Reducing Write Barrier Overheads for Orthogonal Persistence

Yilin Zhang
University of Tokyo

Omkar Dilip Dhawal
IIT Madras

V. Krishna Nandivada
IIT Madras

Shigeru Chiba
University of Tokyo

Tomoharu Ugawa
University of Tokyo

SLE 2024, Pasadena

Reducing Write Barrier Overheads for Orthogonal Persistence

Persistence

- Non-volatile memory (NVM)
  - Intel Optane
  - Objects in byte-addressable NVM can be accessed in the same way as those in DRAM
  - Faster than SSD but slower than DRAM
  - Hybrid systems used in practice

Persistence

- Non-volatile memory (NVM)
  - Intel Optane
  - Objects in byte-addressable NVM can be accessed in the same way as those in DRAM
  - Faster than SSD but slower than DRAM
  - Hybrid systems used in practice
- Issues
  - Which objects must be made persistent ? When should the object become persistent ?
  - Tedious and error-prone task for programmers

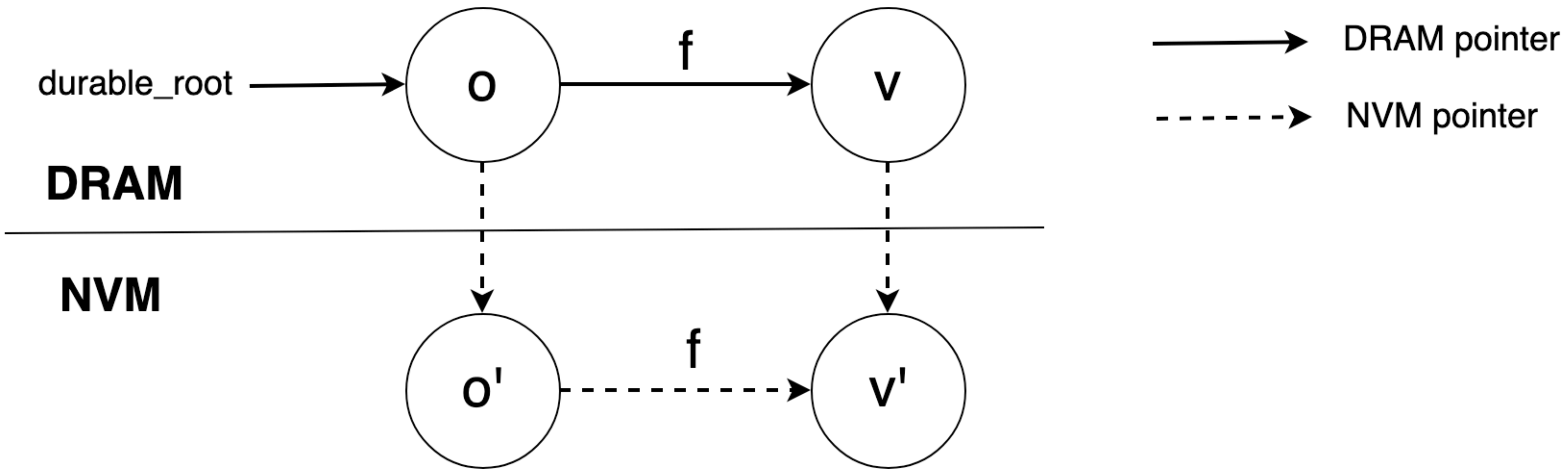
Reducing Write Barrier Overheads for Orthogonal Persistence

Orthogonal Persistence

- Programmers can annotate static fields as durable roots
- Persistence of objects decided by reachability

Orthogonal Persistence

- Programmers can annotate static fields as durable roots
- Persistence of objects decided by reachability
- Objects reachable from persistent roots are copied to NVM without programmer's intervention (Replication-based object persistence) [Matsumoto et al. 2022]



Orthogonal Persistence

- Persistence of objects decided by reachability
- Programmers can annotate static fields as persistent roots
- Objects reachable from persistent roots are copied to NVM without programmer's intervention
- Issues
  - Java supports multi-threading
  - Concurrent access: One thread is modifying an object while another thread is attempting to copy it to NVM.

Concurrent Access

Case 1: Replica of o is absent

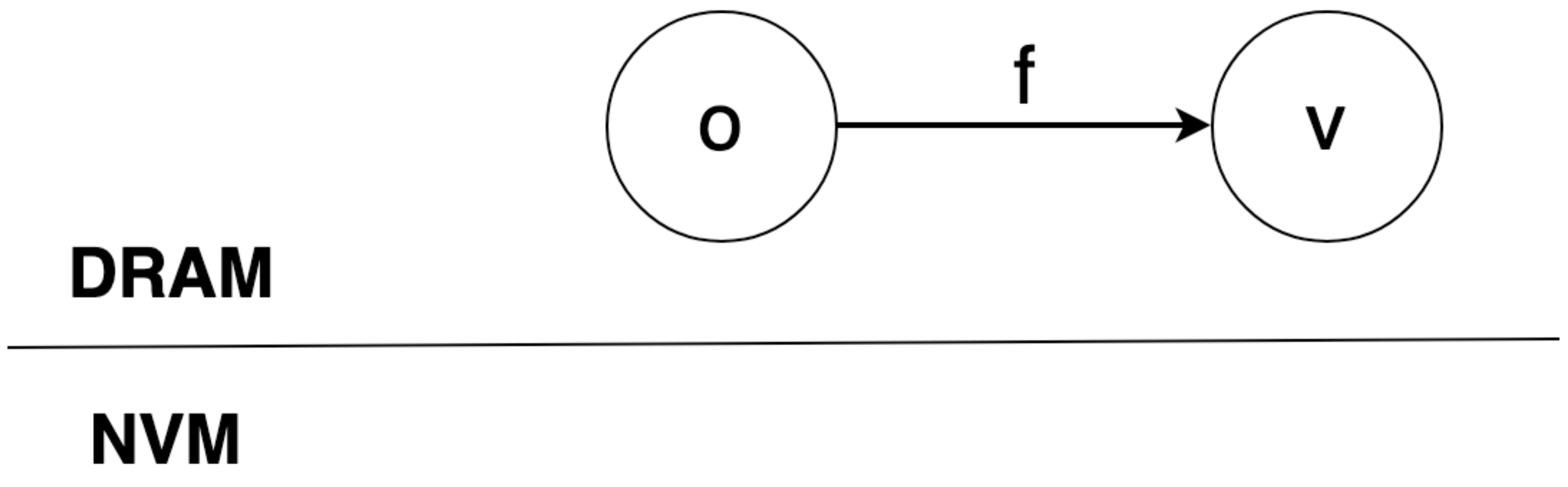
Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
        o[f] = v

        o' = o.replica
        if o' == NULL : return
        make_persistent(v)
        v' = v.replica
        o'[f] = v'
        CLWB(&o'[f])
```



Concurrent Access

Case 1: Replica of o is absent

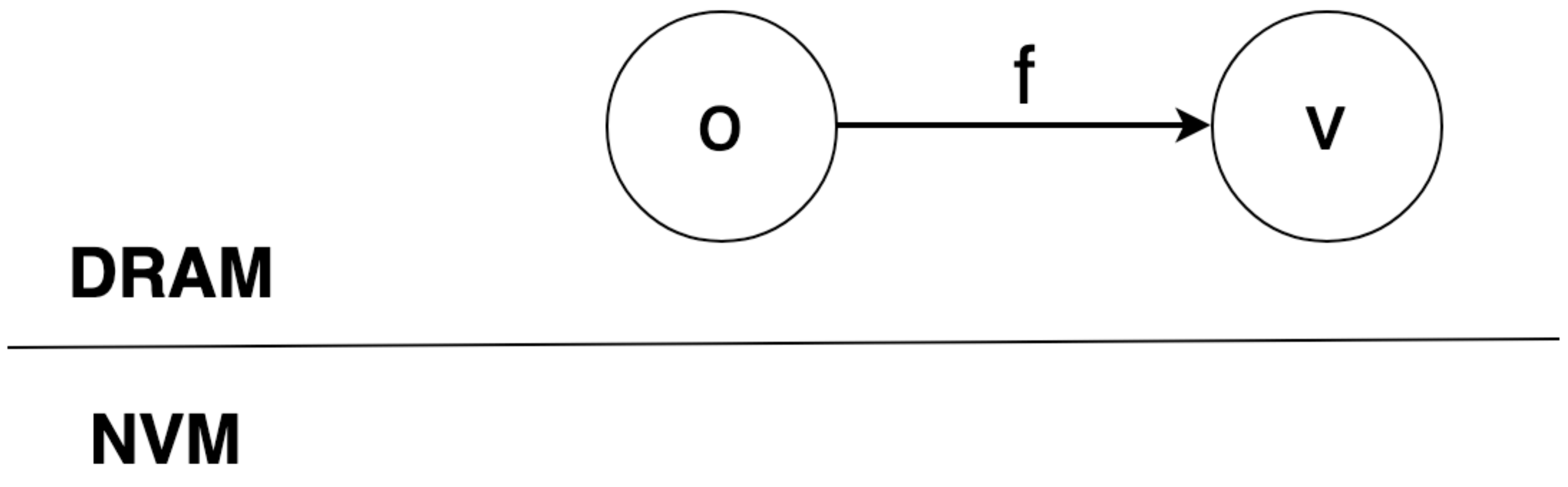
Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
        o[f] = v

        o' = o.replica
        if o' == NULL : return
        make_persistent(v)
        v' = v.replica
        o'[f] = v'
        CLWB(&o'[f])
```



Concurrent Access

Case 1: Replica of o is absent

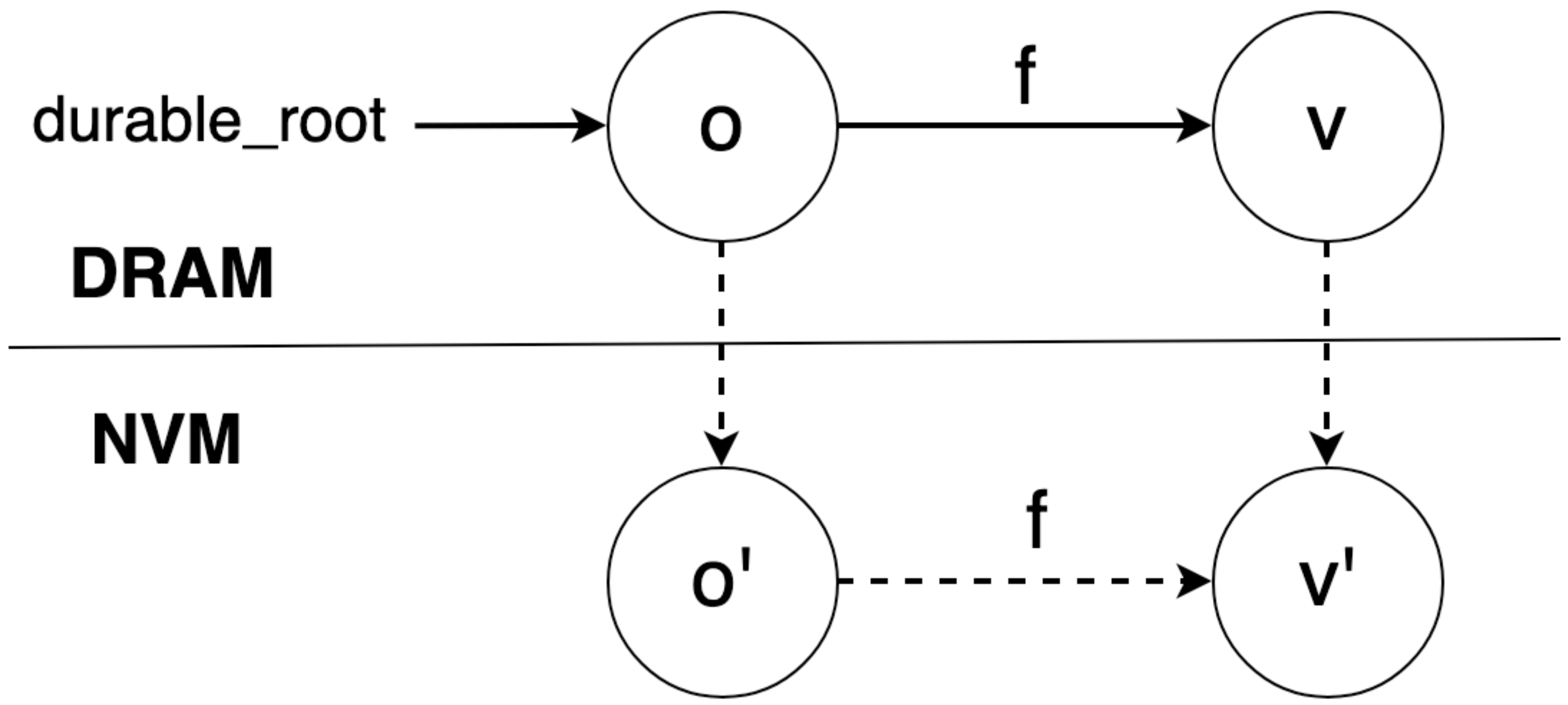
Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
      o[f] = v

