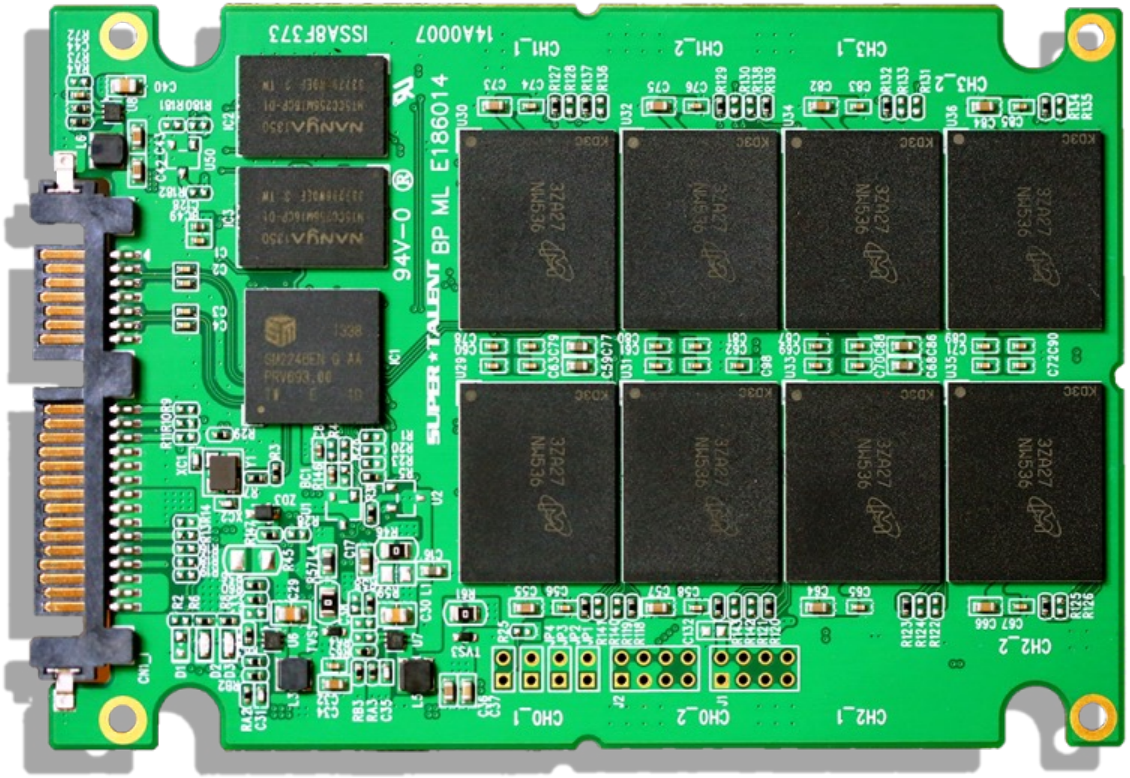
ACEing the Bufferpool Management Paradigm for Modern Storage Devices

Tarikul Islam Papon Manos Athanassoulis

Solid State Drives



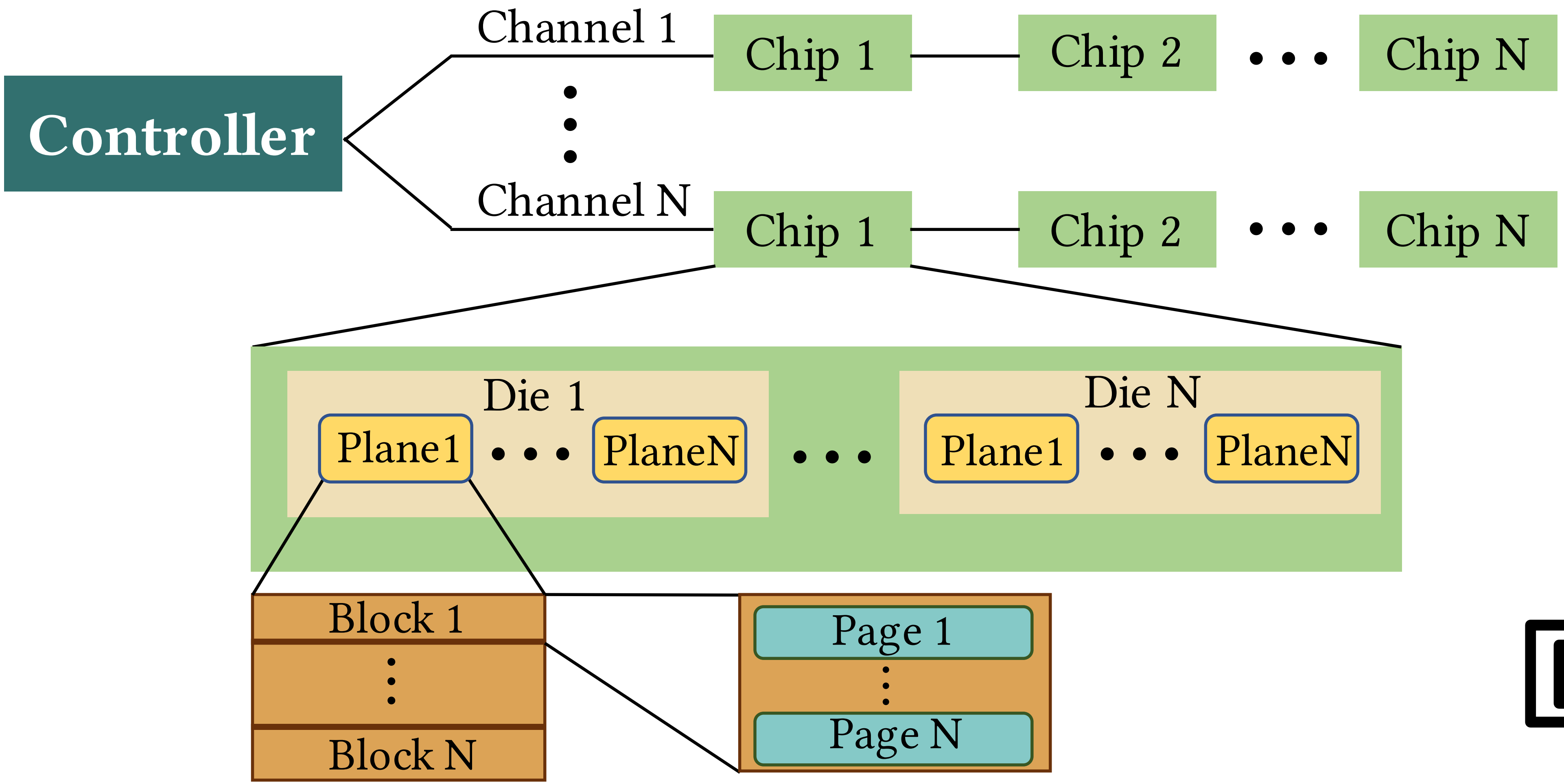
electronic device

fast random access

concurrent I/Os

write latency > read latency

Concurrency


Controller

Channel 1
Chip 1 — Chip 2 • • • Chip N

⋮

Channel N
Chip 1 — Chip 2 • • • Chip N

Die 1
Plane1 • • • PlaneN

• • •

Die N
Plane1 • • • PlaneN

Block 1
⋮
Block N

Page 1
⋮
Page N

Parallelism at different levels (channel, chip, die, plane block, page)

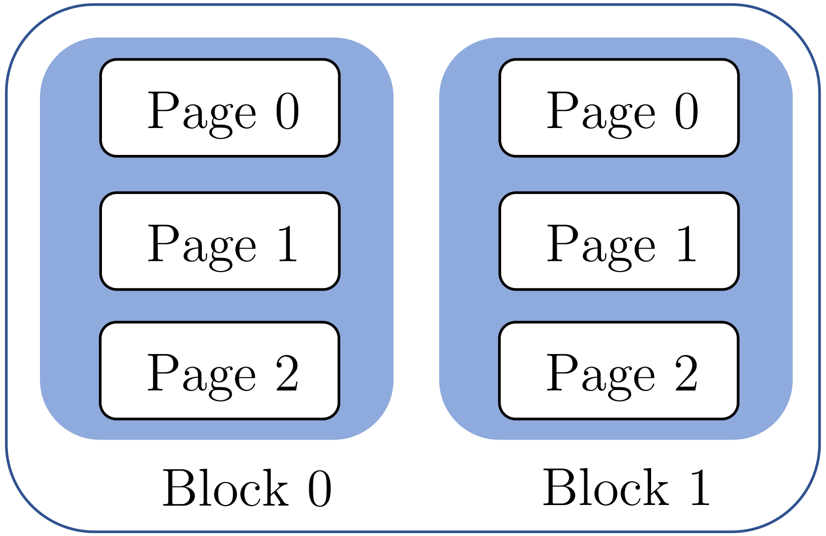
Read/Write Asymmetry

Out-of-place updates cause invalidation

“*Erase before write*” approach

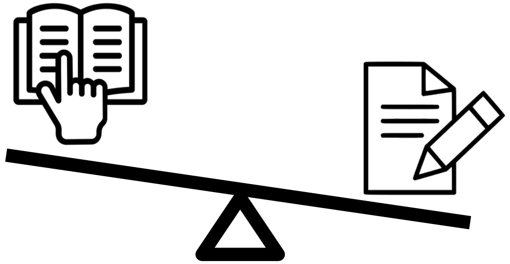
Garbage Collection

Larger erase granularity

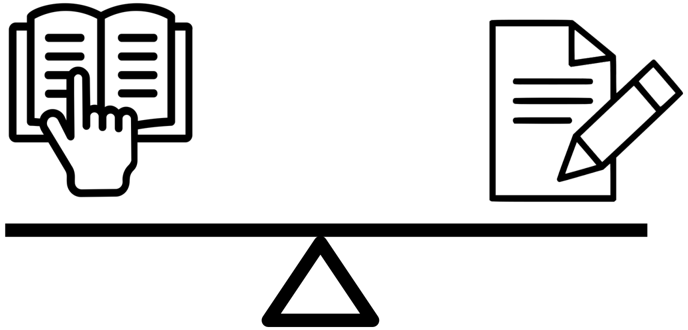


Plane

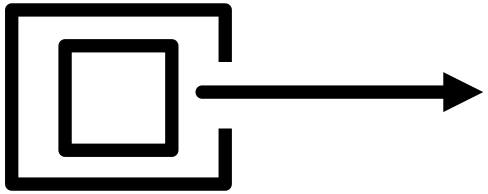
All these results in higher amortized write cost



HDD

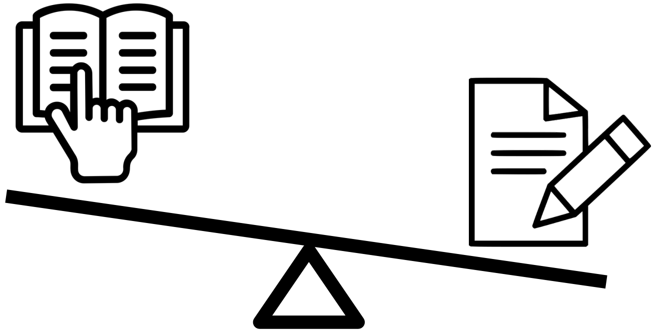


Symmetric cost for Read & Write

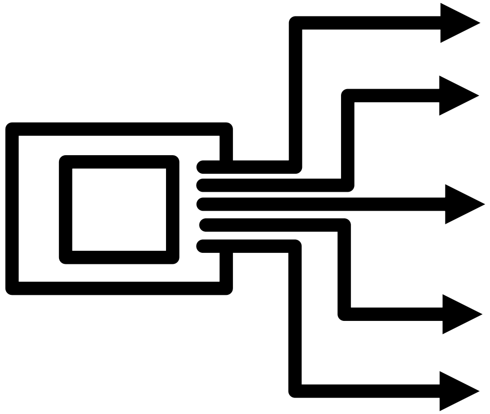


One I/O at a time

SSD



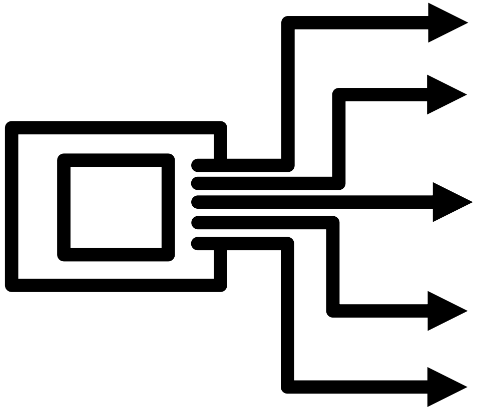
Read/Write Asymmetry (α)

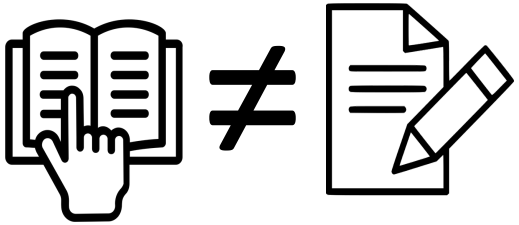


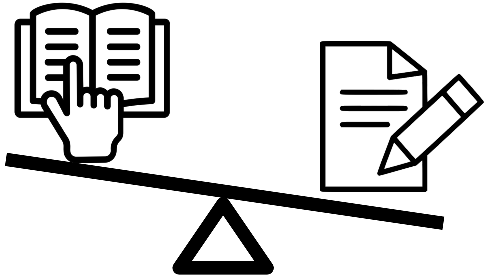
Concurrency (k)

