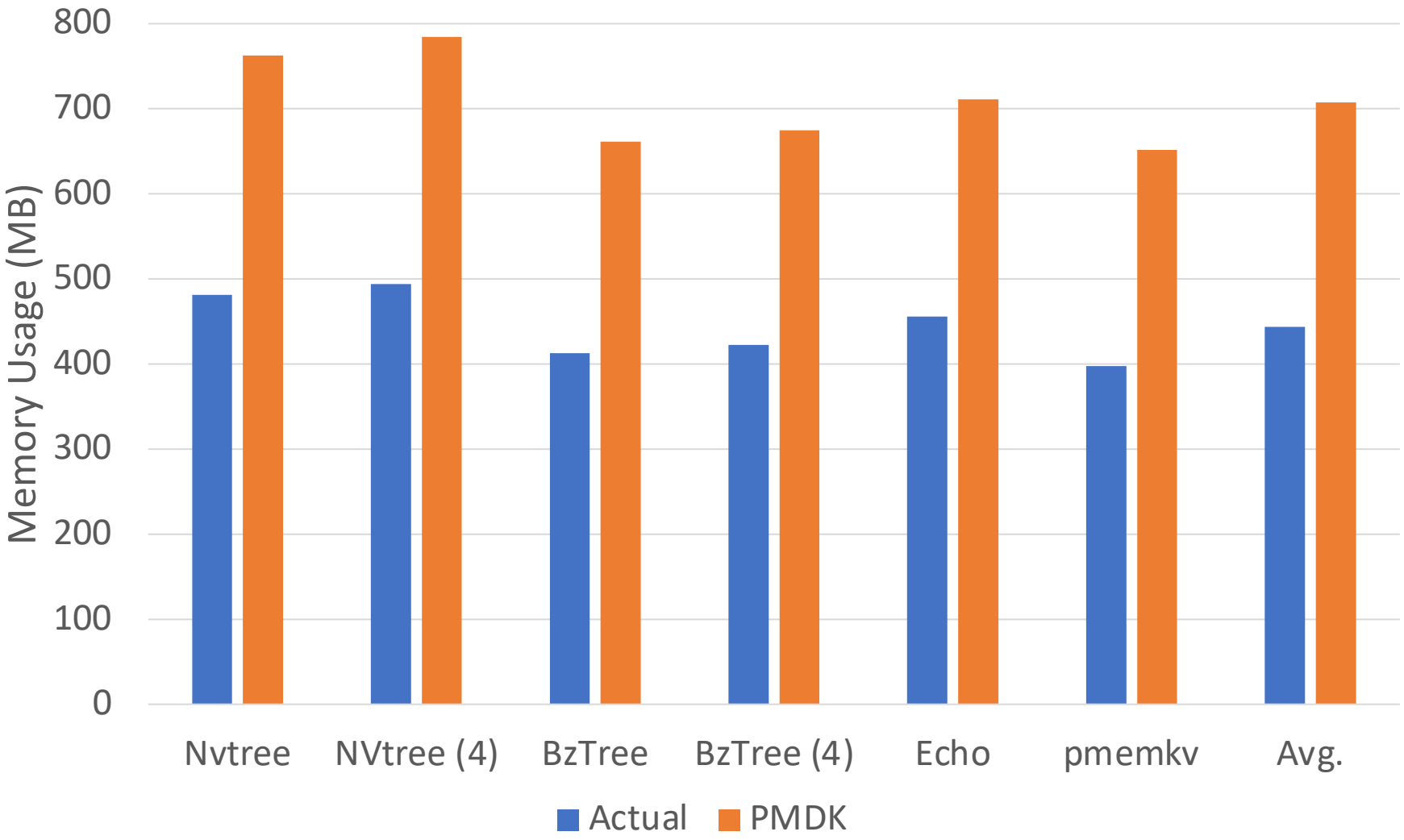
Benchmarks:

- PM data structures
- PM key-value stores

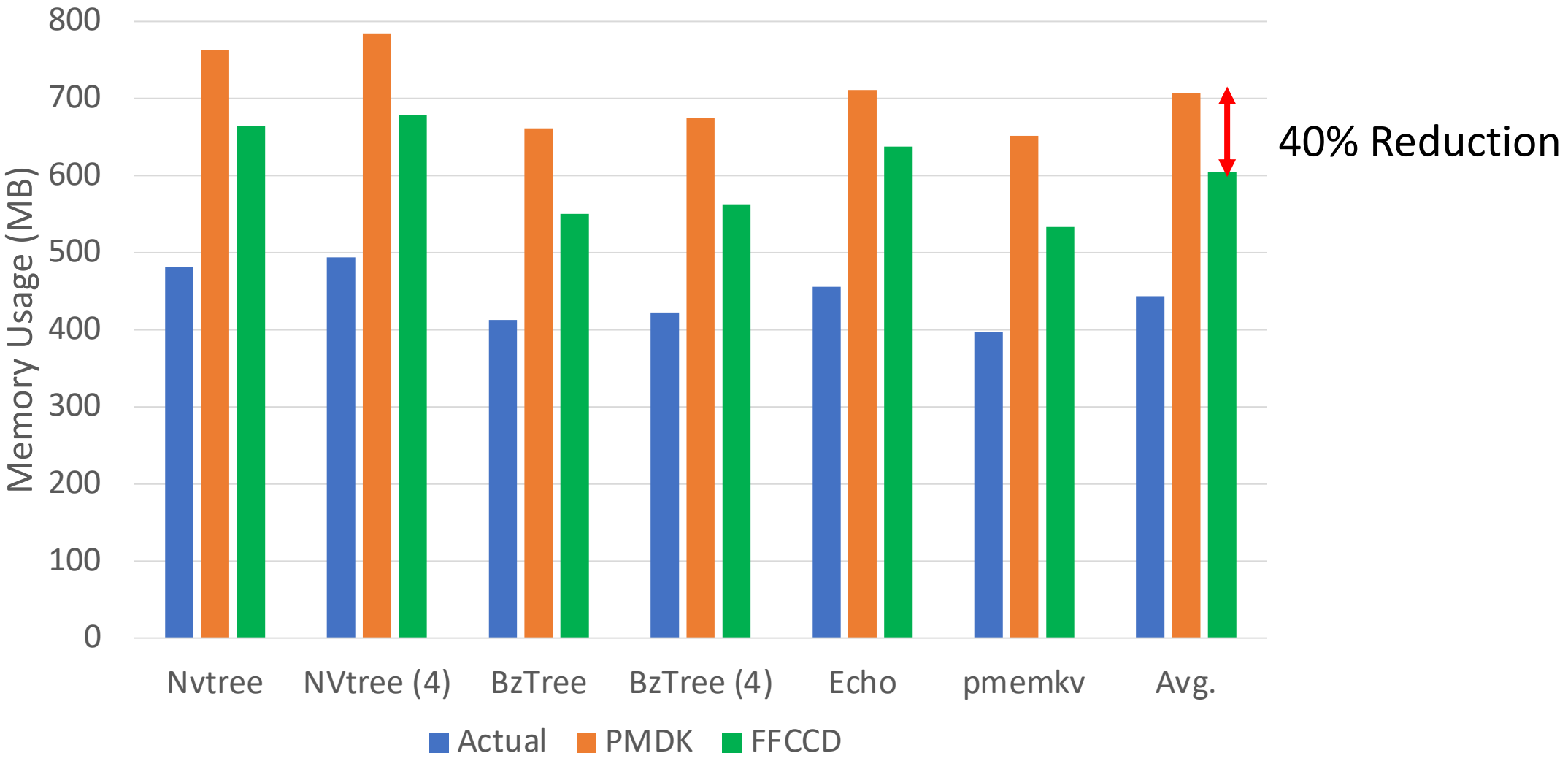
Workload:

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

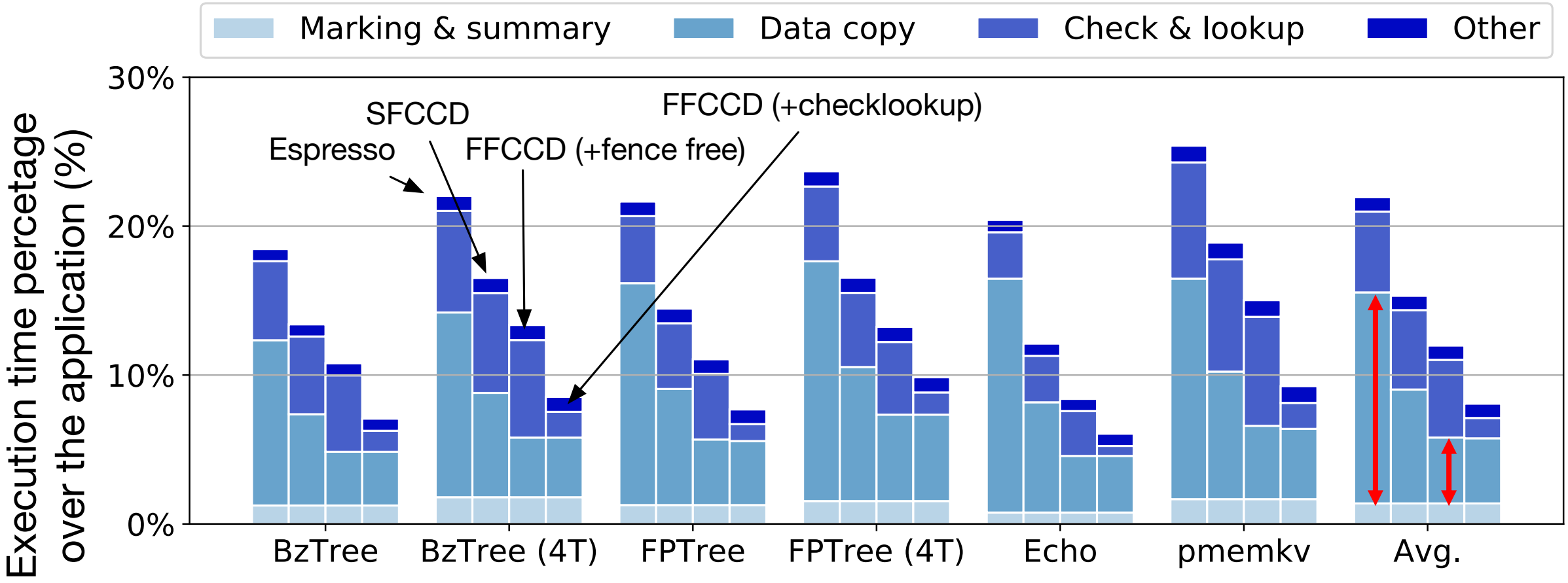
Defragmentation



Defragmentation

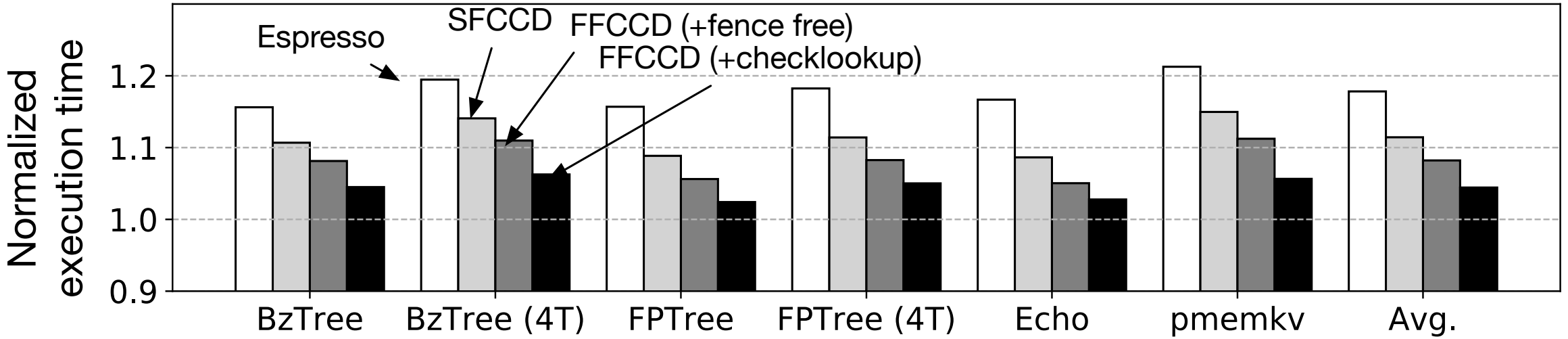


GC Overhead Breakdown



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 ove 2 sfences

Thank you!
Q&A

- FFCCD 28–73% fragmentation reduction

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

Persistent Memory Programming Model Enable Concurrent Compaction

```
o->first = input  //o points to objA
```

Under PMDK libpmemobj

```
D_RW(o)->first = input  //o points to objA
```

D_RW() returns a writable normal pointer from a persistent pointer.

Insert read barrier into D_RW()/D_RO() to enable concurrent compaction

- Typed allocation
- Root nodes
- Persistent pointers and APIs