      o’ = o.replica
      if o’ == NULL : return
      make_persistent(v)
      v’ = v.replica
      o’[f] = v’
      CLWB(&o’[f])
```



Concurrent Access

Case 2: Replica of o is already present

Concurrent Access

Case 2: Replica of o is present

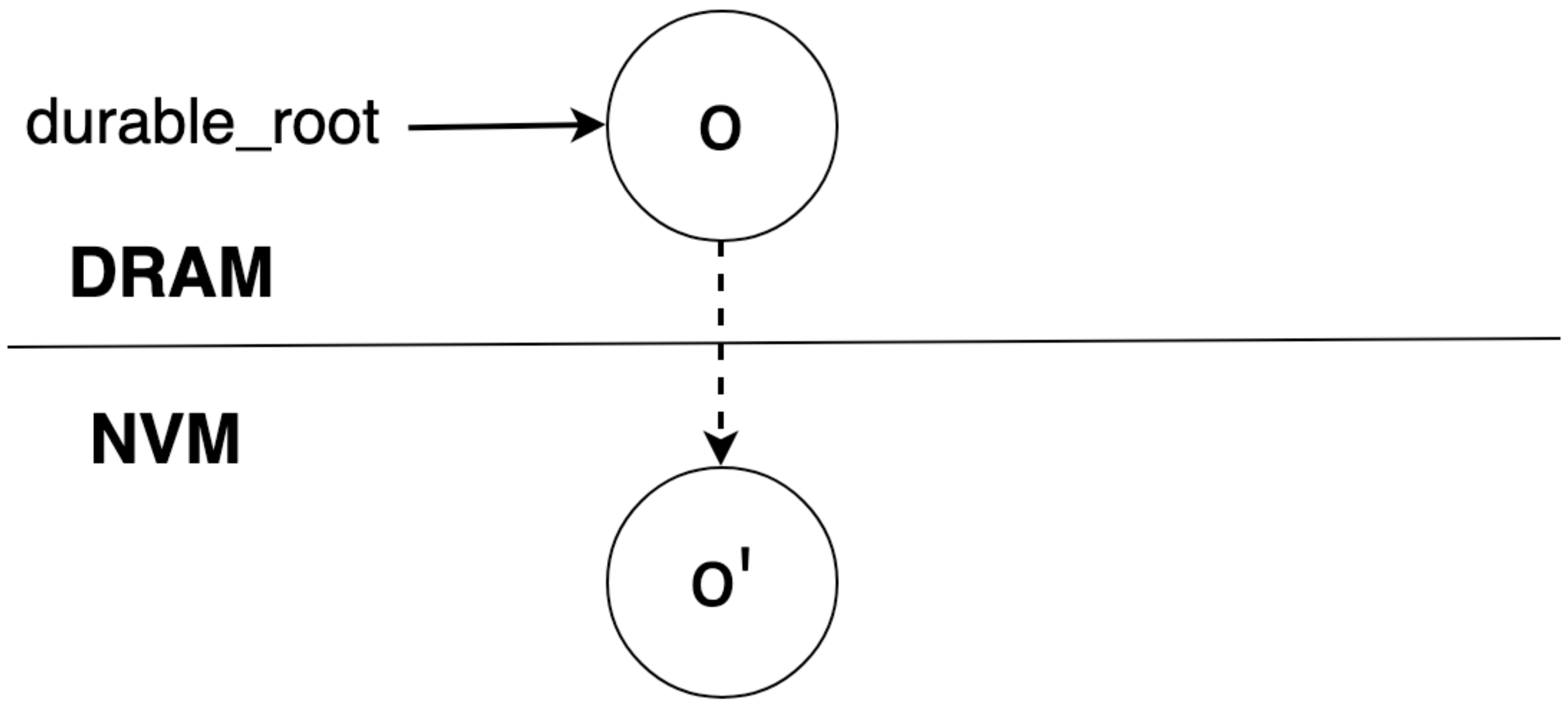
Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
        o[f] = v

        o' = o.replica
        if o' == NULL : return
        make_persistent(v)
        v' = v.replica
        o'[f] = v'
        CLWB(&o'[f])
```



Concurrent Access

Case 2: Replica of o is present

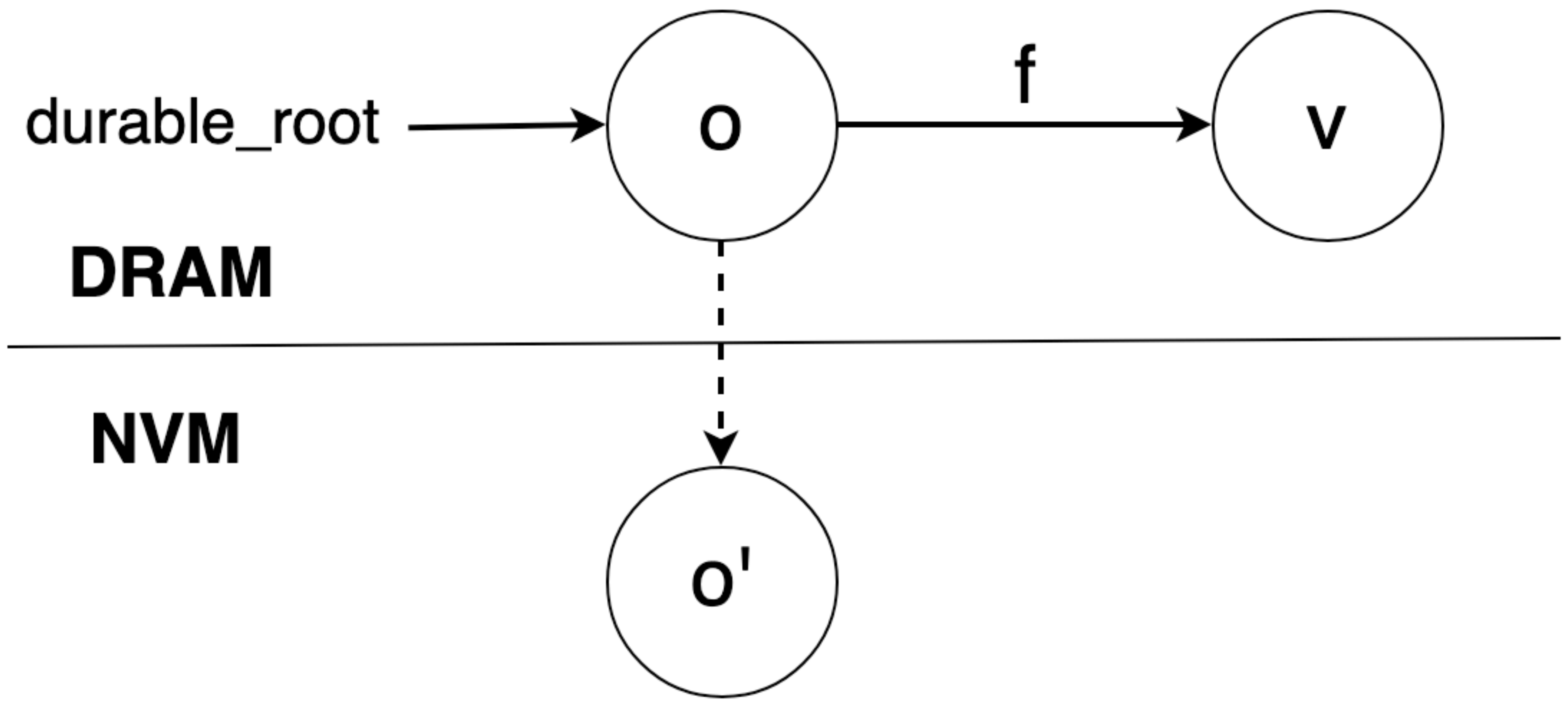
Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
    o[f] = v

    o' = o.replica
    if o' == NULL : return
    make_persistent(v)
    v' = v.replica
    o'[f] = v'
    CLWB(&o'[f])
```



Concurrent Access

Case 2: Replica of o is present

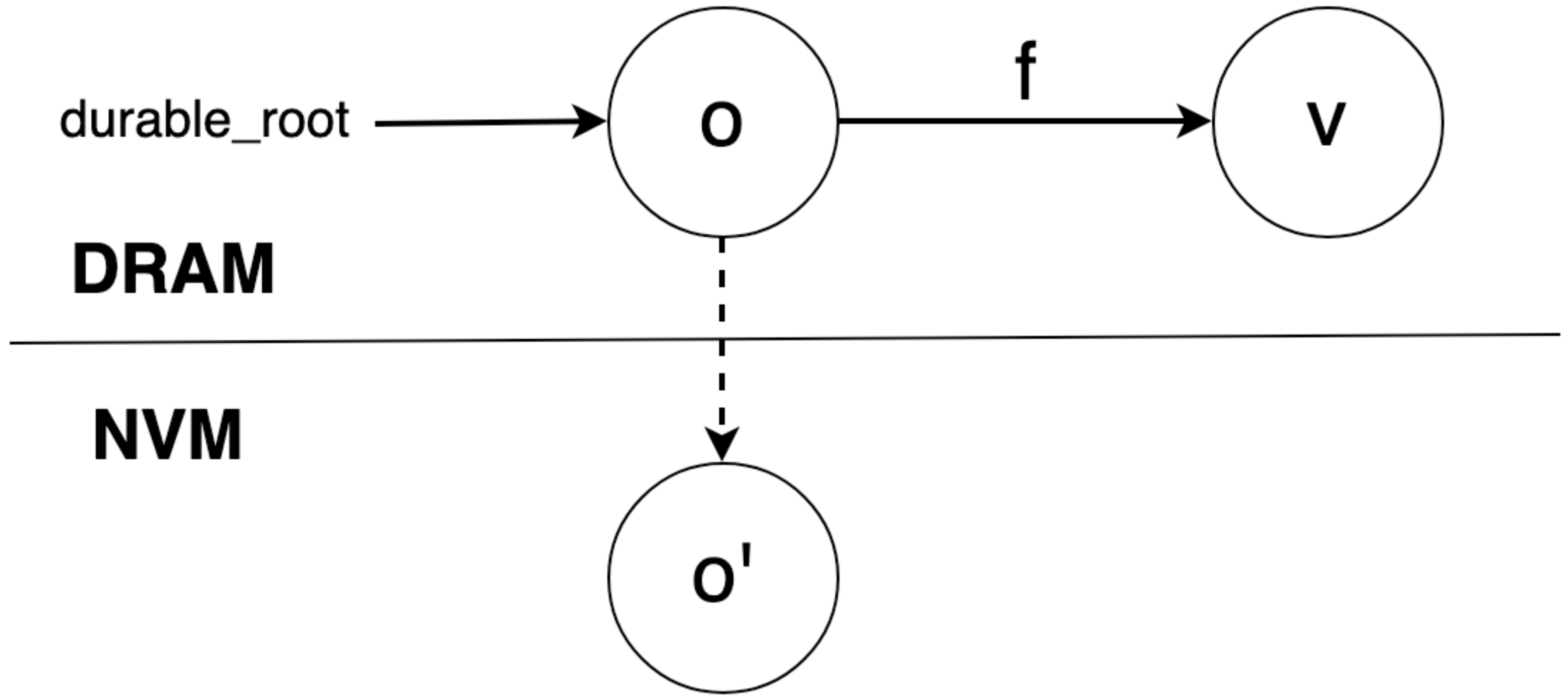
Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
    o[f] = v

    o’ = o.replica
    if o’ == NULL : return
    make_persistent(v)
    v’ = v.replica
    o’[f] = v’
    CLWB(&o’[f])
```



Concurrent Access

Case 2: Replica of o is present

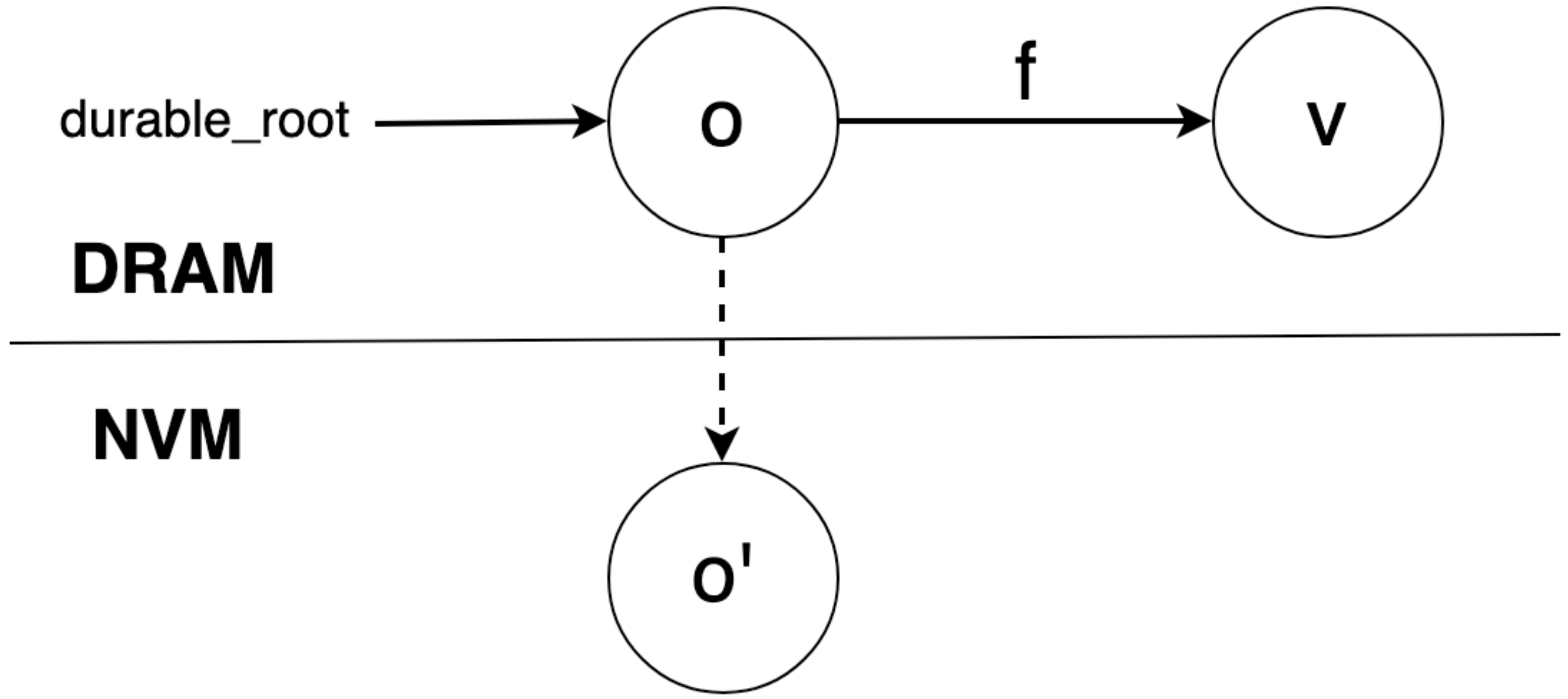
Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
    o[f] = v