"A Parametric I/O Model for Modern Storage Devices"

DaMoN 2021







know Thy Device

exploit k_r and k_w (with care)

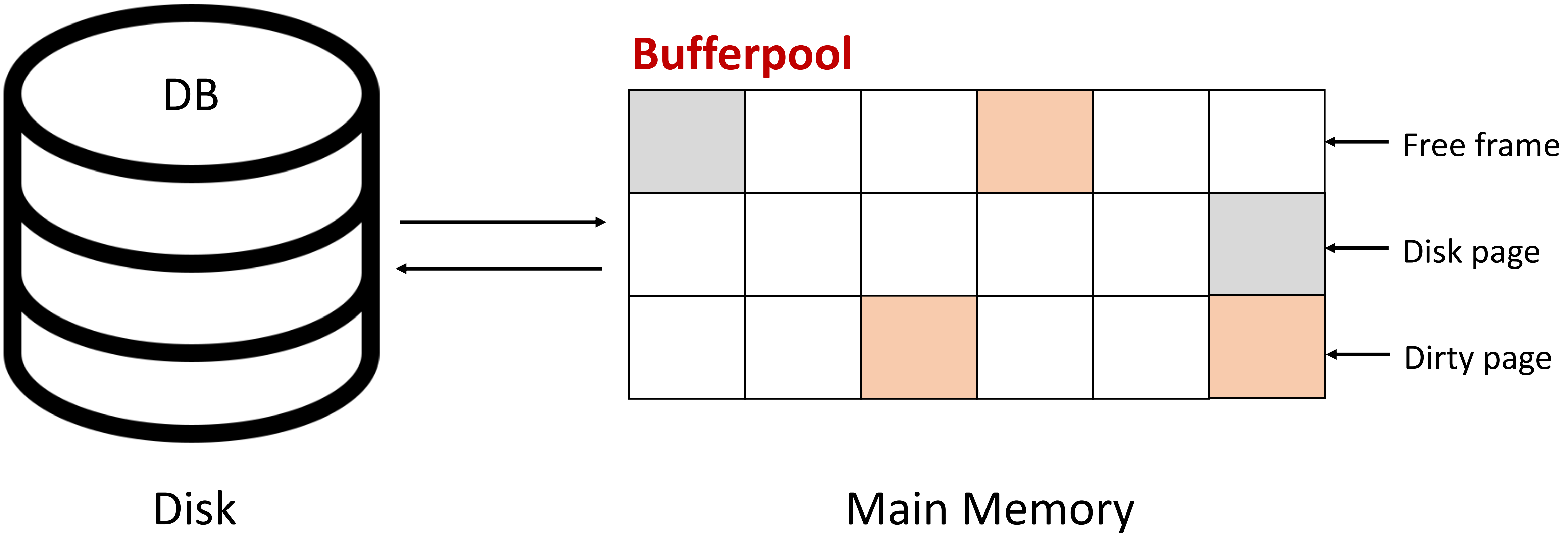
read concurrency

write concurrency

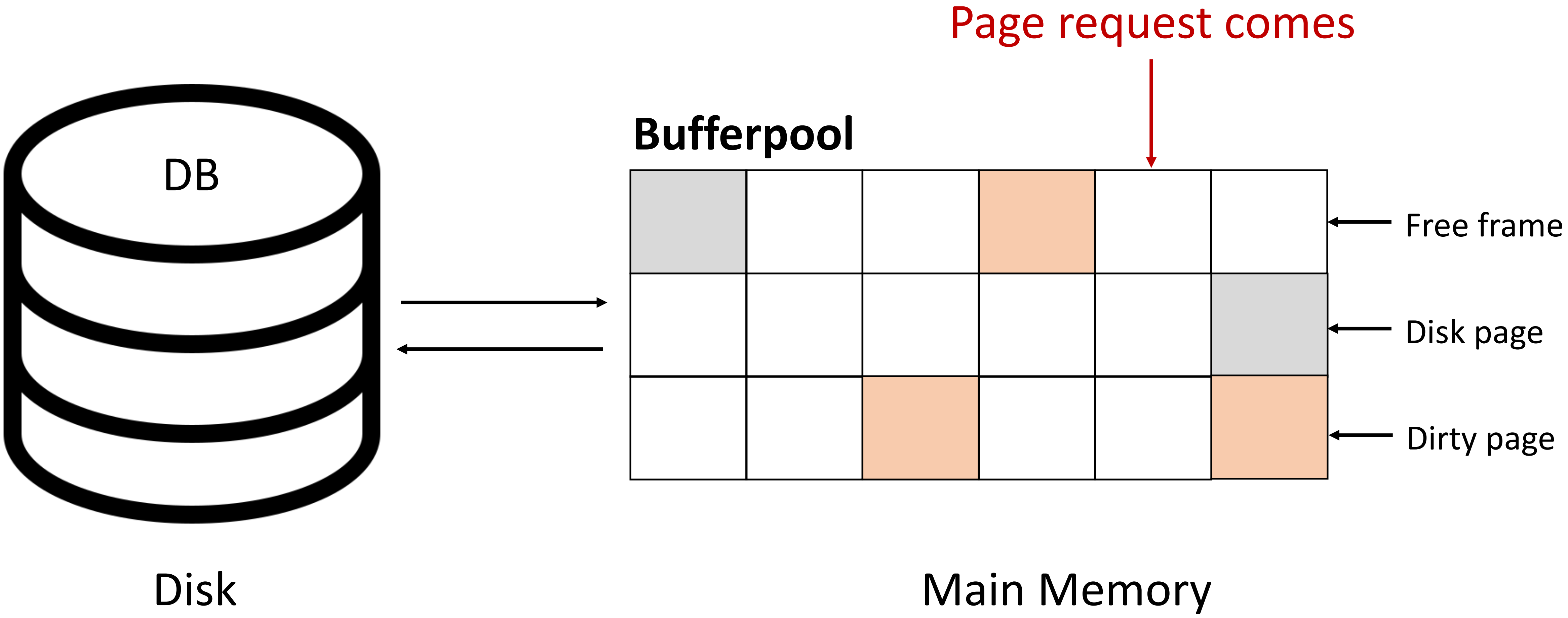
treat read and write differently

asymmetry (α) controls performance

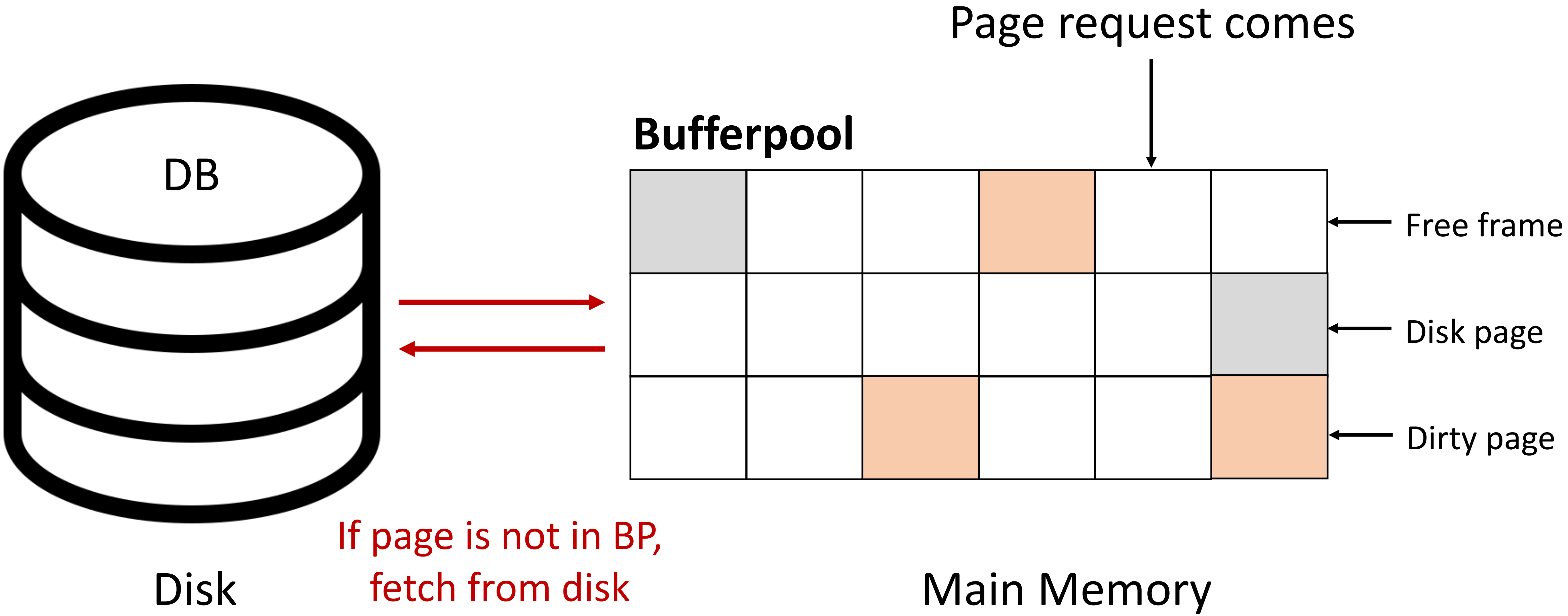
Bufferpool is Tightly Connected to Storage



Traditional Bufferpool Manager

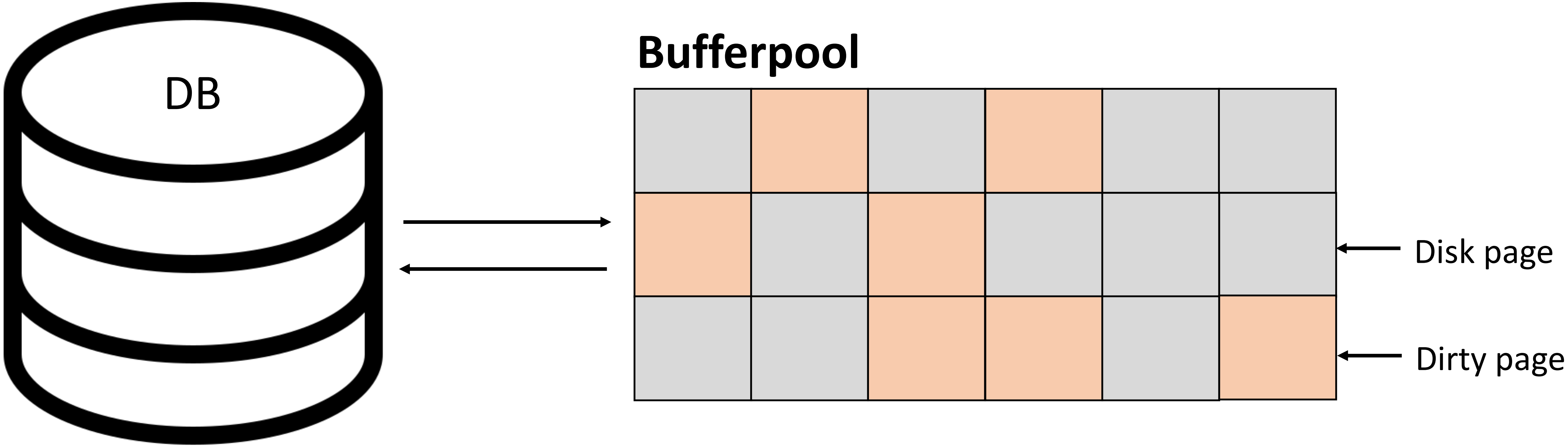


Traditional Bufferpool Manager

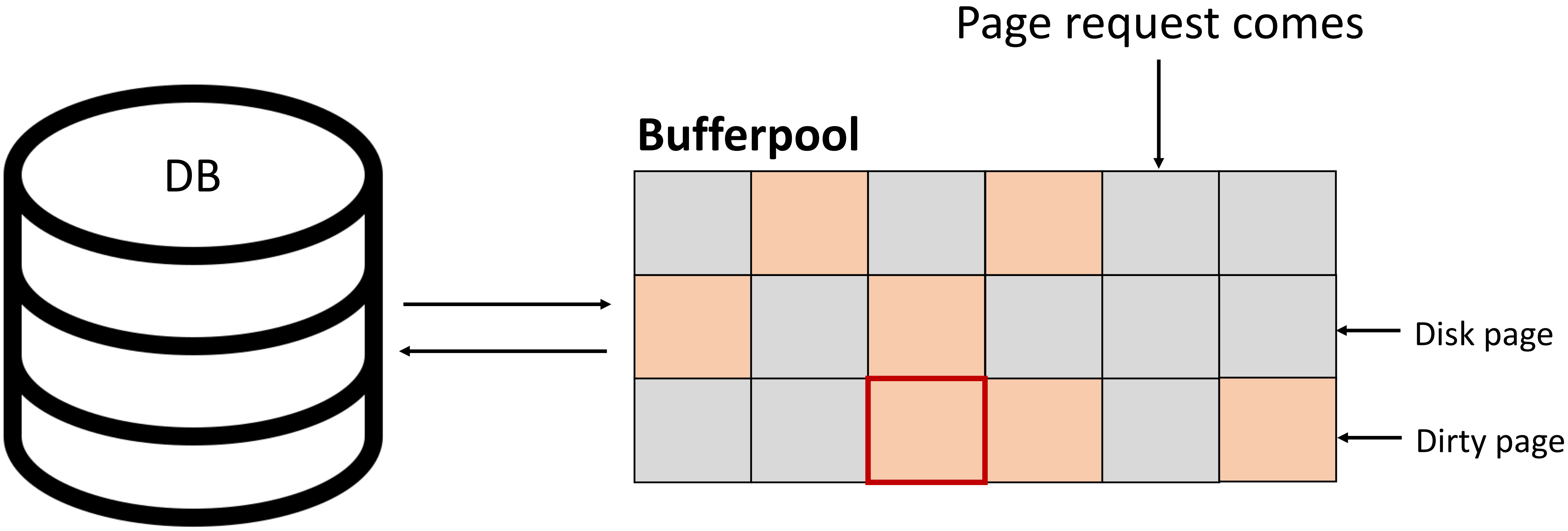


If page is not in BP, fetch from disk

Traditional Bufferpool Manager

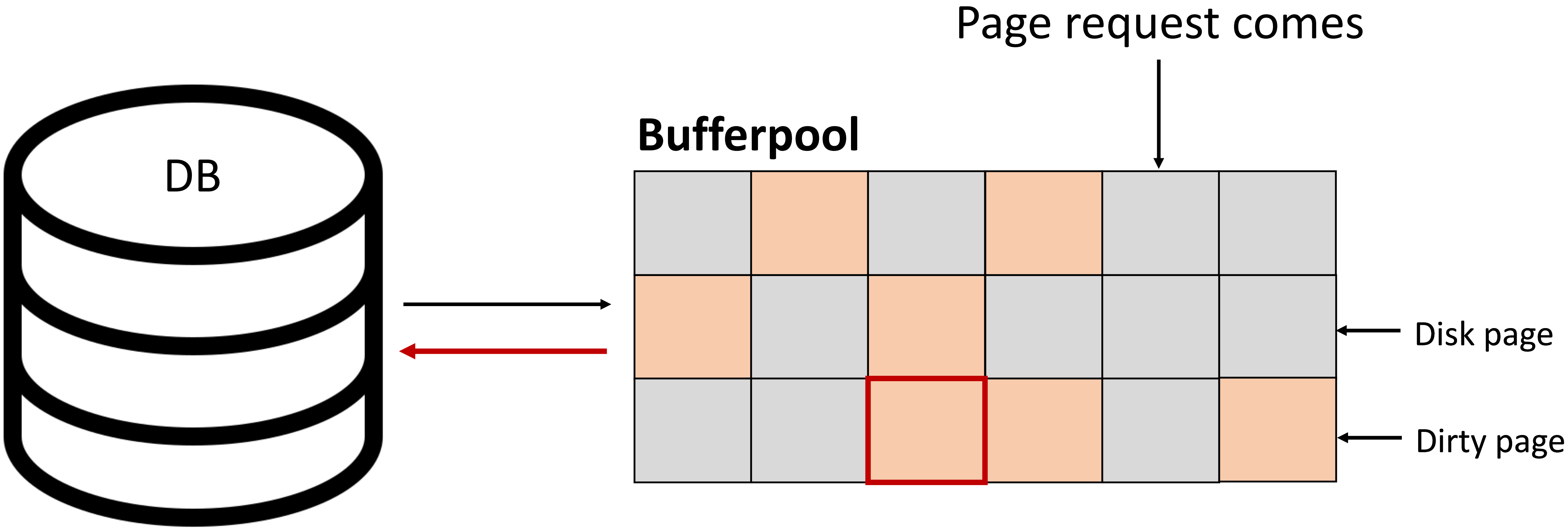


Traditional Bufferpool Manager



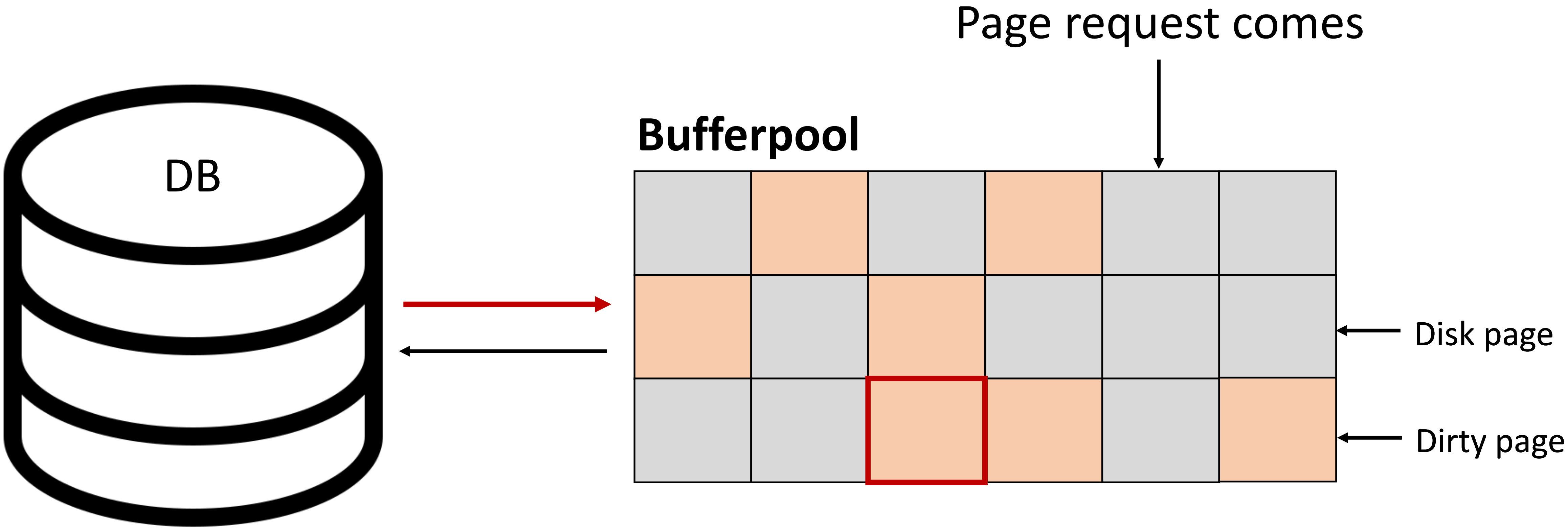
If BP is full, one page is selected for eviction based on **page replacement policy**

Traditional Bufferpool Manager



If the page is dirty, it is written back to disk

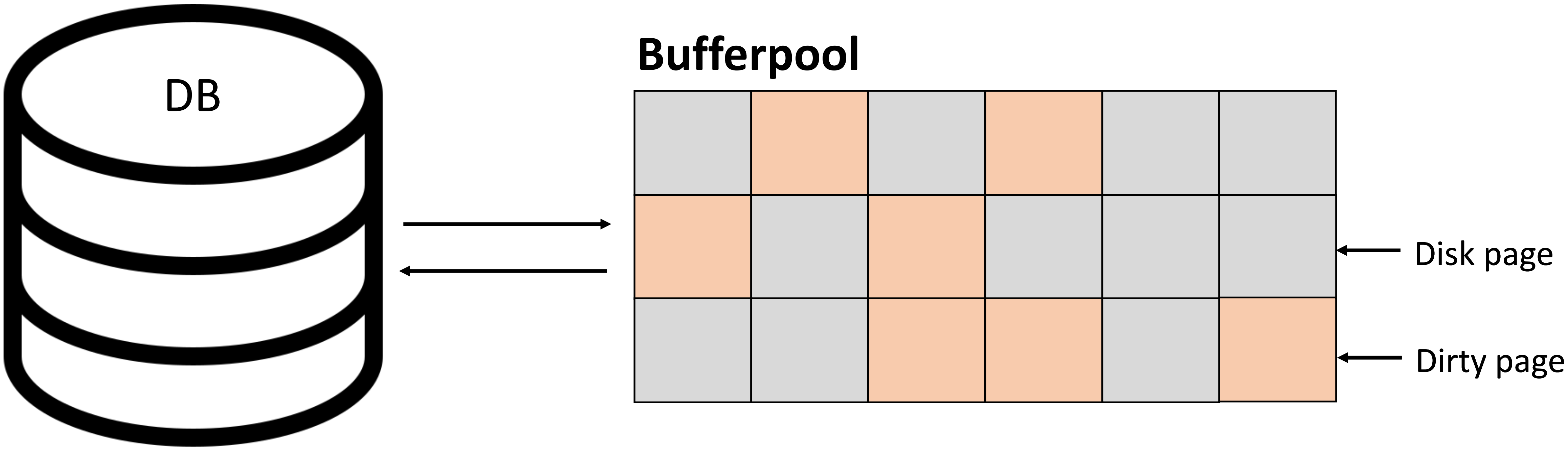
Traditional Bufferpool Manager



Requested page is fetched in its place

(exchanging one write for a read)

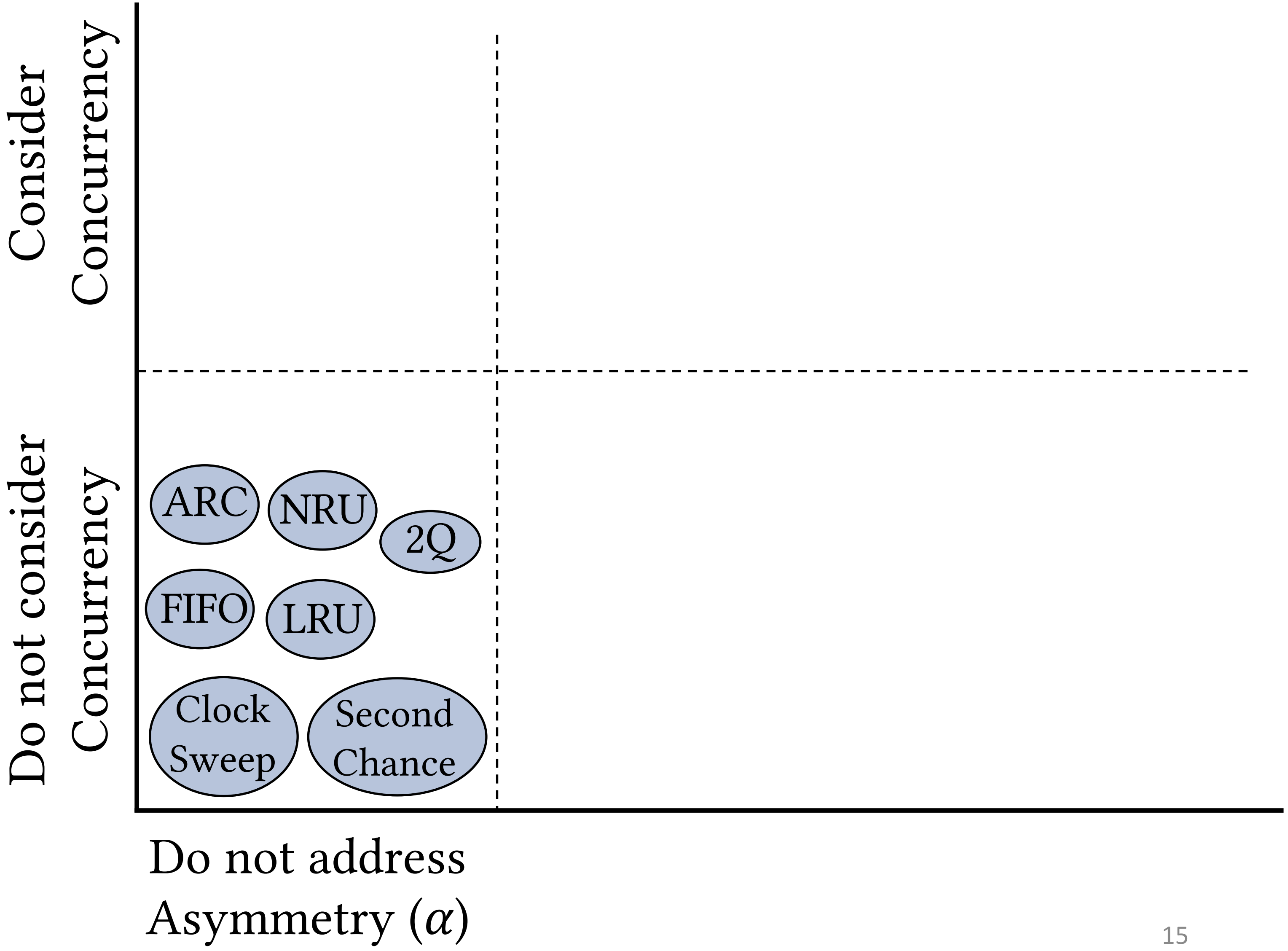
Traditional Bufferpool Manager



Is this Optimal?