    o' = o.replica
    if o' == NULL : return
    make_persistent(v)
    v' = v.replica
    o'[f] = v'
    CLWB(&o'[f])
```



Concurrent Access

Case 2: Replica of o is present

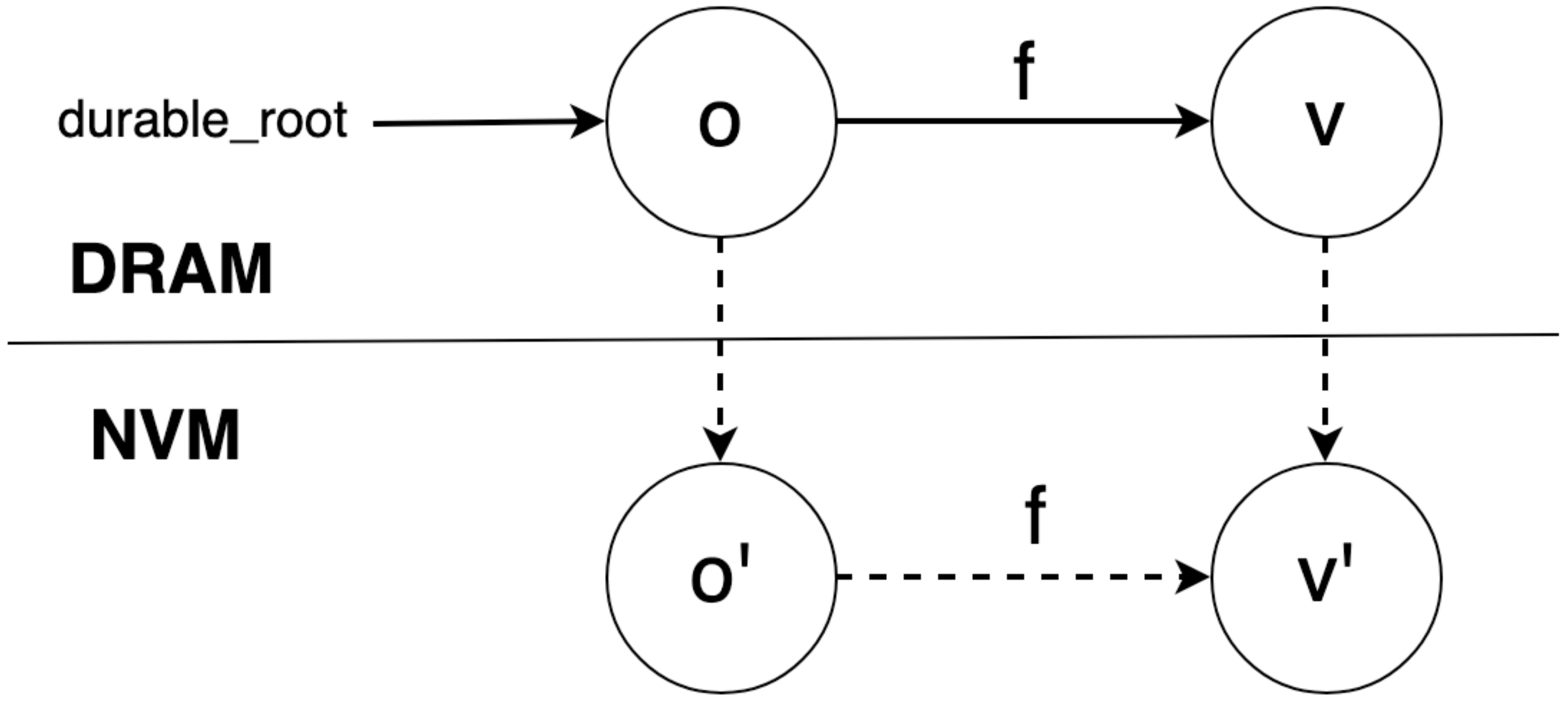
Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
        o[f] = v

        o' = o.replica
        if o' == NULL : return
        make_persistent(v)
        v' = v.replica
        o'[f] = v'
        CLWB(&o'[f])
```



Concurrent Access

Case 3: Instructions are reordered (x86 Weak memory consistency model)

Concurrent Access

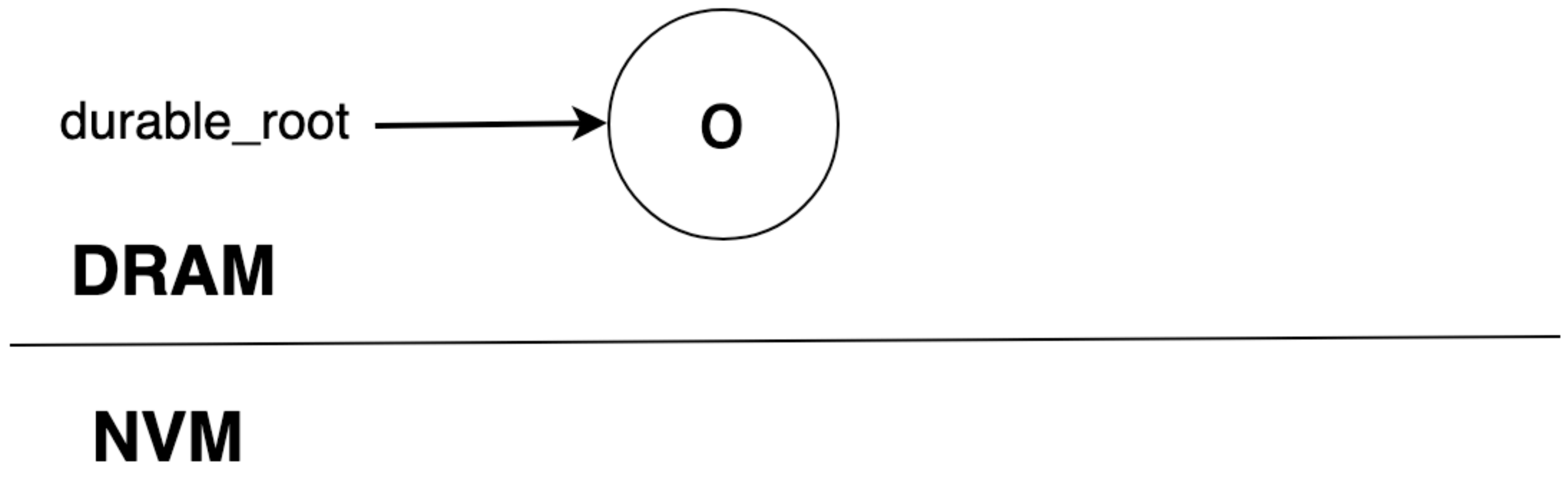
Case 3: Read at Line 4 is reordered with Write at Line 2

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o' = o.replica
5        if o' == NULL : return
6        make_persistent(v)
7        v' = v.replica
8        o'[f] = v'
9        CLWB(&o'[f])
```



Concurrent Access

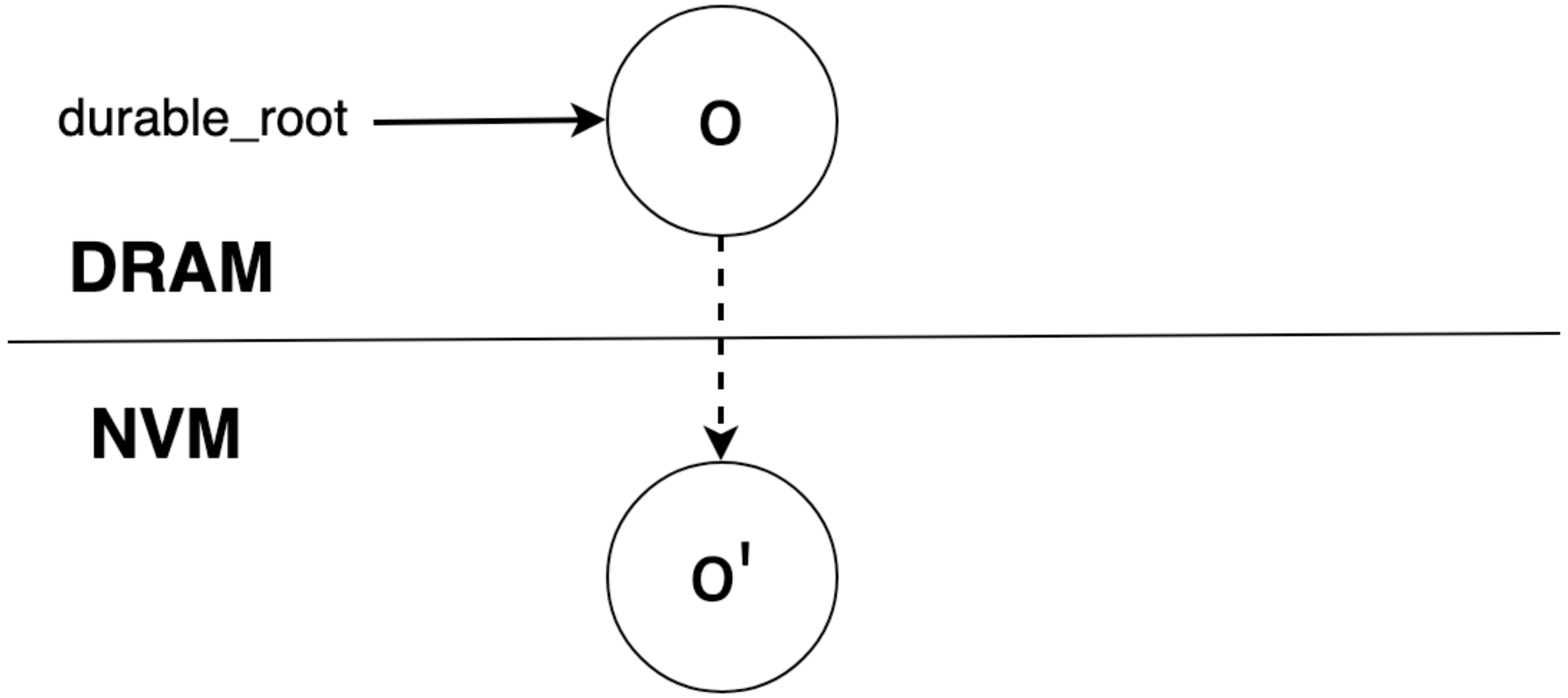
Case 3: Read at Line 4 is reordered with Write at Line 2

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o’ = o.replica
5        if o’ == NULL : return
6        make_persistent(v)
7        v’ = v.replica
8        o’[f] = v’
9        CLWB(&o’[f])
```



Concurrent Access

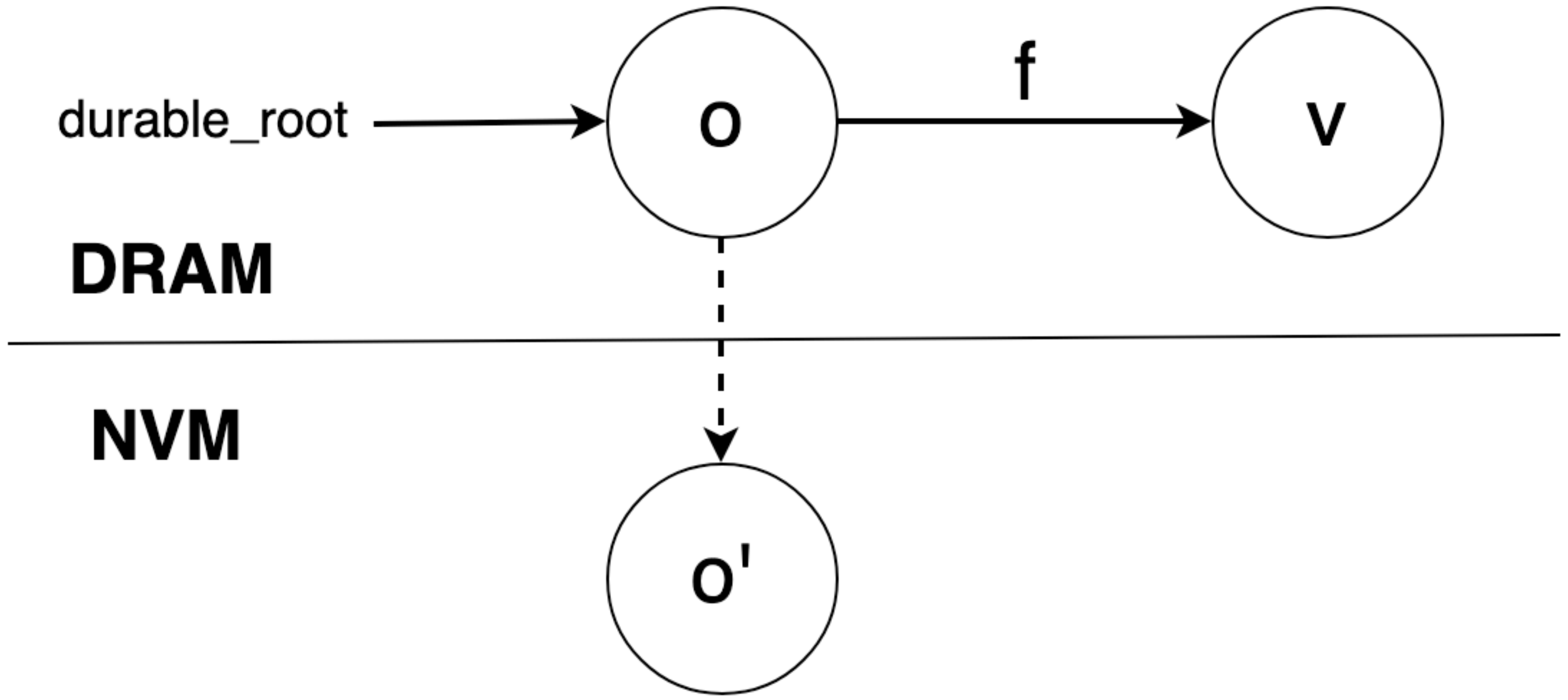
Case 3: Read at Line 4 is reordered with Write at Line 2