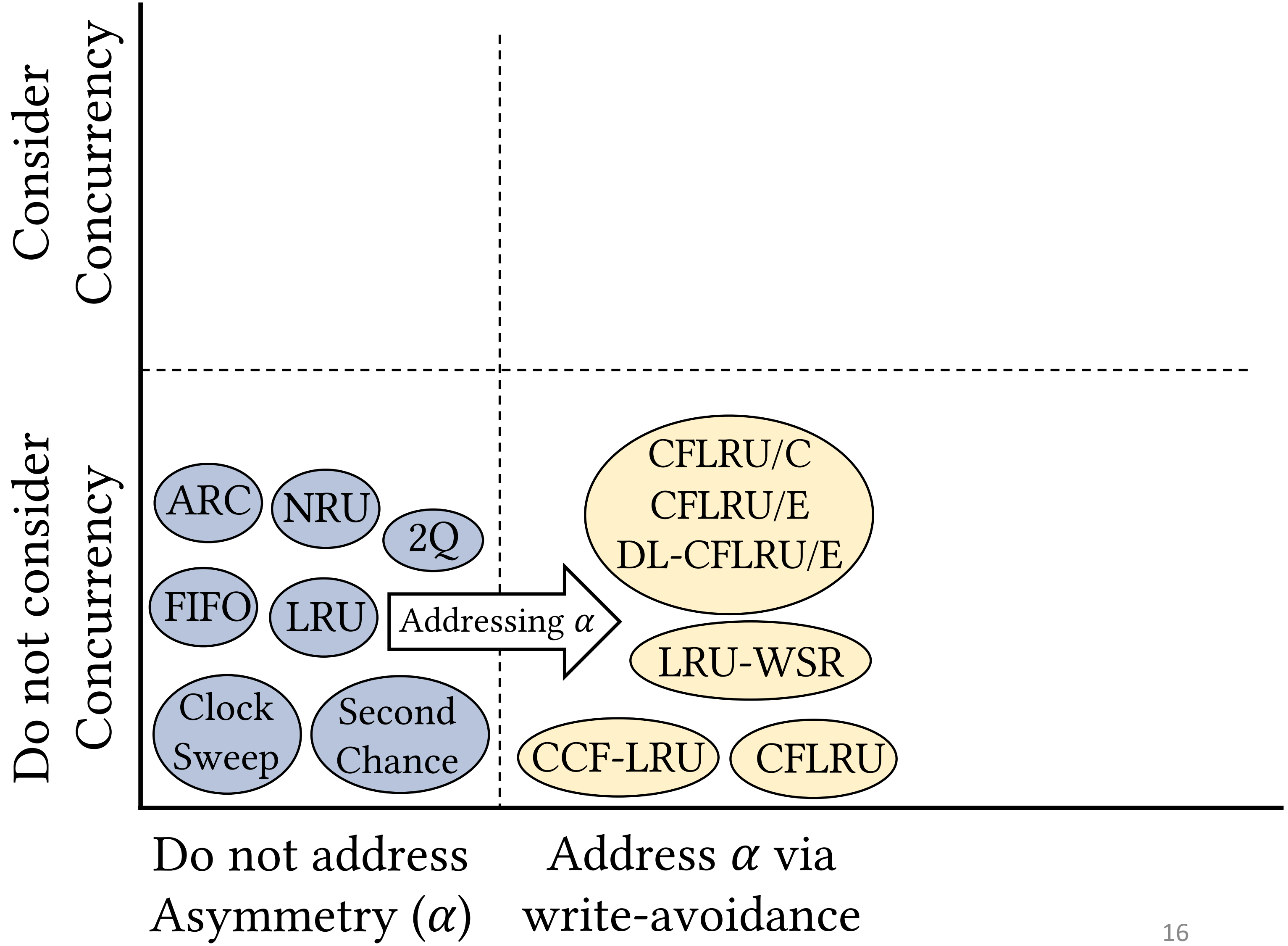
The Challenge

- With write asymmetry, it is **NOT** fair to exchange one write for one read.



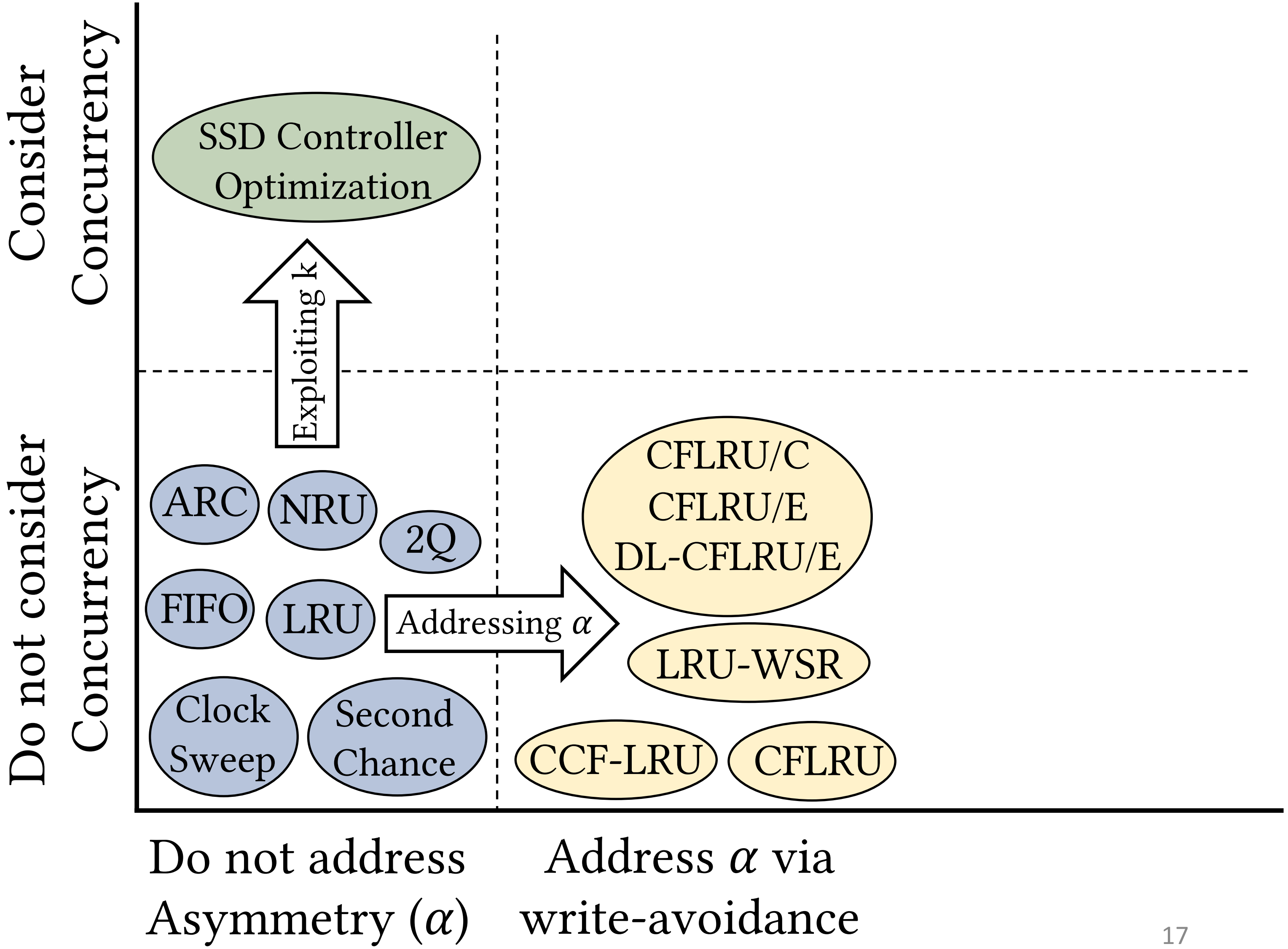
The Challenge

- With write asymmetry, it is **NOT** fair to exchange one write for one read.



The Challenge

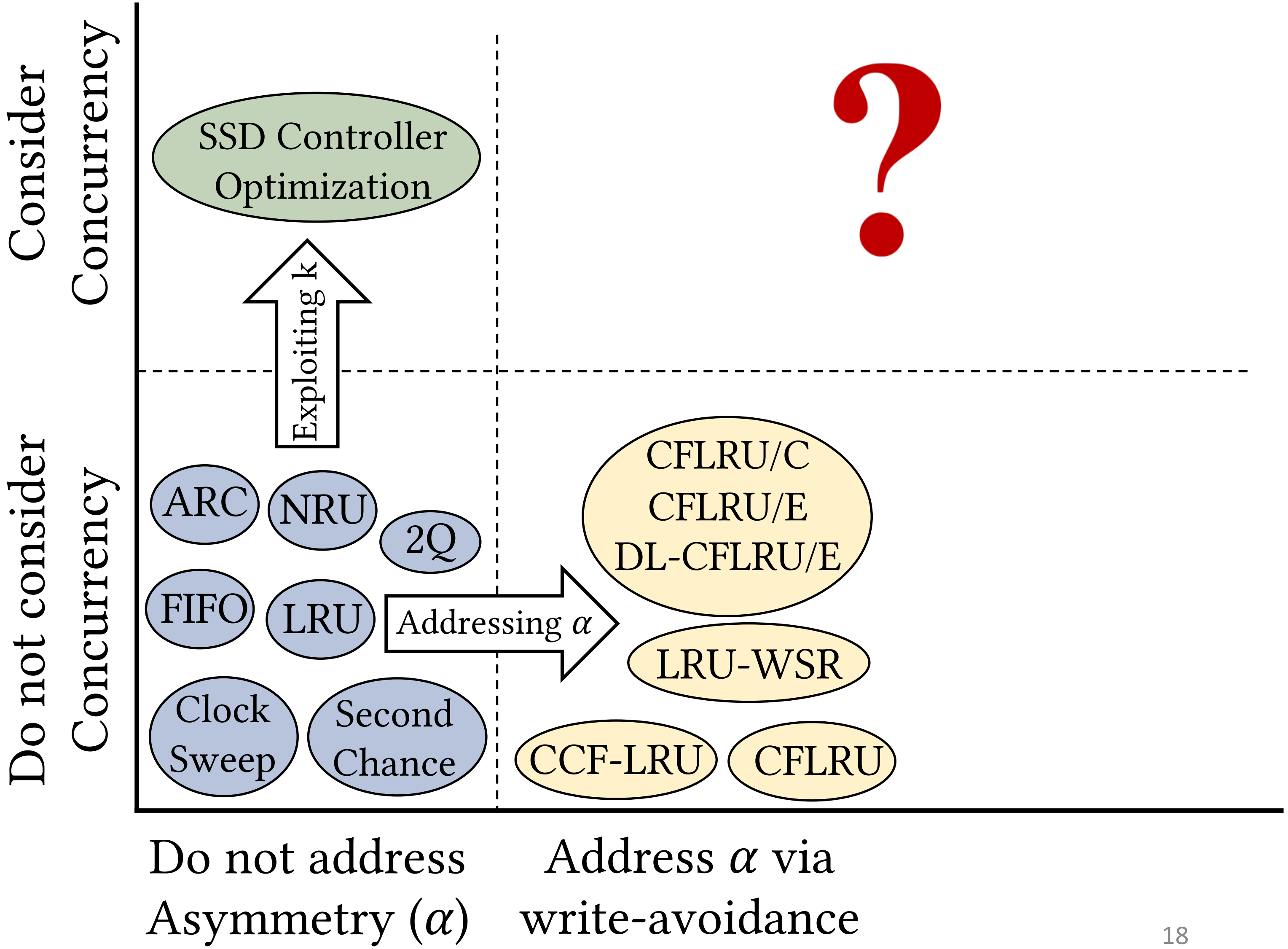
- With write asymmetry, it is **NOT** fair to exchange one write for one read.
- Do not expressly utilize the device concurrency.



The Challenge

- With write asymmetry, it is **NOT** fair to exchange one write for one read.

- Do not expressly utilize the device concurrency.



Consider Concurrency

Do not consider Concurrency

SSD Controller Optimization

Exploiting k

?

ARC
NRU
2Q
FIFO
LRU
Clock Sweep
Second Chance

Addressing α

CFLRU/C
CFLRU/E
DL-CFLRU/E
LRU-WSR
CCF-LRU
CFLRU

Do not address Asymmetry (α)

Address α via write-avoidance

The Challenge

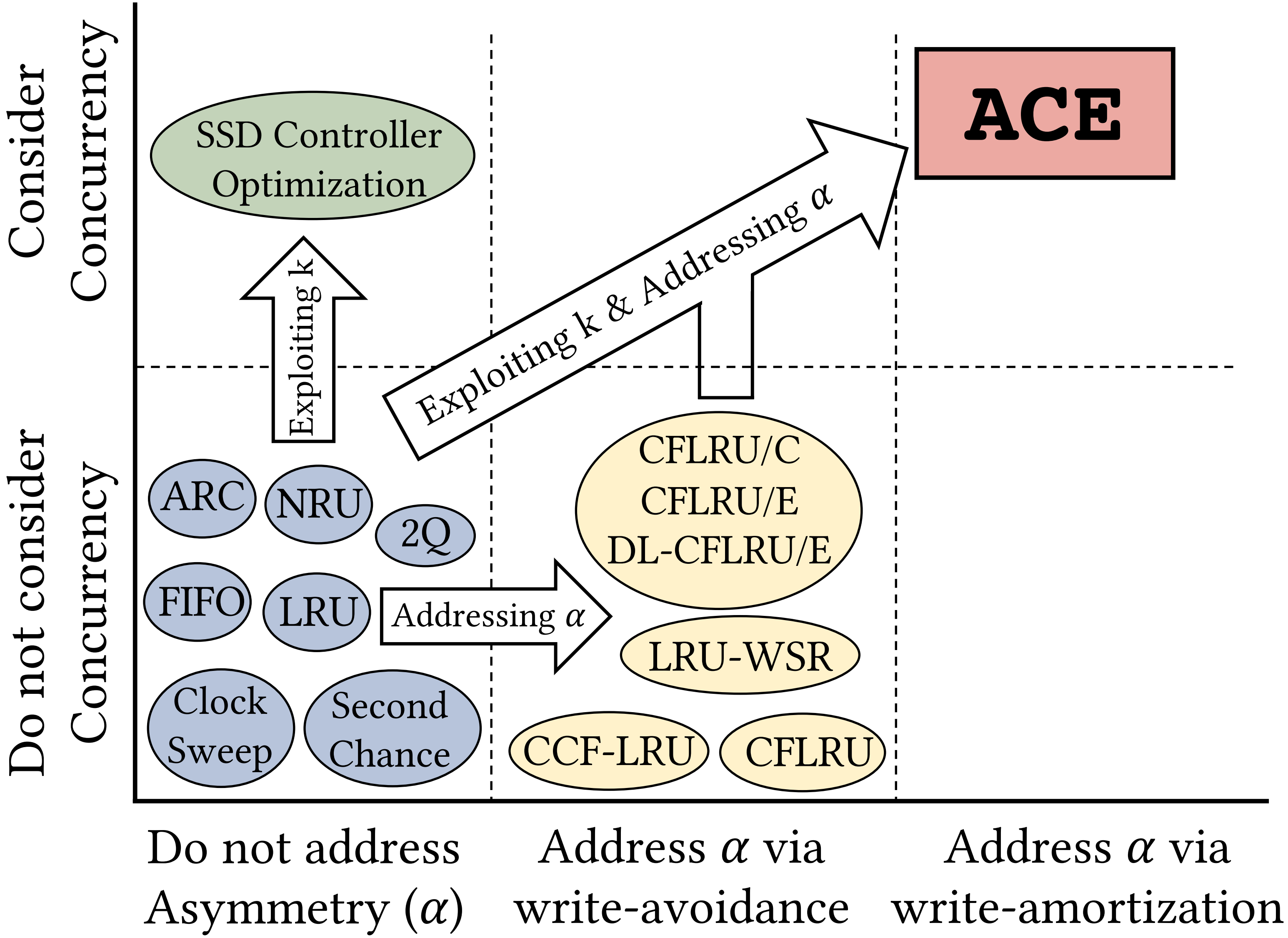
device under-utilization

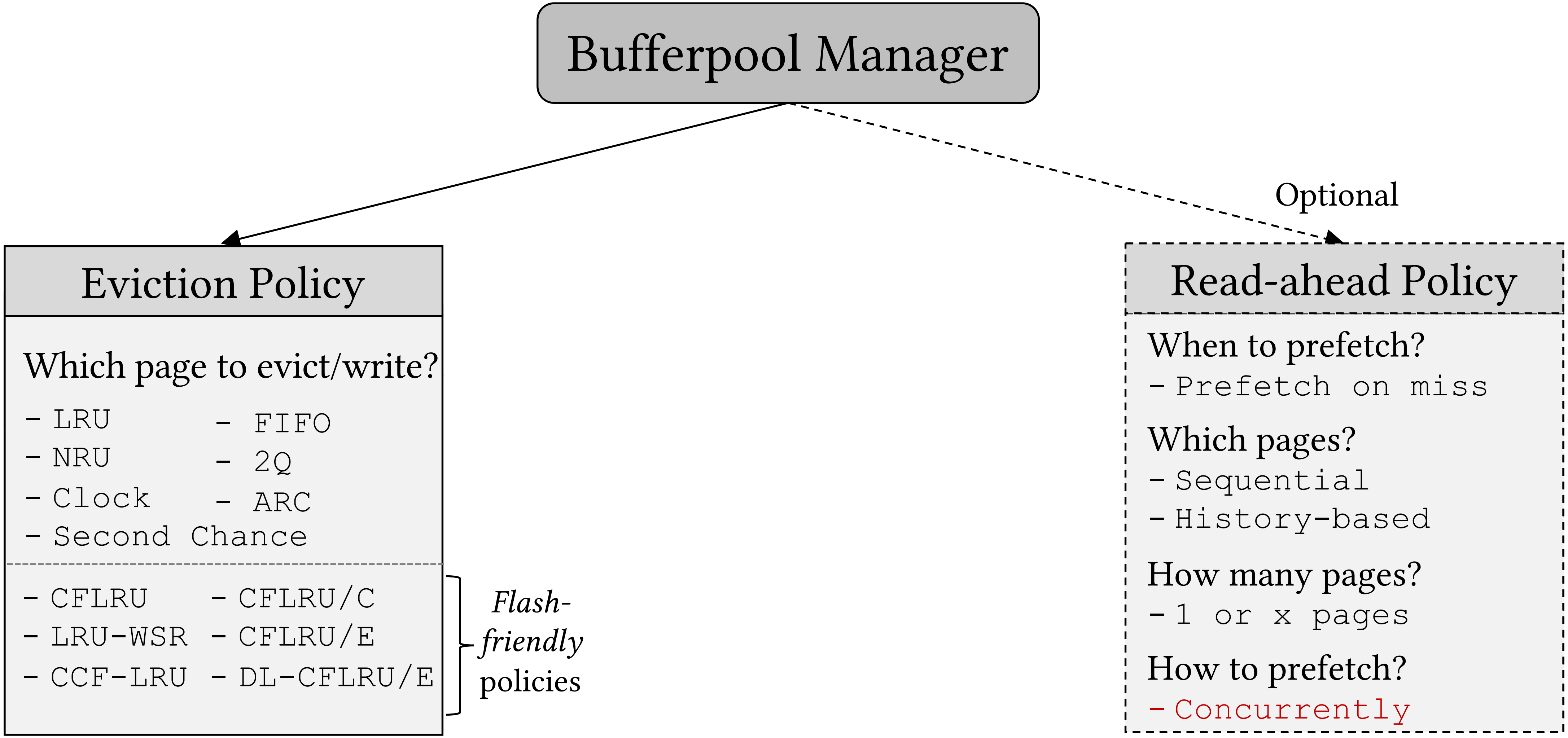
poor end-to-end performance

high deployment cost

The Solution

- device under-utilization ✓
- poor end-to-end performance ✓
- high deployment cost ✓
- **ease of integration** ✓





Bufferpool Manager

Eviction Policy

Which page to evict/write?

- LRU
- NRU
- Clock
- Second Chance
- FIFO
- 2Q
- ARC

- CFLRU
- LRU-WSR
- CCF-LRU
- CFLRU/C
- CFLRU/E
- DL-CFLRU/E

Flash-friendly policies

Optional

Read-ahead Policy

When to prefetch?

- Prefetch on miss

Which pages?

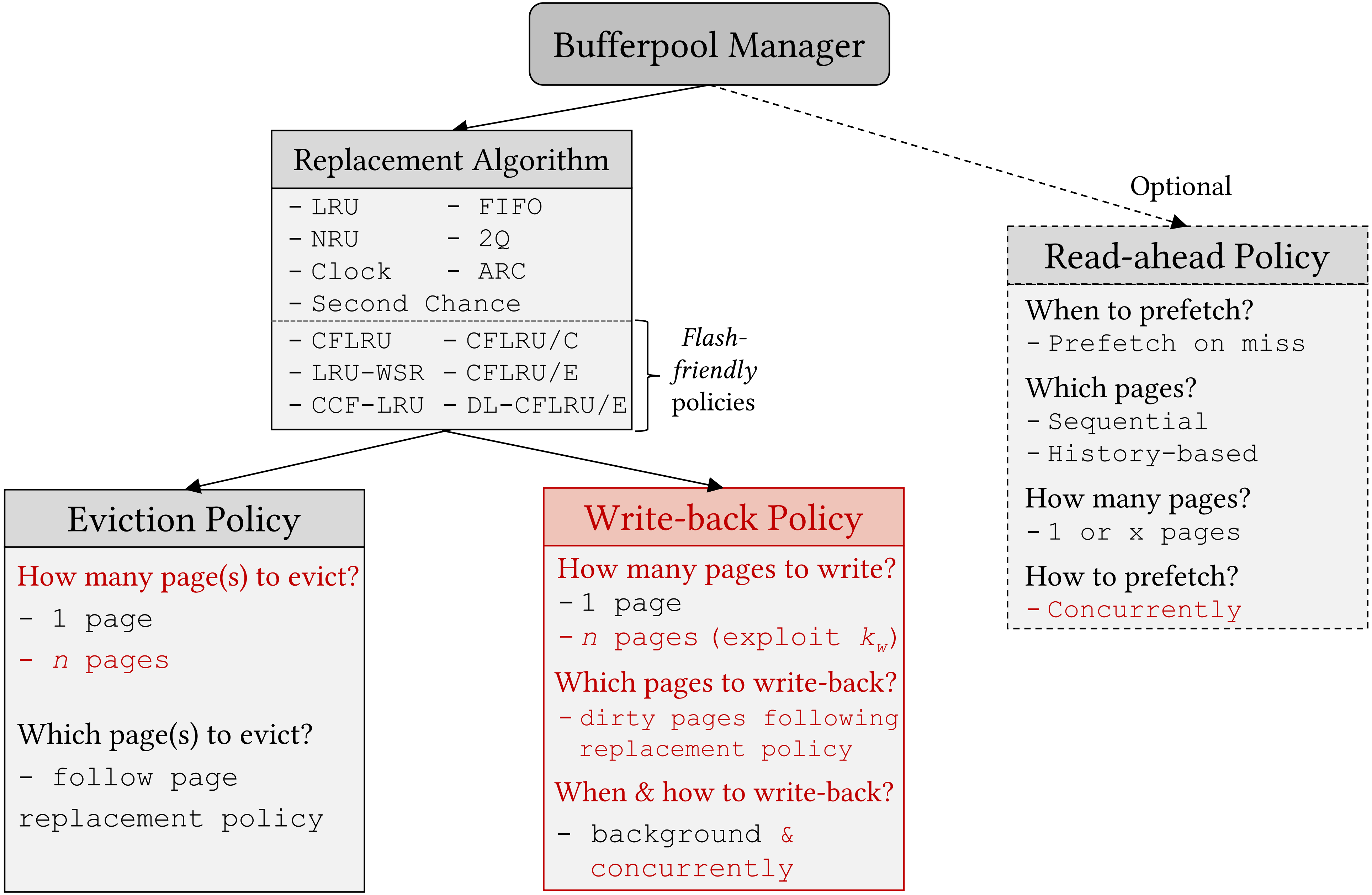
- Sequential
- History-based

How many pages?

- 1 or x pages

How to prefetch?