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o’ = o.replica
5        if o’ == NULL : return
6        make_persistent(v)
7        v’ = v.replica
8        o’[f] = v’
9        CLWB(&o’[f])
```



Concurrent Access

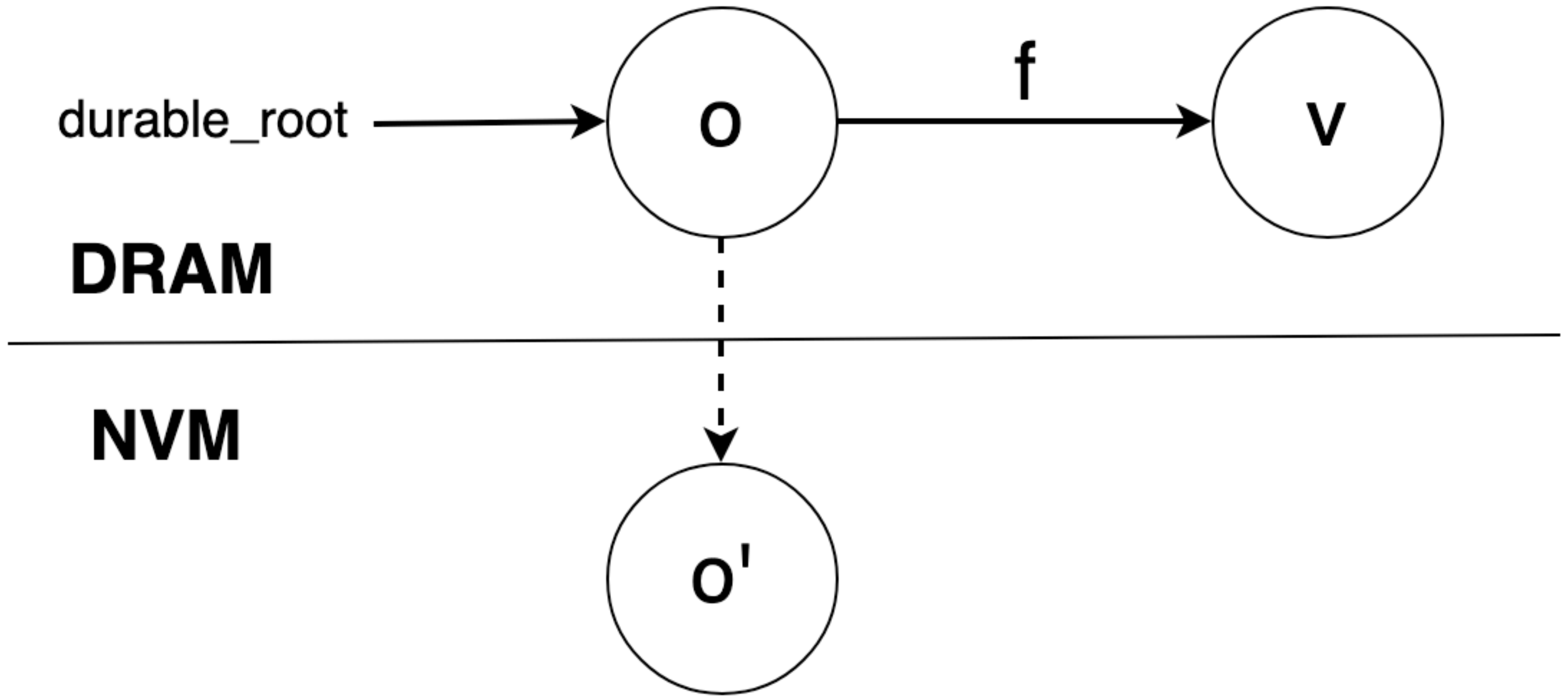
Case 3: Read at Line 4 is reordered with Write at Line 2

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o' = o.replica
5        if o' == NULL : return
6        make_persistent(v)
7        v' = v.replica
8        o'[f] = v'
9        CLWB(&o'[f])
```



Concurrent Access

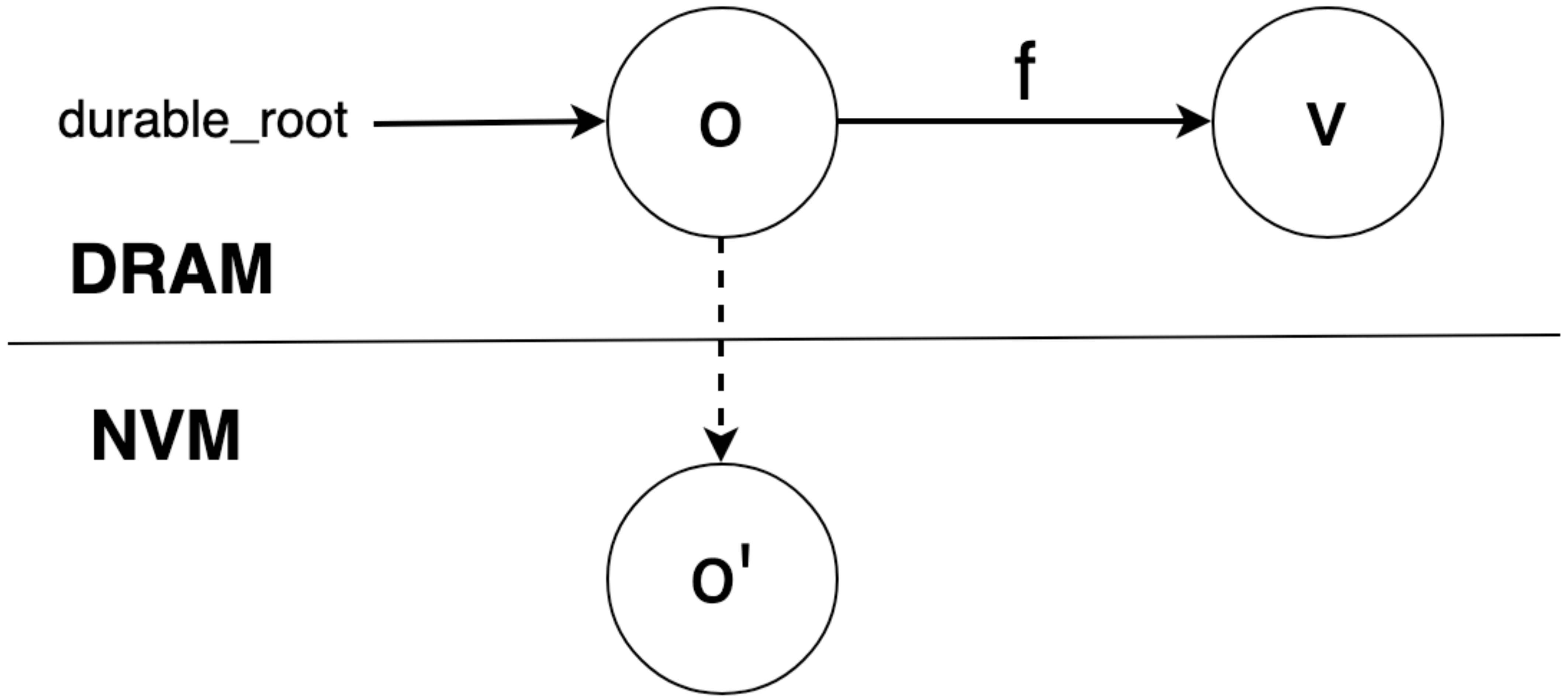
Case 3: Read at Line 4 is reordered with Write at Line 2

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2       o[f] = v
3
4       o' = o.replica
5       if o' == NULL : return
6       make_persistent(v)
7       v' = v.replica
8       o'[f] = v'
9       CLWB(&o'[f])
```



Copies inconsistent !!

[Matsumoto et al. 2022]

Reducing Write Barrier Overheads for Orthogonal Persistence

Write Barrier

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
    o[f] = v
    MFENCE
    o' = o.replica
    if o' == NULL : return
    make_persistent(v)
    v' = v.replica
    o'[f] = v'
    CLWB(&o'[f])
```

- **MFENCE** ensures all preceding load and store instructions become globally visible before any that follow it.
- Writer-wait approach
- **Issue** - MFENCE is executed for all field write instructions.

Write Barrier

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
    o[f] = v
    MFENCE
    o' = o.replica
    if o' == NULL : return
    make_persistent(v)
    v' = v.replica
    o'[f] = v'
    CLWB(&o'[f])
```

- **MFENCE** ensures all preceding load and store instructions become globally visible before any that follow it.
- Writer-wait approach

Write Barrier

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
    o[f] = v
    MFENCE
    o' = o.replica
    if o' == NULL : return
    make_persistent(v)
    v' = v.replica
    o'[f] = v'
    CLWB(&o'[f])
```

- **MFENCE** ensures all preceding load and store instructions become globally visible before any that follow it.
- Writer-wait approach
- **Issue** - MFENCE is executed for all field write instructions.

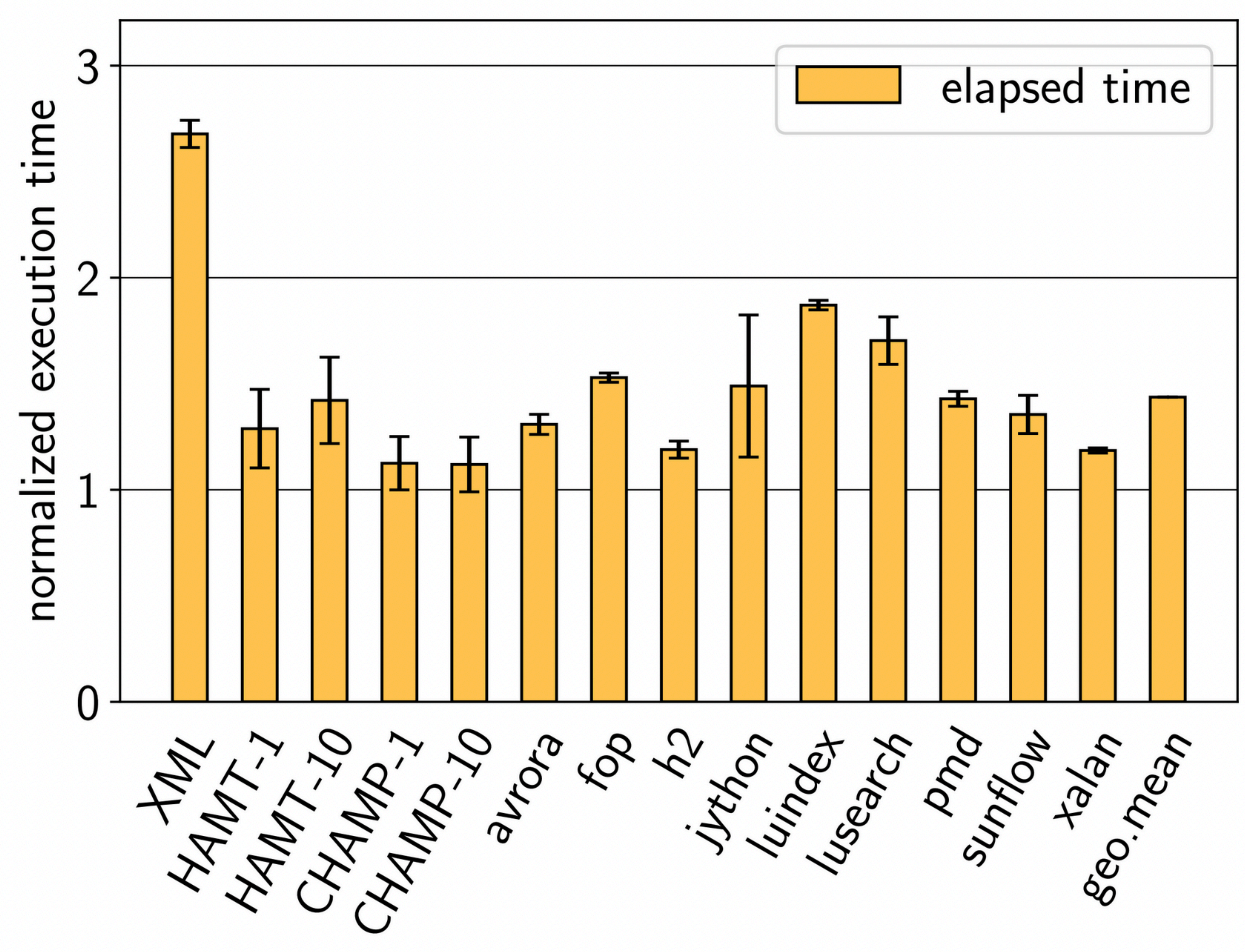
Reducing Write Barrier Overheads for Orthogonal Persistence

Write Barrier Overhead

- All putfield instructions will execute the write barrier
- Even if durable roots are absent, write barrier is executed.
- Increases execution time of the program

Write Barrier Overhead

- All putfield instructions will execute the write barrier
- Even if durable roots are absent, write barrier is executed.
- Increases execution time of the program
- Average overhead of 43.7% on the benchmarks in the absence of durable roots.