- Concurrently



Bufferpool Manager

Replacement Algorithm

- LRU
- NRU
- Clock
- Second Chance
- FIFO
- 2Q
- ARC

- CFLRU
- LRU-WSR
- CCF-LRU
- CFLRU/C
- CFLRU/E
- DL-CFLRU/E

Flash-friendly policies

Optional

Read-ahead Policy

When to prefetch?
- Prefetch on miss

Which pages?
- Sequential
- History-based

How many pages?
- 1 or x pages

How to prefetch?
- Concurrently

Eviction Policy

How many page(s) to evict?
- 1 page
- *n* pages

Which page(s) to evict?
- follow page replacement policy

Write-back Policy

How many pages to write?
- 1 page
- *n* pages (exploit k_w)

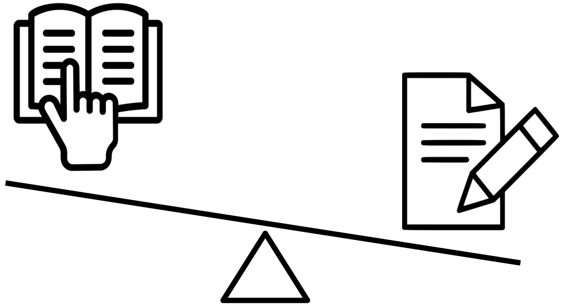
Which pages to write-back?
- dirty pages following replacement policy

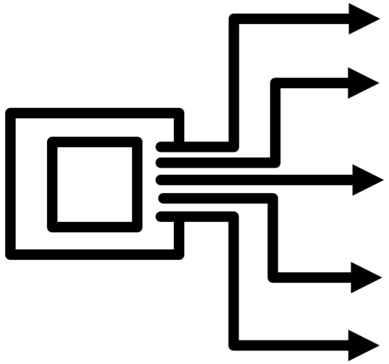
When & how to write-back?
- background & concurrently

Asymmetry/Concurrency-Aware (**`ACE`**) Bufferpool Manager

`ACE` Bufferpool Manager

Use device’s properties





`ACE` Bufferpool Manager

$1 <= n_e <=$ read concurrency (k_r)

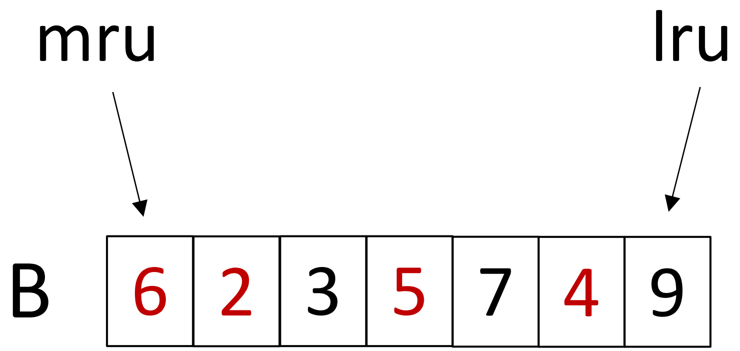
n_w = device's write concurrency (k_w)

write n_w dirty pages concurrently

evict n_e pages

prefetch n_e - 1 pages concurrently

An Example



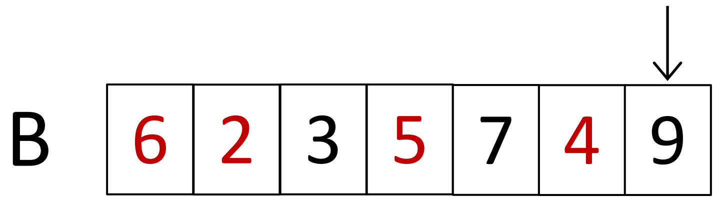
Let's assume: $k_w = 3$, LRU is the baseline replacement policy & red indicates dirty page

Write request of page 8 comes

An Example ($k_w = 3$)

write page 8

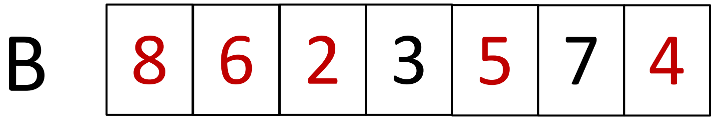
Candidate for eviction



B							
	6	2	3	5	7	4	9

Since candidate page is clean, we simply evict 9

After eviction:



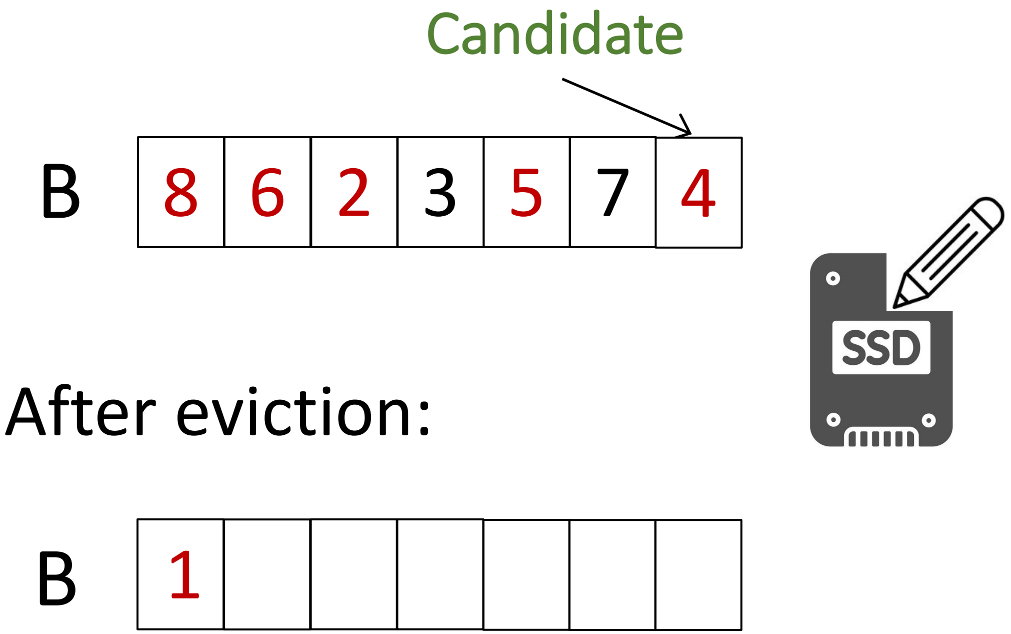
B							
	8	6	2	3	5	7	4

Write request of page 1 comes

An Example ($k_w = 3$)

`write page 1`

`LRU`

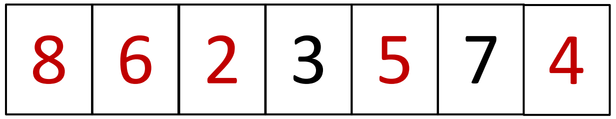


After eviction:

An Example ($k_w = 3$)

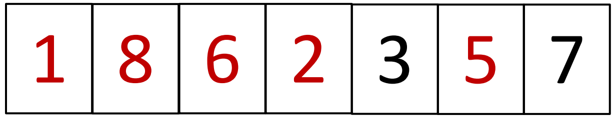
write page 1

LRU



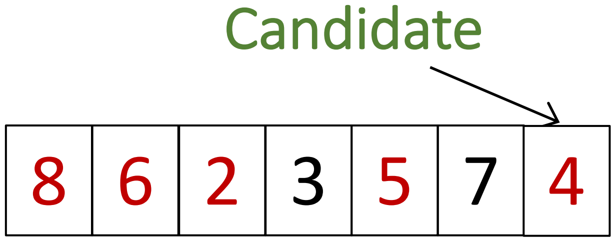
B						
8	6	2	3	5	7	4

After eviction:



B						
1	8	6	2	3	5	7

LRU+ACE (w/o PF)



Candidate

8	6	2	3	5	7	4

An Example ($k_w = 3$)

write page 1

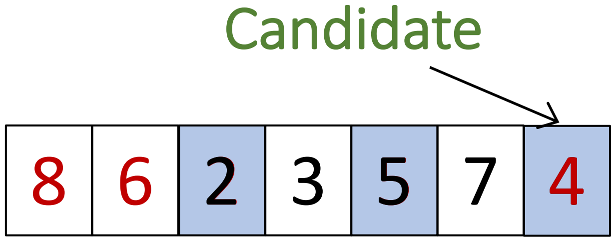
LRU

B	8	6	2	3	5	7	4

After eviction:

B	1	8	6	2	3	5	7

LRU+ACE (w/o PF)



8	6	2	3	5	7	4

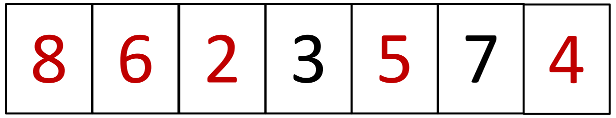
4,5,2 concurrently written

4 evicted

An Example ($k_w = 3$)

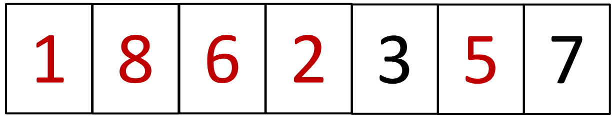
write page 1

LRU



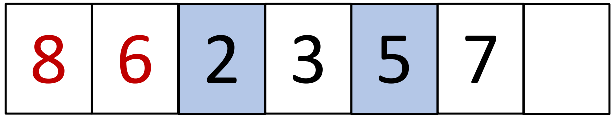
B							
	8	6	2	3	5	7	4

After eviction:



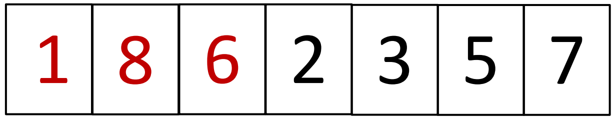
B							
	1	8	6	2	3	5	7

LRU+ACE (w/o PF)



8	6	2	3	5	7	

After eviction:

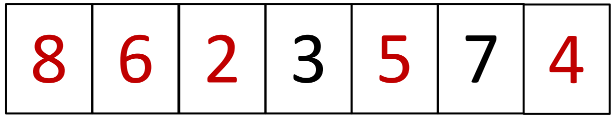


1	8	6	2	3	5	7

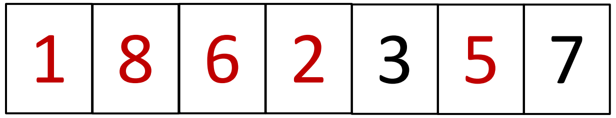
An Example ($k_w = 3, n_e = 2$)

`write page 1`

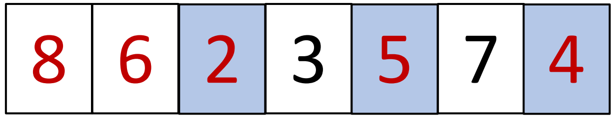
`LRU`

B

8 6 2 3 5 7 4

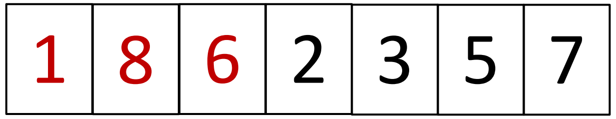
After eviction:

B

1 8 6 2 3 5 7

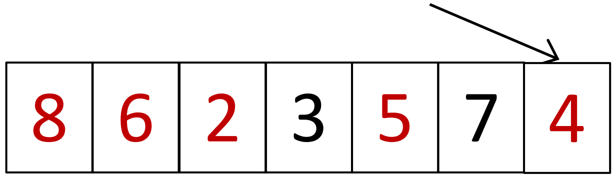
`LRU+ACE (w/o PF)`


8 6 2 3 5 7 4

After eviction:


1 8 6 2 3 5 7

`LRU+ACE (w/PF)`

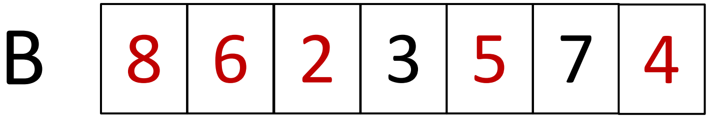

Candidate

8 6 2 3 5 7 4

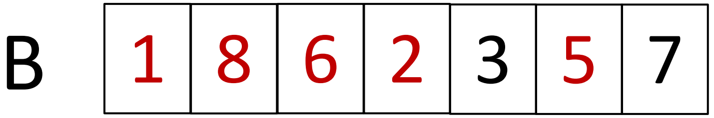
An Example ($k_w = 3, n_e = 2$)

write page 1

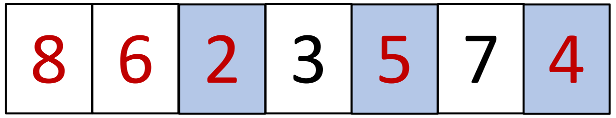
LRU


B 8 6 2 3 5 7 4

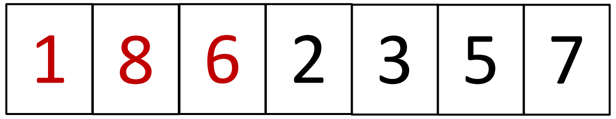
After eviction:


B 1 8 6 2 3 5 7

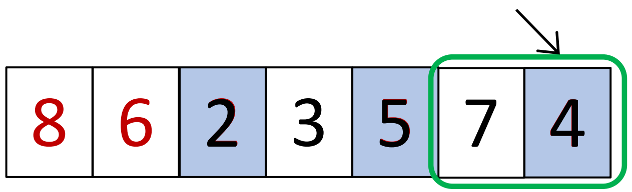
LRU+ACE (w/o PF)


8 6 2 3 5 7 4

After eviction:


1 8 6 2 3 5 7

LRU+ACE (w/PF)



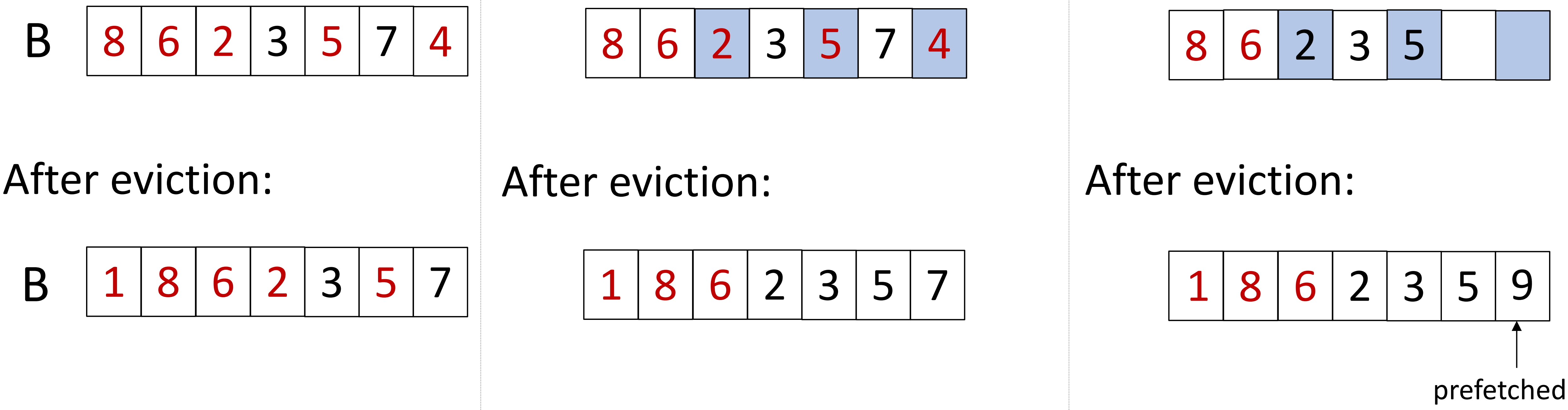
8 6 2 3 5 7 4

4,5,2 concurrently written

4,7 evicted

An Example ($k_w = 3, n_e = 2$)