Elapsed times normalised to standard HotSpot VM

Reducing Write Barrier Overheads for Orthogonal Persistence

Intuition

- Frequency of copying << Frequency of writing
  - Shift the overhead to copier
- Copier thread performs a **Handshake** with all the threads and waits for acknowledgement
- Copier thread performs copy only when **Handshake** is acknowledged

Concurrent Access

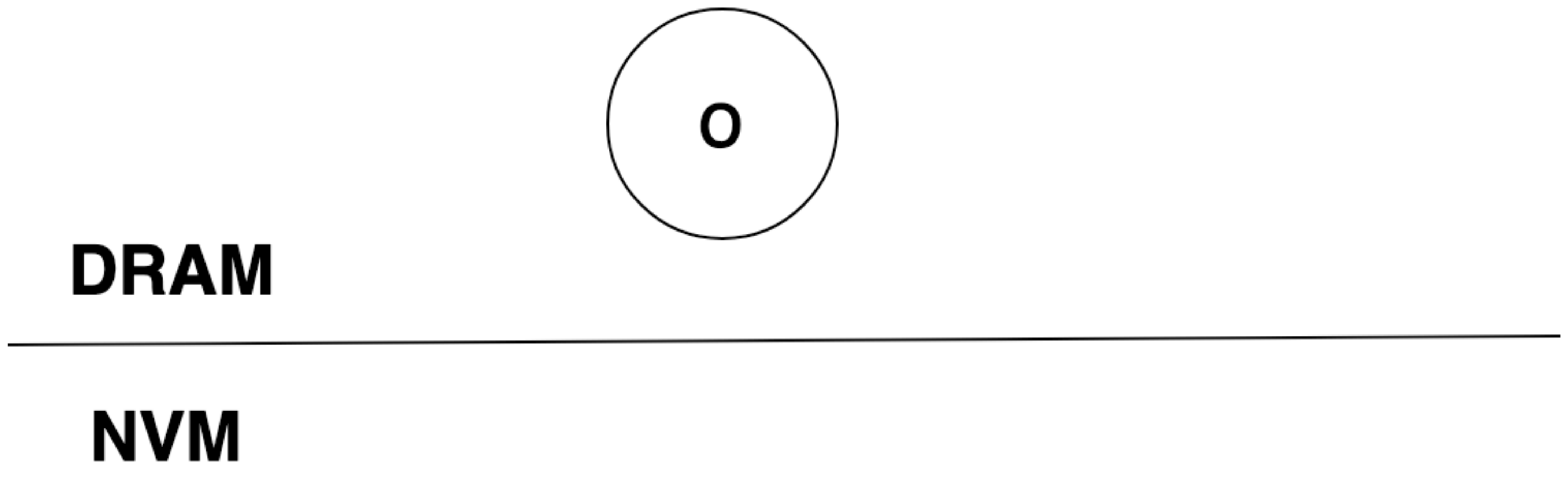
Case 3: Read at Line 4 is reordered with Write at Line 2

Copier Thread

durable_root = o

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o' = o.replica
5        if o' == NULL : return
6        make_persistent(v)
7        v' = v.replica
8        o'[f] = v'
9        CLWB(&o'[f])
```



Concurrent Access

Case 3: Read at Line 4 is reordered with Write at Line 2

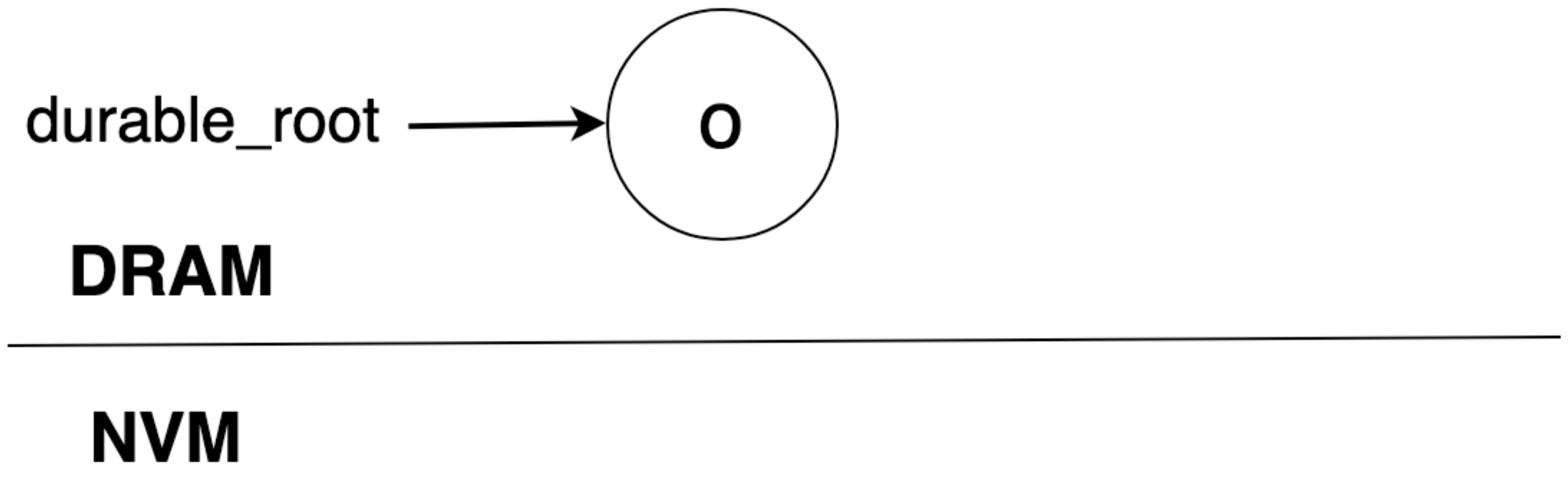
Copier Thread

durable_root = o

```
do:
  for x in reachable from o:
      x.replica = allocate_in_NVM()
  handshake()
  for x in reachable from o:
      copy x to x.replica
until all reachable from o are copied
```

Writer Thread (o.f = v)

```
1 def putfield( o, f, v ):
2       o[f] = v
3
4       o’ = o.replica
5       if o’ == NULL : return
6       make_persistent(v)
7       v’ = v.replica
8       o’[f] = v’
9       CLWB(&o’[f])
```



Concurrent Access

Case 3: Read at Line 4 is reordered with Write at Line 2

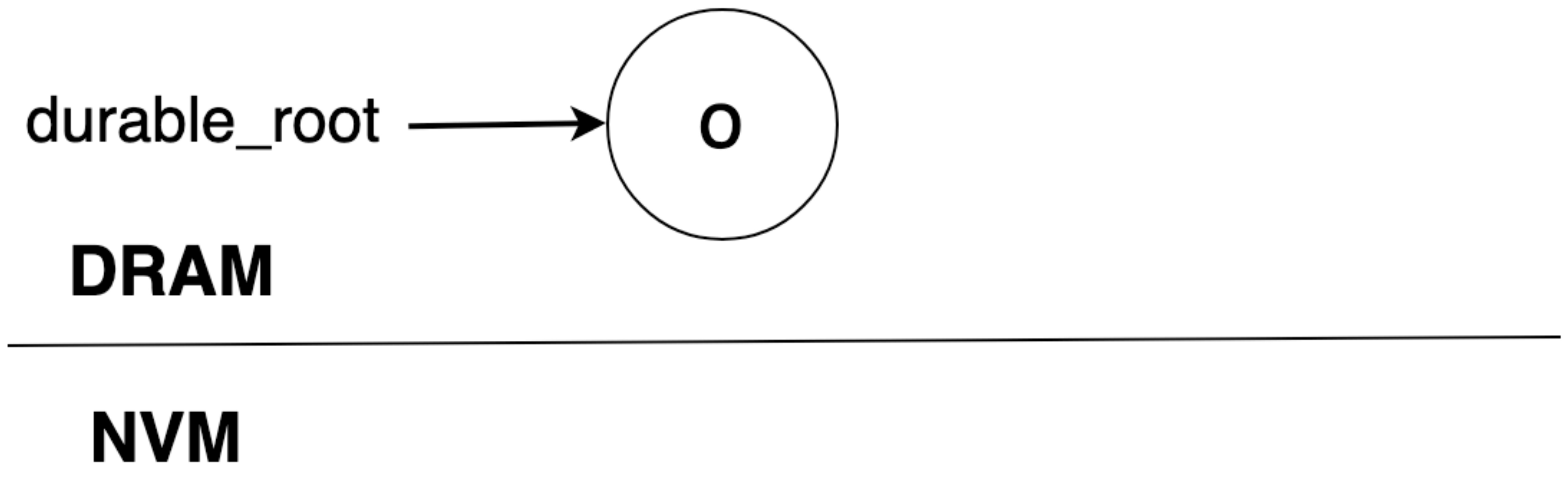
Copier Thread

durable_root = o

```
do:
  for x in reachable from o:
      x.replica = allocate_in_NVM()
  handshake()
  for x in reachable from o:
      copy x to x.replica
until all reachable from o are copied
```

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o' = o.replica
5        if o' == NULL : return
6        make_persistent(v)
7        v' = v.replica
8        o'[f] = v'
9        CLWB(&o'[f])
```



Concurrent Access

Case 3: Read at Line 4 is reordered with Write at Line 2

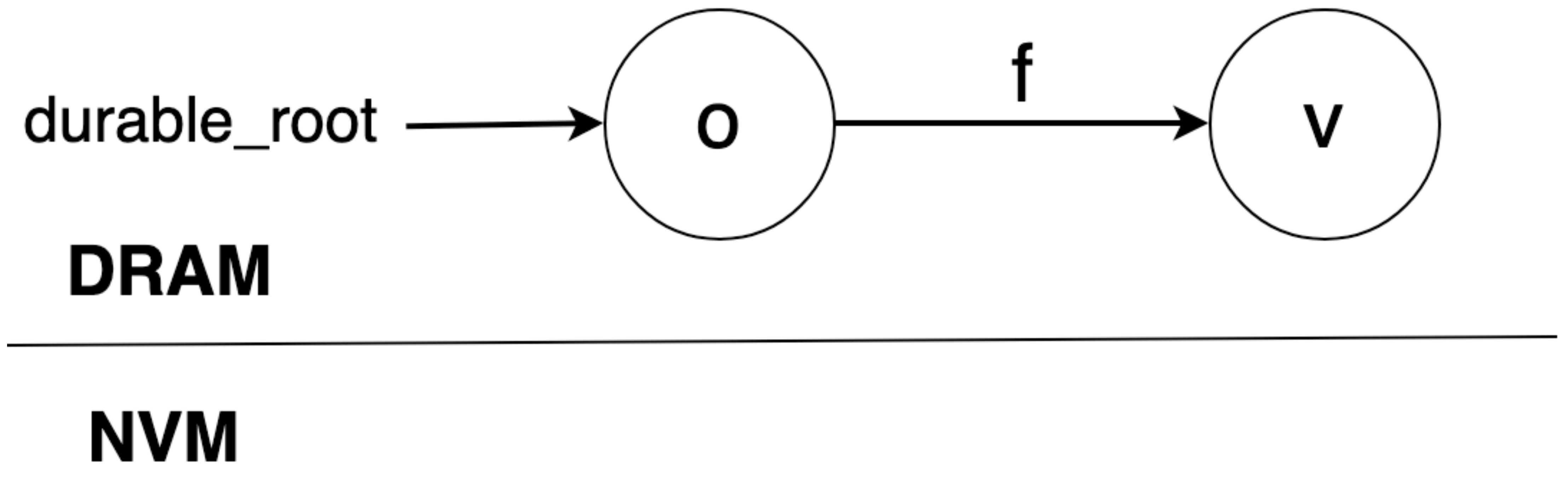
Copier Thread

durable_root = o

```
// handshake
// copy
```

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2      o[f] = v
3
4      o' = o.replica
5      if o' == NULL : return
6      make_persistent(v)
7      v' = v.replica
8      o'[f] = v'
9      CLWB(&o'[f])
```



Concurrent Access

Case 3: Read at Line 4 is reordered with Write at Line 2

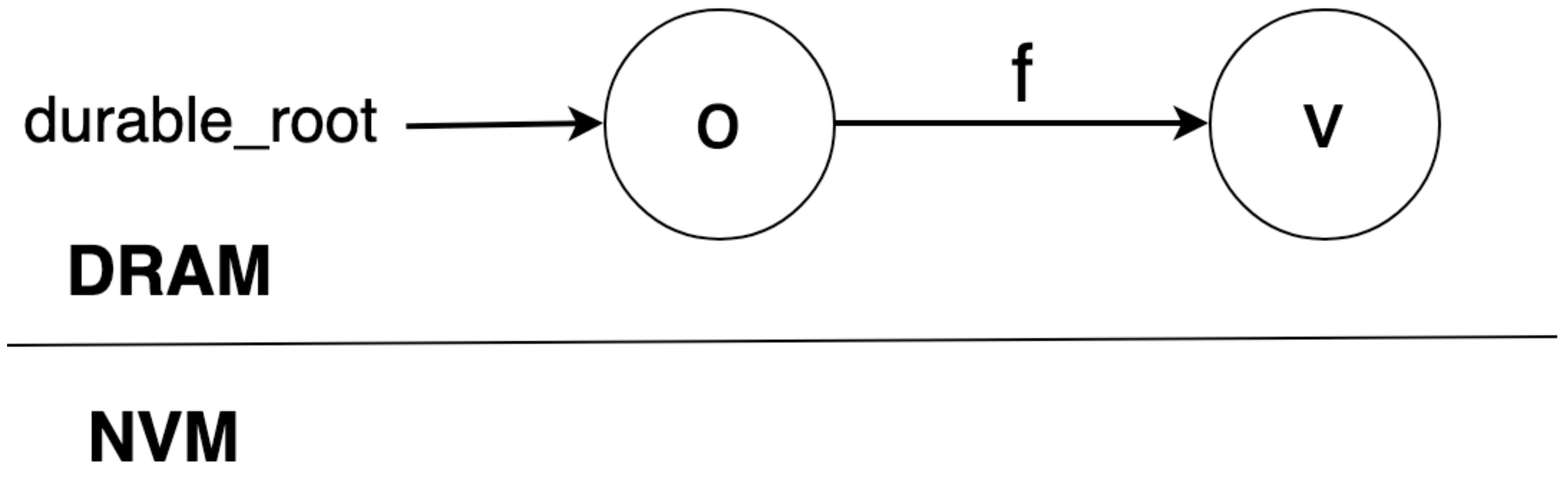
Copier Thread

durable_root = o

// handshake
// copy

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o’ = o.replica
5        if o’ == NULL : return
6        make_persistent(v)
7        v’ = v.replica
8        o’[f] = v’
9        CLWB(&o’[f])
```