`write page 1`



LRU

B	8	6	2	3	5	7	4

After eviction:

B	1	8	6	2	3	5	7

LRU+ACE (w/o PF)

8	6	2	3	5	7	4

After eviction:

1	8	6	2	3	5	7

LRU+ACE (w/PF)

8	6	2	3	5		

After eviction:

1	8	6	2	3	5	9

prefetched

Experimental Evaluation

PostgreSQL
11.5

Clock Sweep
LRU
CFLRU
LRU-WSR

vs

their ACE counterparts

Device	α	k_r	k_w
Optane SSD	1.1	6	5
PCIe SSD	2.8	80	8
SATA SSD	1.5	25	9
Virtual SSD	2.0	11	19

Workload:

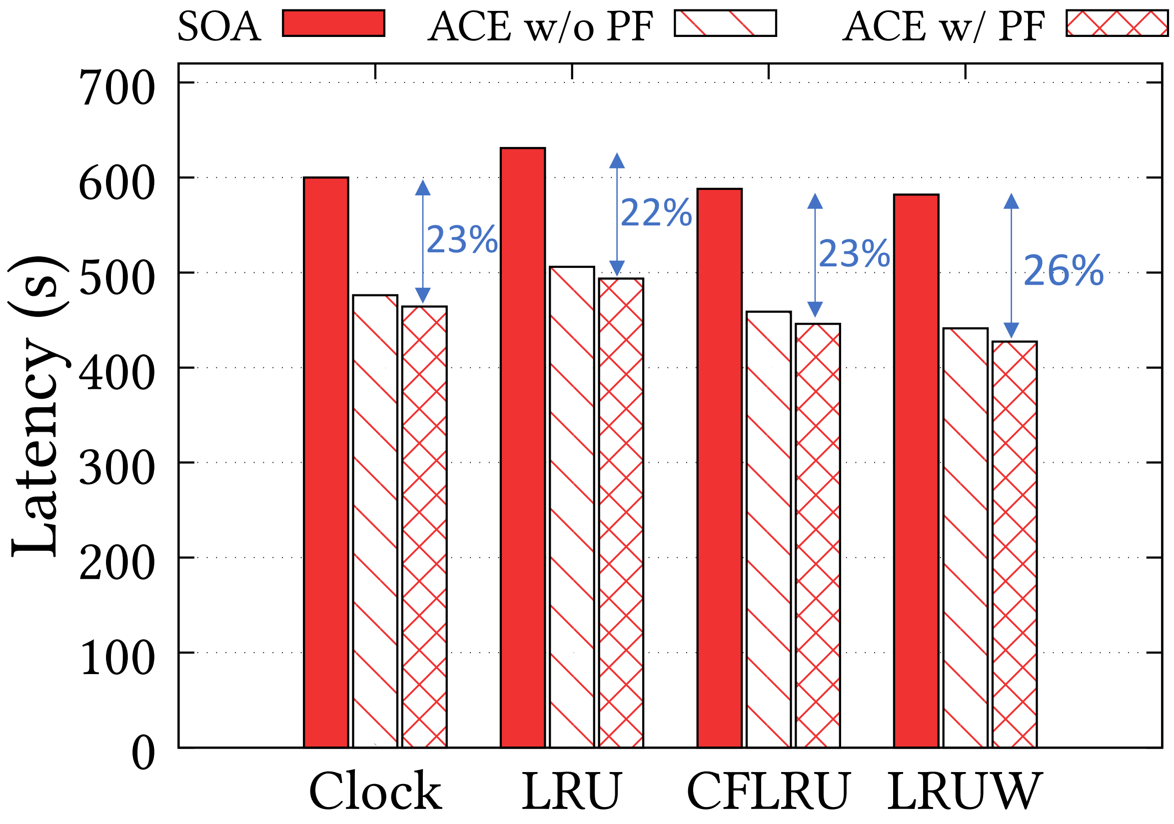
synthesized traces

TPC-C benchmark

ACE Improves Runtime

Device: PCIe SSD

$\alpha = 2.8, k_w = 8$



Mixed Skewed Trace
(r/w: 50/50, locality 90/10)

ACE improves runtime by 22-26%

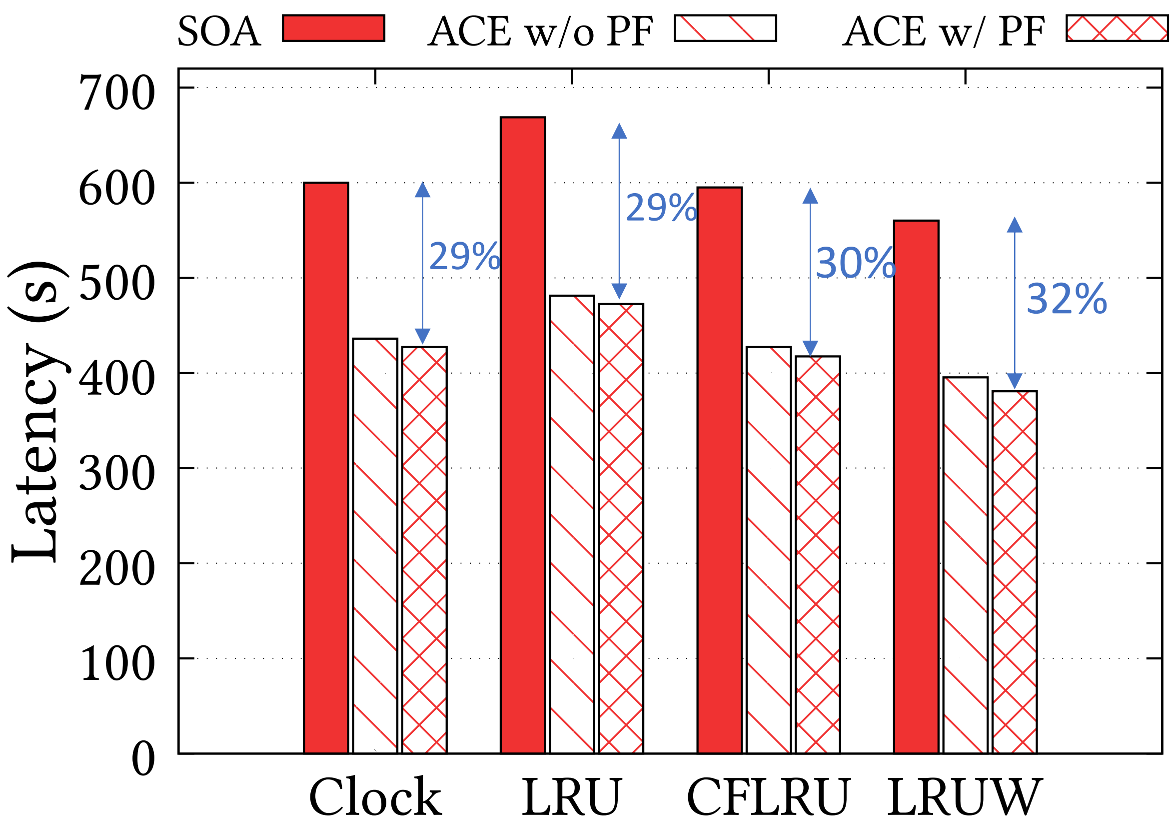
Negligible increase in buffer miss (<0.009%)

Benefit comes at no cost

Higher Gain for Write-Heavy Workload

Device: PCIe SSD

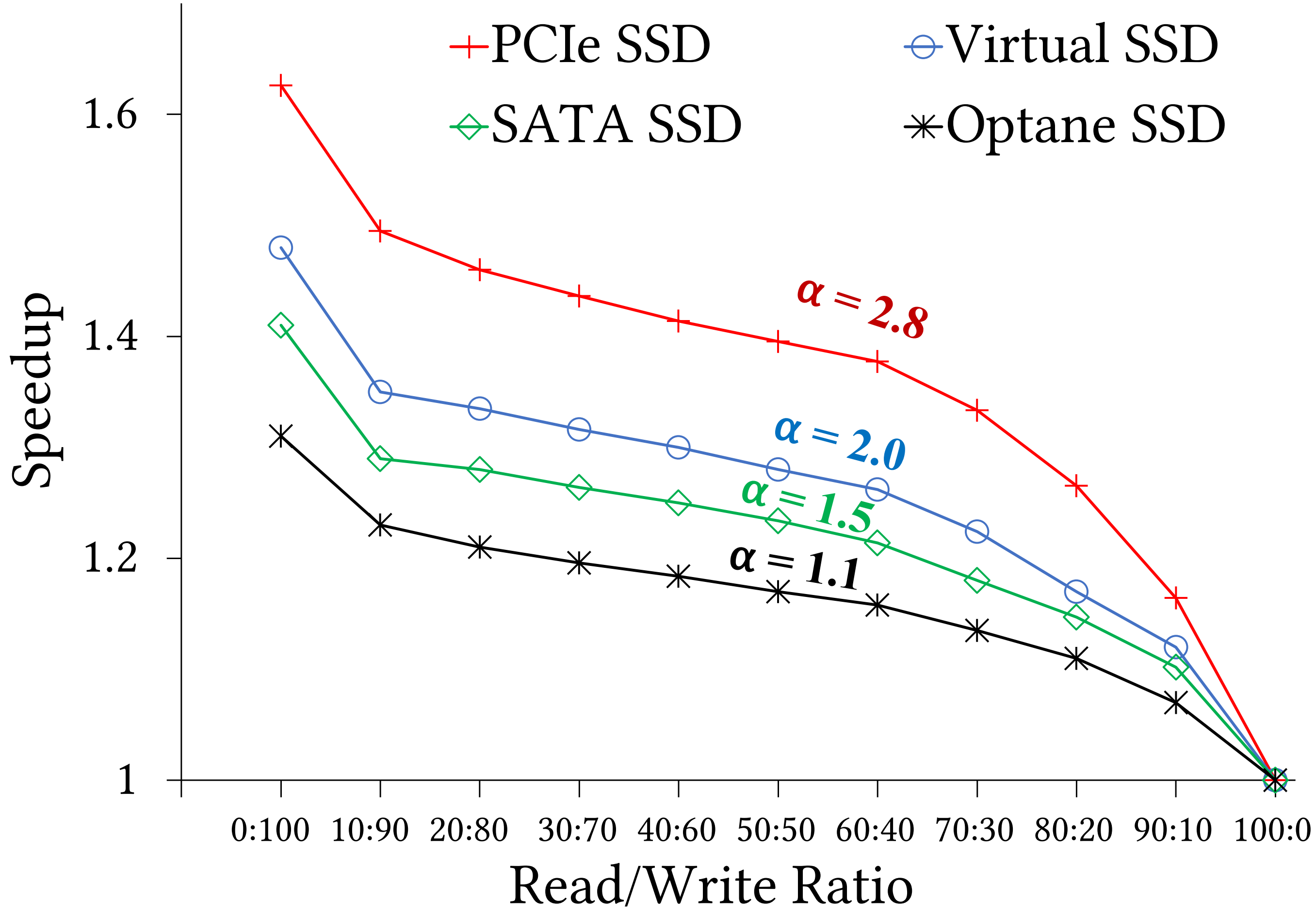
$\alpha = 2.8, k_w = 8$



Mixed Skewed Trace
(r/w: 50/50, locality 90/10)

Write-intensive workloads have higher benefit (up to 32%)