Concurrent Access

Case 3: Read at Line 4 is reordered with Write at Line 2

Copier Thread

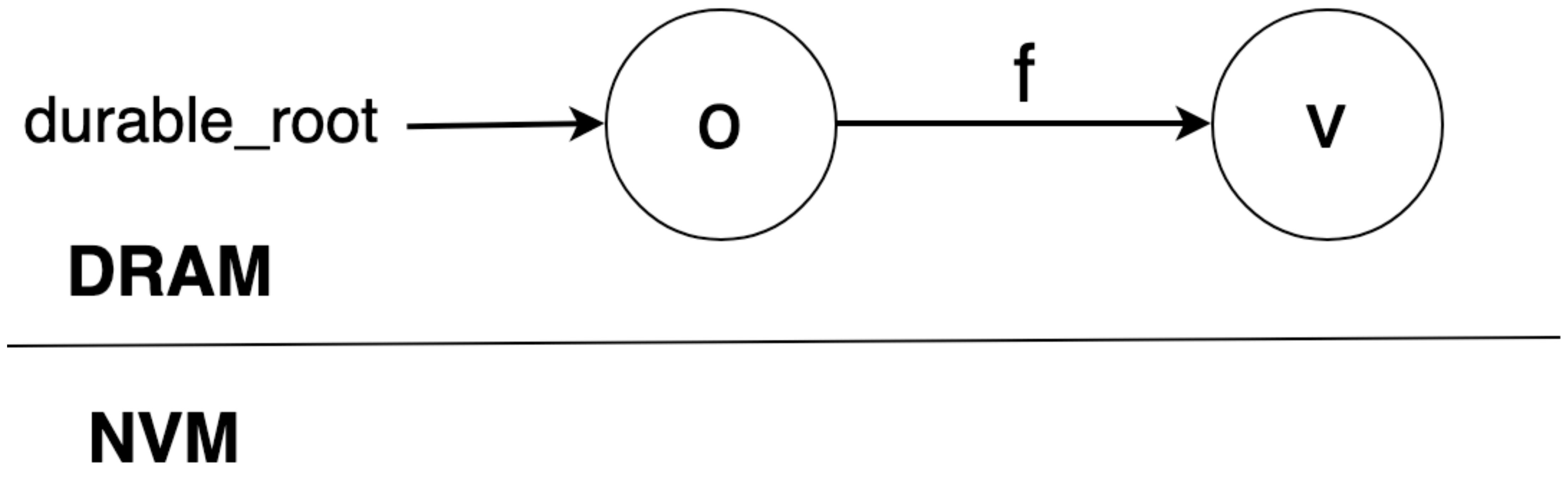
durable_root = o

```
do:
  for x in reachable from o:
      x.replica = allocate_in_NVM()
  handshake()
  for x in reachable from o:
      copy x to x.replica
until all reachable from o are copied
```

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o' = o.replica
5        if o' == NULL : return
6        make_persistent(v)
7        v' = v.replica
8        o'[f] = v'
9        CLWB(&o'[f])
```

GC safepoint



Concurrent Access

Case 3: Read at Line 4 is reordered with Write at Line 2

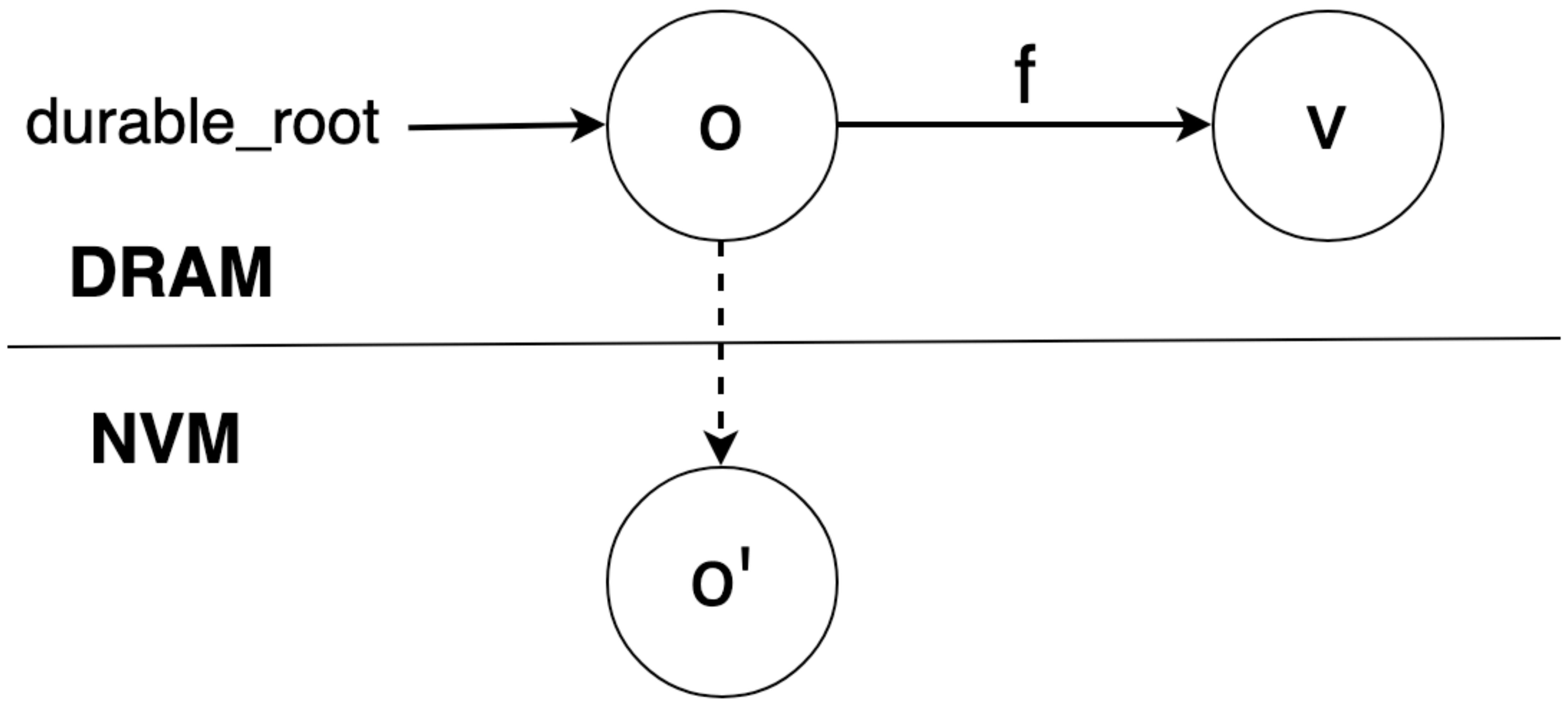
Copier Thread

durable_root = o

```
do:
  for x in reachable from o:
      x.replica = allocate_in_NVM()
  handshake()
  for x in reachable from o:
      copy x to x.replica
until all reachable from o are copied
```

Writer Thread (o.f = v)

```
1  def putfield( o, f, v ):
2        o[f] = v
3
4        o’ = o.replica
5        if o’ == NULL : return
6        make_persistent(v)
7        v’ = v.replica
8        o’[f] = v’
9        CLWB(&o’[f])
```



Concurrent Access

Case 3: Read at Line 4 is reordered with Write at Line 2

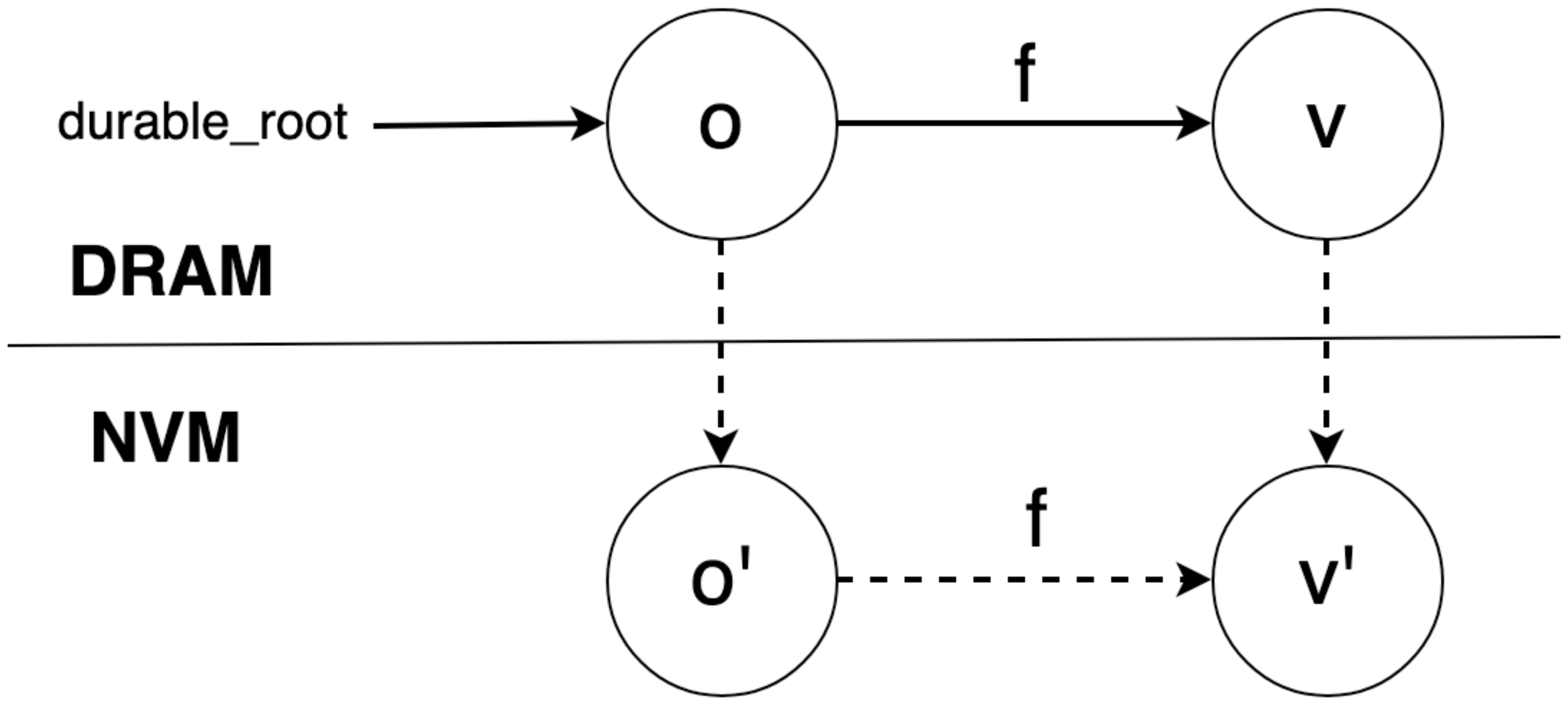
Copier Thread

durable_root = o

```
do:
  for x in reachable from o:
      x.replica = allocate_in_NVM()
  handshake()
  for x in reachable from o:
      copy x to x.replica
until all reachable from o are copied
```

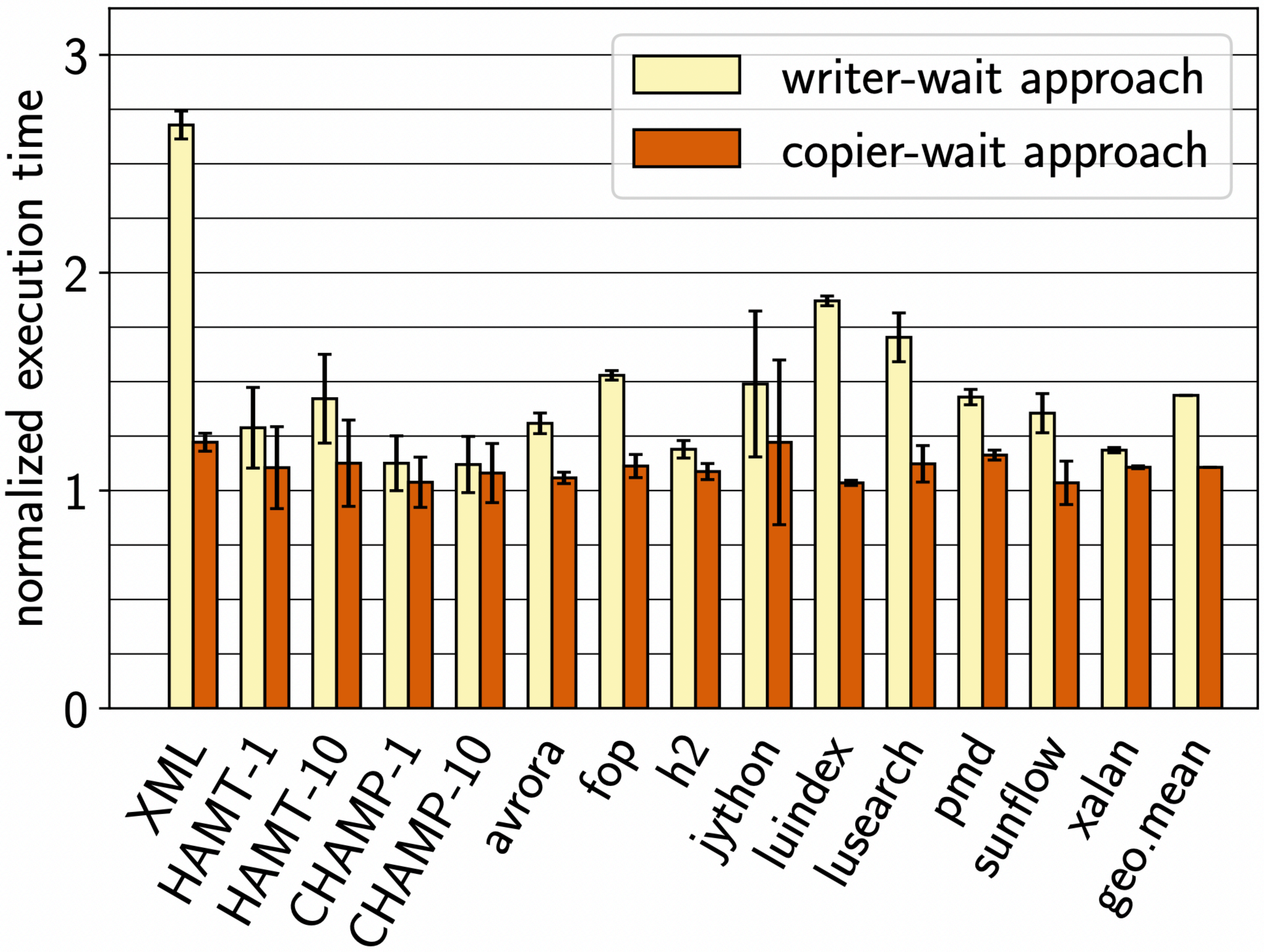
Writer Thread (o.f = v)

```
1 def putfield( o, f, v ):
2       o[f] = v
3
4       o' = o.replica
5       if o' == NULL : return
6       make_persistent(v)
7       v' = v.replica
8       o'[f] = v'
9       CLWB(&o'[f])
```



Does shifting overhead to the copier benefit programs that rarely make objects persistent ?

No durable roots in program



Elapsed times normalised to standard HotSpot VM

writer-wait approach
43.7% overhead on average

copier-wait approach
10.6% overhead on average

What about programs that frequently make objects persistent ?

Overheads of Copier-wait

- When objects are frequently made persistent, copier-wait approach has high overheads

- Handshake overhead of 37.9 % compared to writer-wait approach when all static fields are annotated as durable roots.

Overheads of Copier-wait

- When objects are frequently made persistent, copier-wait approach has high overheads
- Handshake overhead of 37.9 % compared to writer-wait approach when all static fields are annotated as durable roots.
- Do we always need handshake ?

Do we always need handshake ?

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
    o[f] = v

    o' = o.replica
    if o' == NULL : return
    make_persistent(v)
    v' = v.replica
    o'[f] = v'
    CLWB(&o'[f])
```

```
do:
  for x in reachable from o:
      x.replica = allocate_in_NVM()
  handshake()
  for x in reachable from o:
      copy x to x.replica
until all reachable from o are copied
```

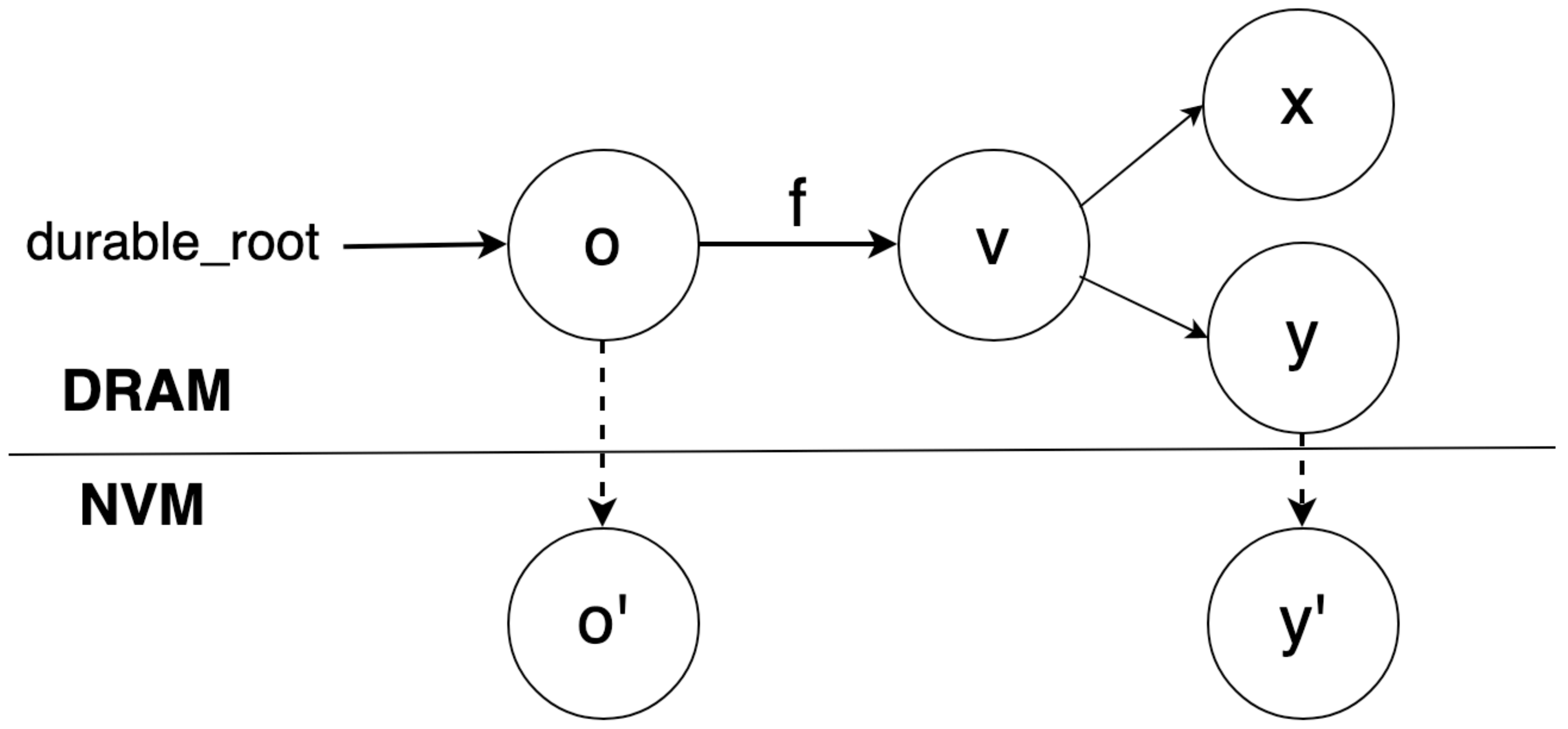
Do we always need handshake ?

Writer Thread (o.f = v)

```
def putfield( o, f, v ):
    o[f] = v

    o' = o.replica
    if o' == NULL : return
    make_persistent(v)
    v' = v.replica
    o'[f] = v'
    CLWB(&o'[f])
```

```
do:
  for x in reachable from o:
      x.replica = allocate_in_NVM()
  handshake()
  for x in reachable from o:
      copy x to x.replica
until all reachable from o are copied
```



If all the objects reachable from v (including) are, either

- Thread Local or
- Persistent

Then handshake can be elided

Persistence-Aware Escape Analysis

- Combined Points-to-Escape analysis is used to identify thread-local abstract objects
- Modify escape analysis to recognise a special abstract object
  - Persistent Object (**P**)

Persistence-Aware Escape Analysis

- Combined Points-to-Escape analysis is used to identify thread-local abstract objects
- Modify escape analysis to recognise a special abstract object
  - Persistent Object (**P**)
- Escape Analysis

x = A.durable_root	x -> { **E** }
y = x.f	y -> { **E** }
y.f = z	y.f -> { **E** }
t = new Thread(w)	t -> { **E** } w -> { **E** }

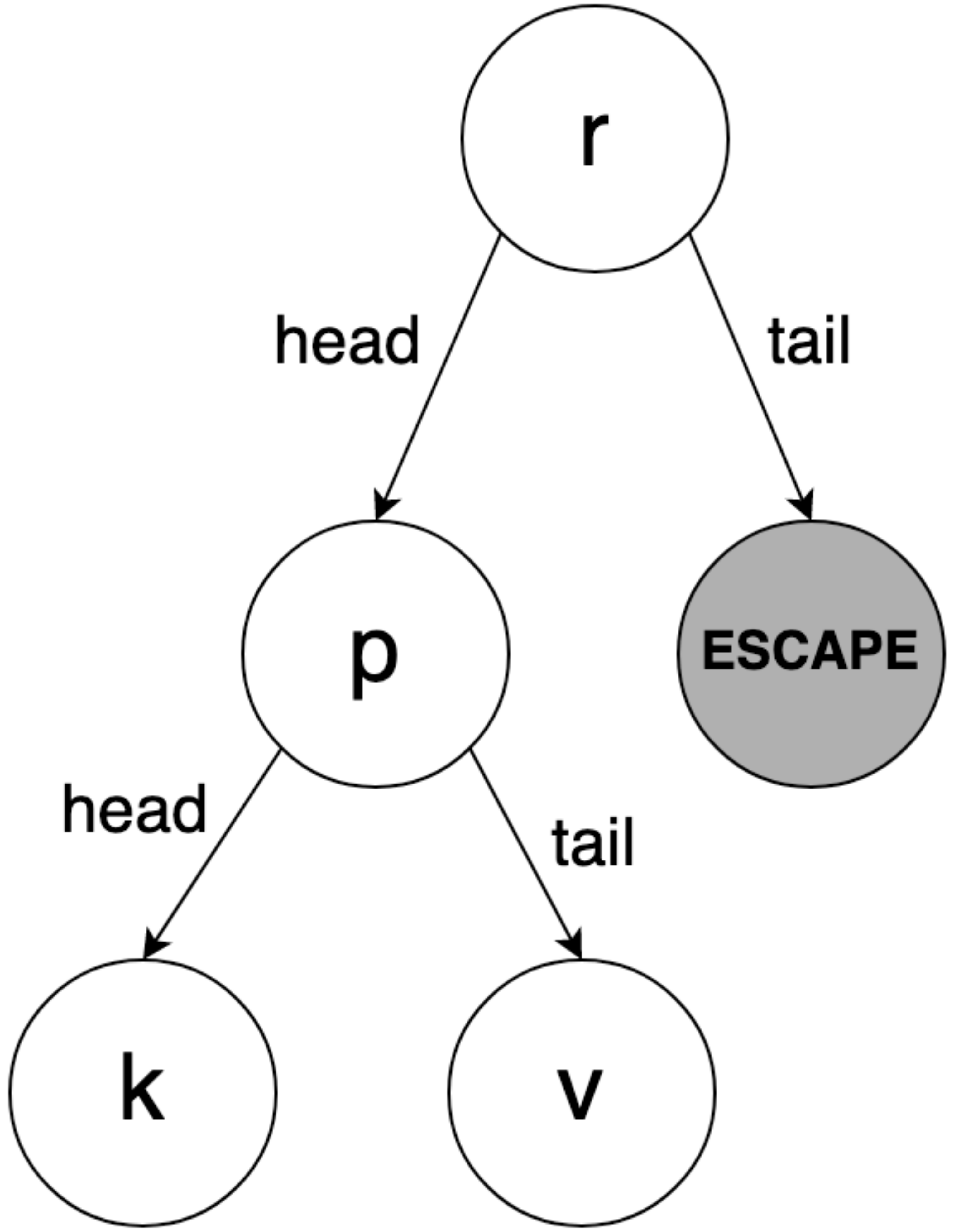
Persistence-Aware Escape Analysis

- Combined Points-to-Escape analysis is used to identify thread-local abstract objects
- Modify escape analysis to recognise a special abstract object
  - Persistent Object (**P**)
- Persistence-Aware Escape Analysis

x = A.durable_root	x -> { **P** }
y = x.f	y -> { **P** }
y.f = z	y.f -> { **P** }
t = new Thread(w)	t -> { **E** } w -> { **E** }

Example

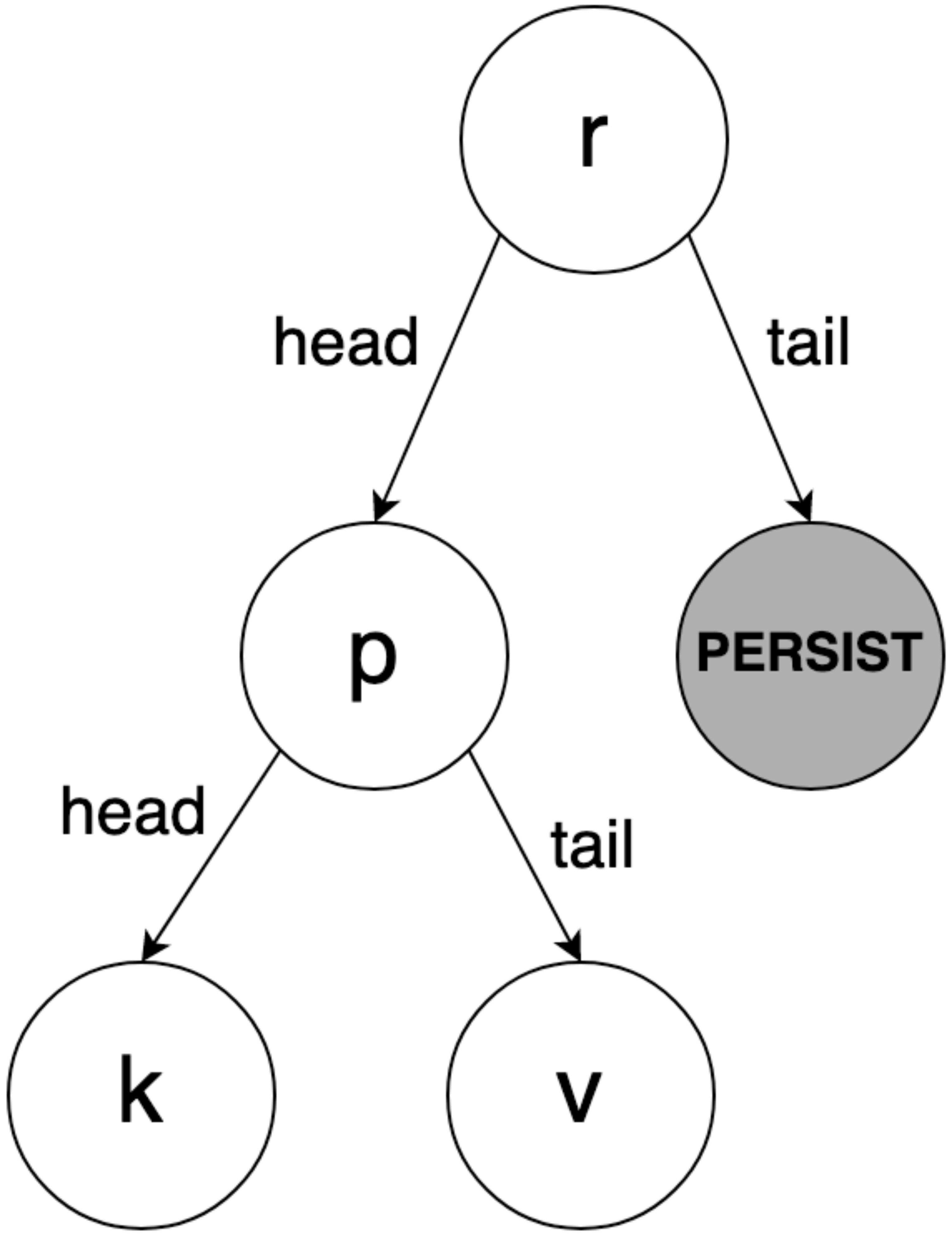
```java
class Program {
  @durable_root static Pair assocList;
  @durable_root static int count;
  static void assoc(Object k, Object v) {
    Pair p = new Pair(); // p is volatile object
    p.head = k;
    p.tail = v;
    Pair r = new Pair(); // r is volatile object
    r.head = p;
    r.tail = Program.assocList;
    Program.assocList = r; // p and r become persistent
    Program.count = Program.count + 1;   } }
```



White objects are thread-local objects

Example

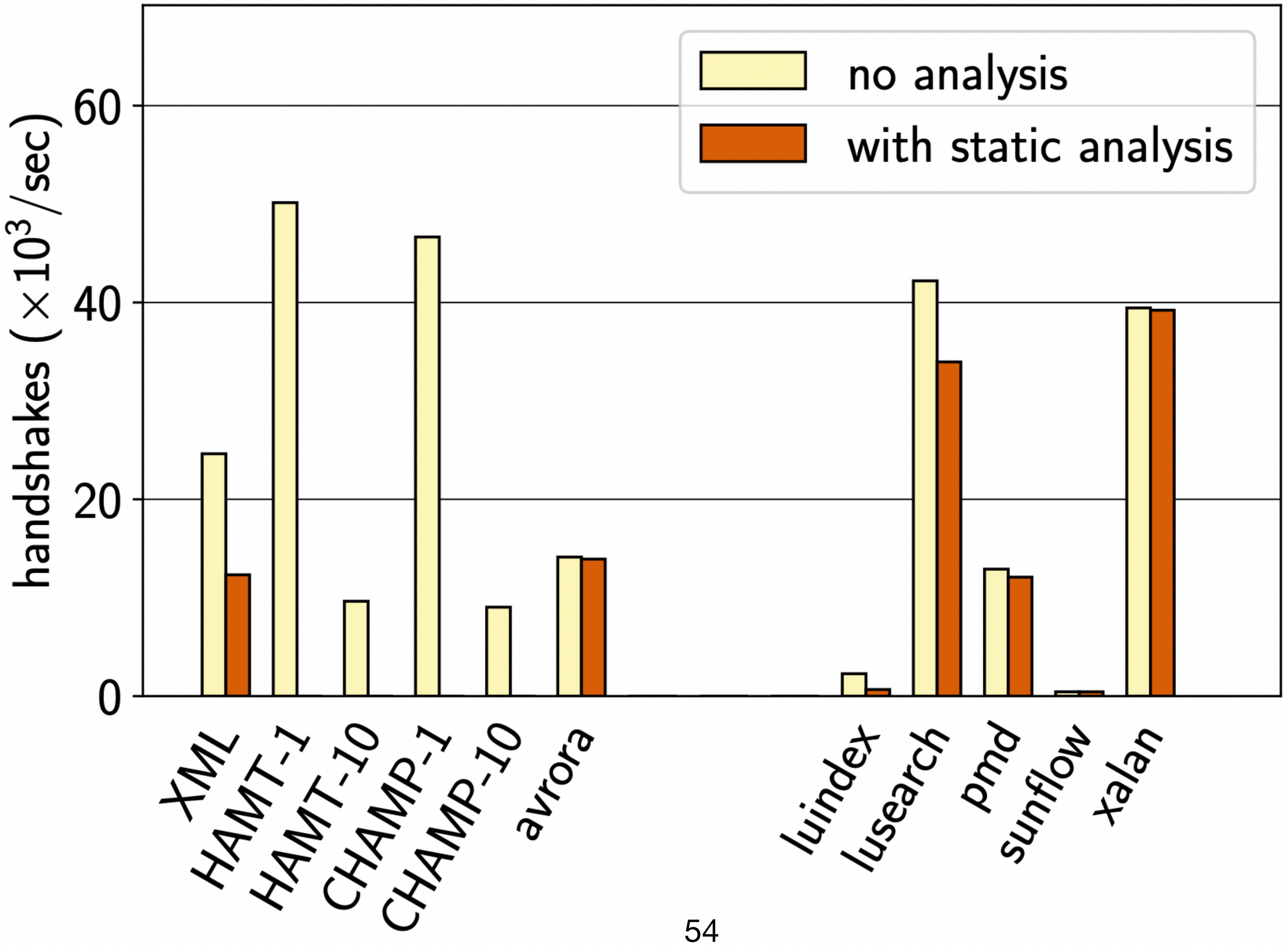
```
class Program {
  @durable_root static Pair assocList;
  @durable_root static int count;
  static void assoc(Object k, Object v) {
    Pair p = new Pair(); // p is volatile object
    p.head = k;
    p.tail = v;
    Pair r = new Pair(); // r is volatile object
    r.head = p;
    r.tail = Program.assocList;
    Program.assocList = r; // p and r become persistent
    Program.count = Program.count + 1;  } }
```



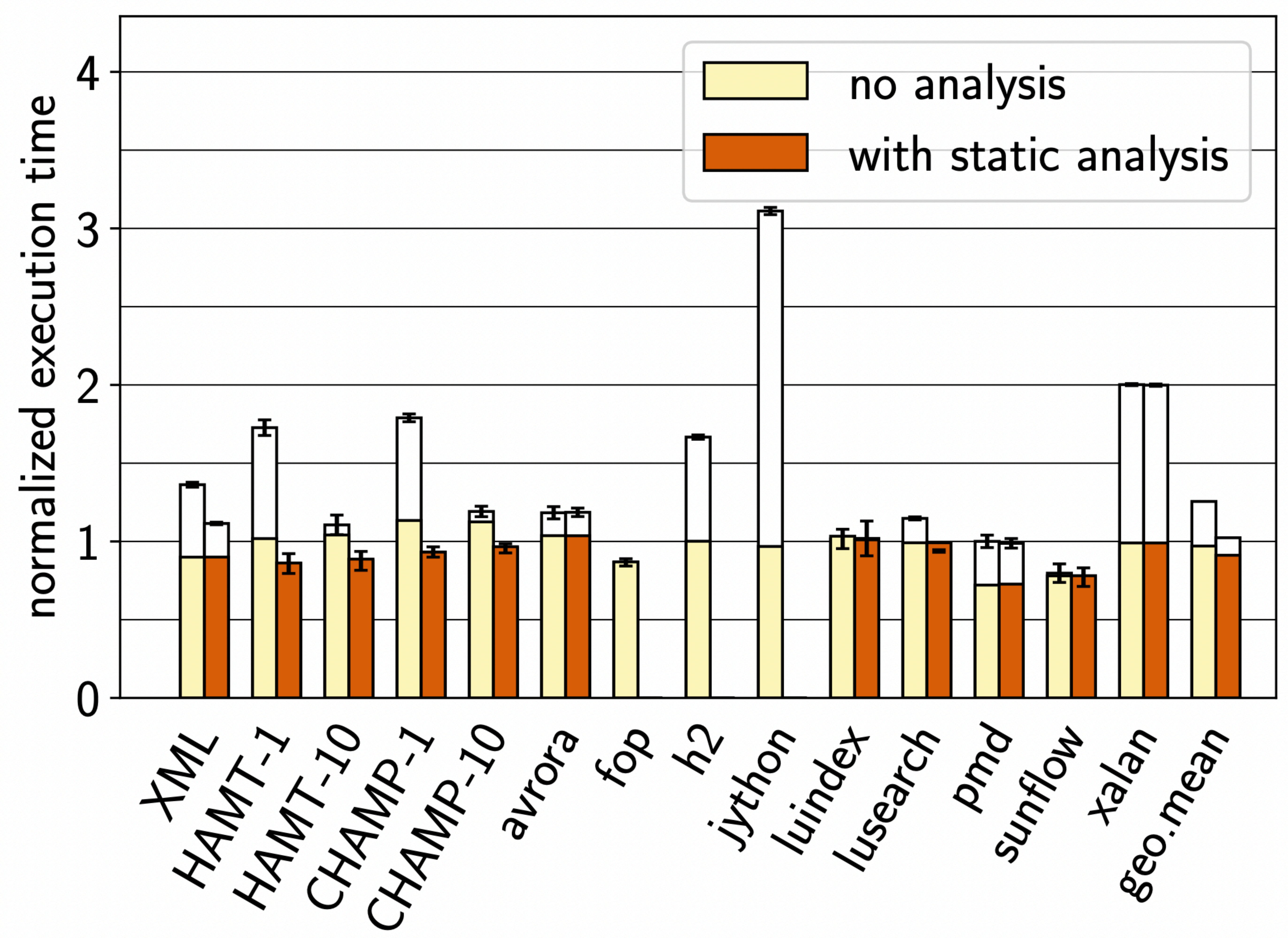
White objects are thread-local objects

Was static analysis successful in eliminating handshakes ?

Number of handshakes per second



Execution time



Execution time normalised to writer-wait approach

Static analysis eliminated 52% overheads on average

Copier-wait slower by 2.4% on average

Related Work

- AutoPersist [Shull et al. 2019] uses copier flag that is accessed atomically. Copier fails when race is detected
- QuickCheck [Shull et al. 2019] and P-INSPECT [Kokolis et al. 2020] try to reduce the write barrier overheads but fail to handle races.
- StaticPersist [Bansal 2023] uses static analysis to verify if programmer has correctly made the objects persistent (No pointer from persistent object to volatile object)

Covered in the paper

- Explanation of how the transitive closure of an object is copied.
- Handle race between copier and writer [Ragged Synchronization]
- Synchronization for multiple copiers trying to copy same object
- Correctness Argument
- Additional overheads eliminated by Persistence-aware Escape analysis
- Flow functions for Persistence-aware Escape analysis

Conclusion

- Shifted the overheads from writer thread to copier thread
- 23% performance improvement on average for programs that rarely make objects persistent
- Static analysis decreases 52% of persistence-related overheads in the copier-wait approach
- Performance of copier-wait approach comparable (better in some case) to writer-wait approach

Conclusion

- Shifted the overheads from writer thread to copier thread
- 23% performance improvement on average for programs that rarely make objects persistent
- Static analysis decreases 52% of persistence-related overheads in the copier-wait approach
- Performance of copier-wait approach comparable (better in some case) to writer-wait approach

Thank You