Impact of R/W Ratio & Asymmetry

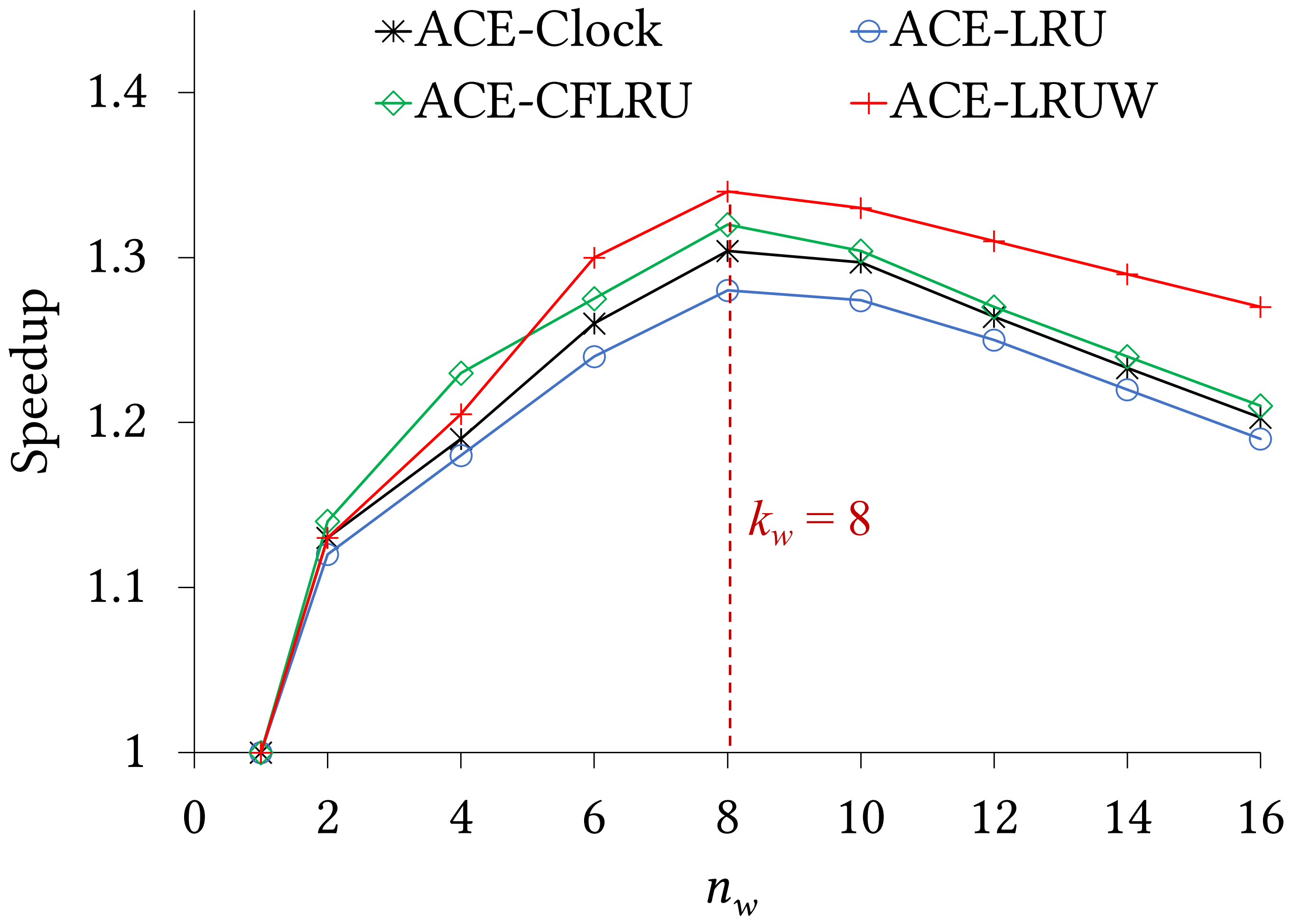


more writes, more speedup

higher asymmetry, higher speedup

good benefit even for low asymmetry

Impact of #Concurrent I/Os



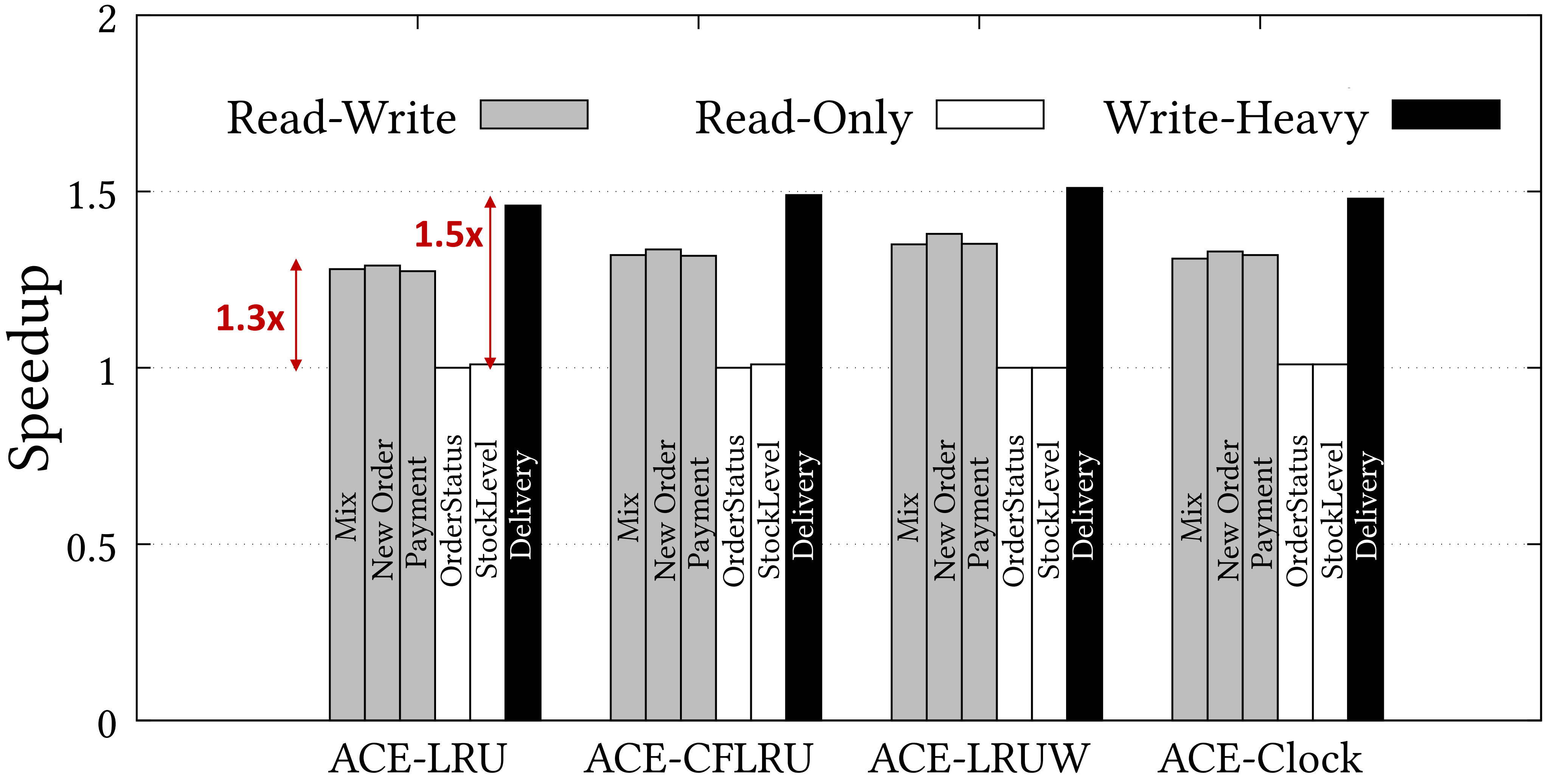
Mixed Skewed Trace
(r/w: 50/50, locality 90/10)

Device: PCIe SSD

$\alpha = 2.8$, $k_w = 8$

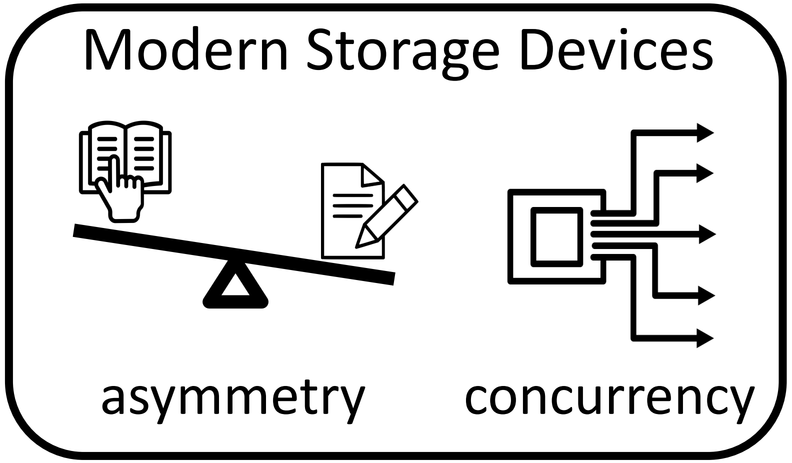
Highest speedup when optimal concurrency is used

Experimental Evaluation (TPC-C)



ACE Achieves 1.3x for mixed TPC-C

Summary

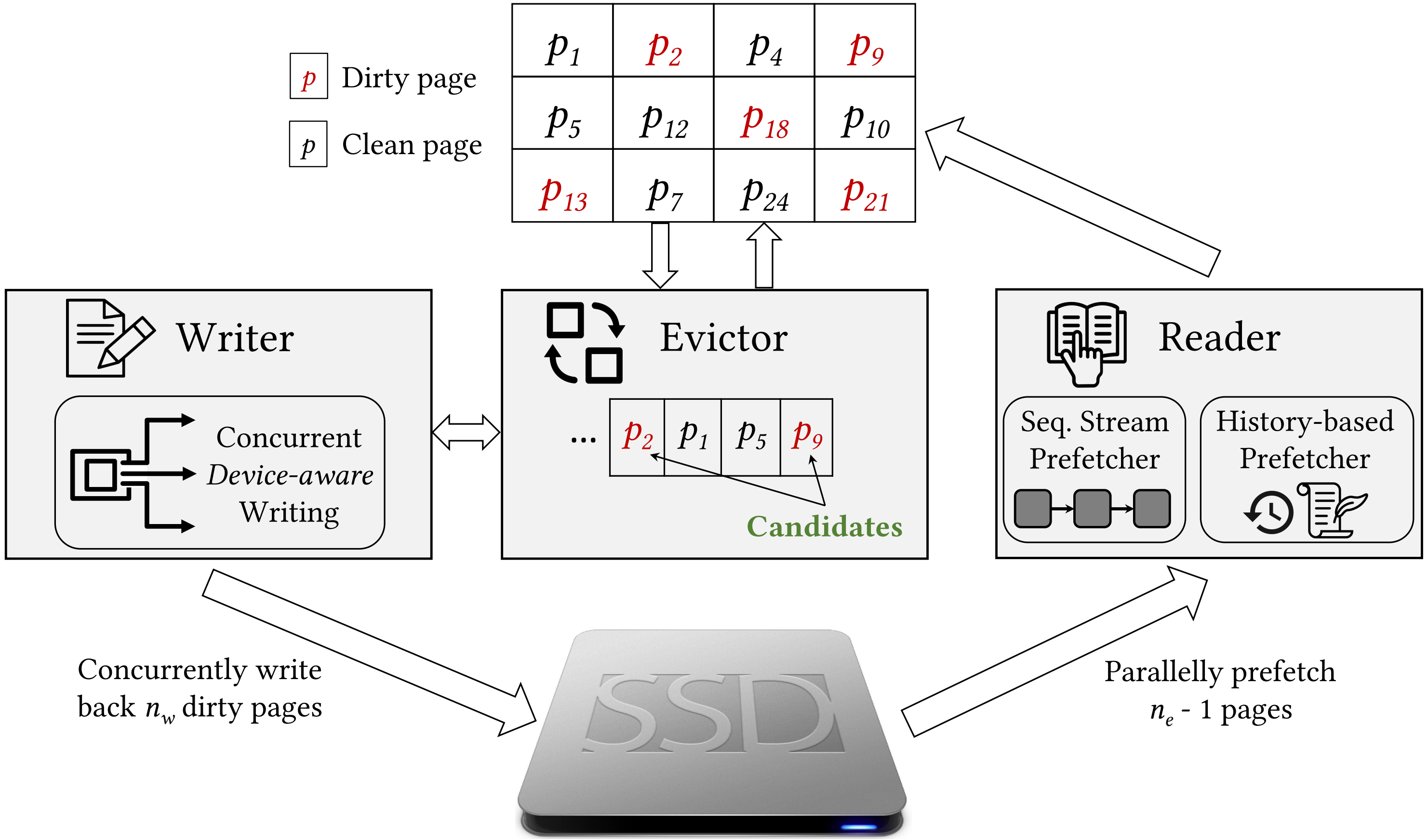


Modern Storage Devices

asymmetry

concurrency

ACE Bufferpool



p Dirty page

p Clean page

p_1	p_2	p_4	p_9
p_5	p_{12}	p_{18}	p_{10}
p_{13}	p_7	p_{24}	p_{21}

Writer

Concurrent *Device-aware* Writing

Evictor

… p_2 p_1 p_5 p_9

Candidates

Reader

Seq. Stream Prefetcher

History-based Prefetcher

Concurrently write back n_w dirty pages

Parallelly prefetch n_e - 1 pages

decoupled eviction and write-back mechanism

can be integrated with **any** replacement policy

benefit comes with no penalty

Thank You!

papon@bu.edu

Read/Write Asymmetry - Example

Device	Advertised Rand Read IOPS	Advertised Rand Write IOPS	Advertised Asymmetry
PCIe D5-P4320	427k	36k	11.9
PCIe DC-P4500	626k	51k	12.3
PCIe P4510	465k	145k	3.2
SATA D3-S4610	92k	28k	3.3
Optane P4800X	550k	500k	1.1

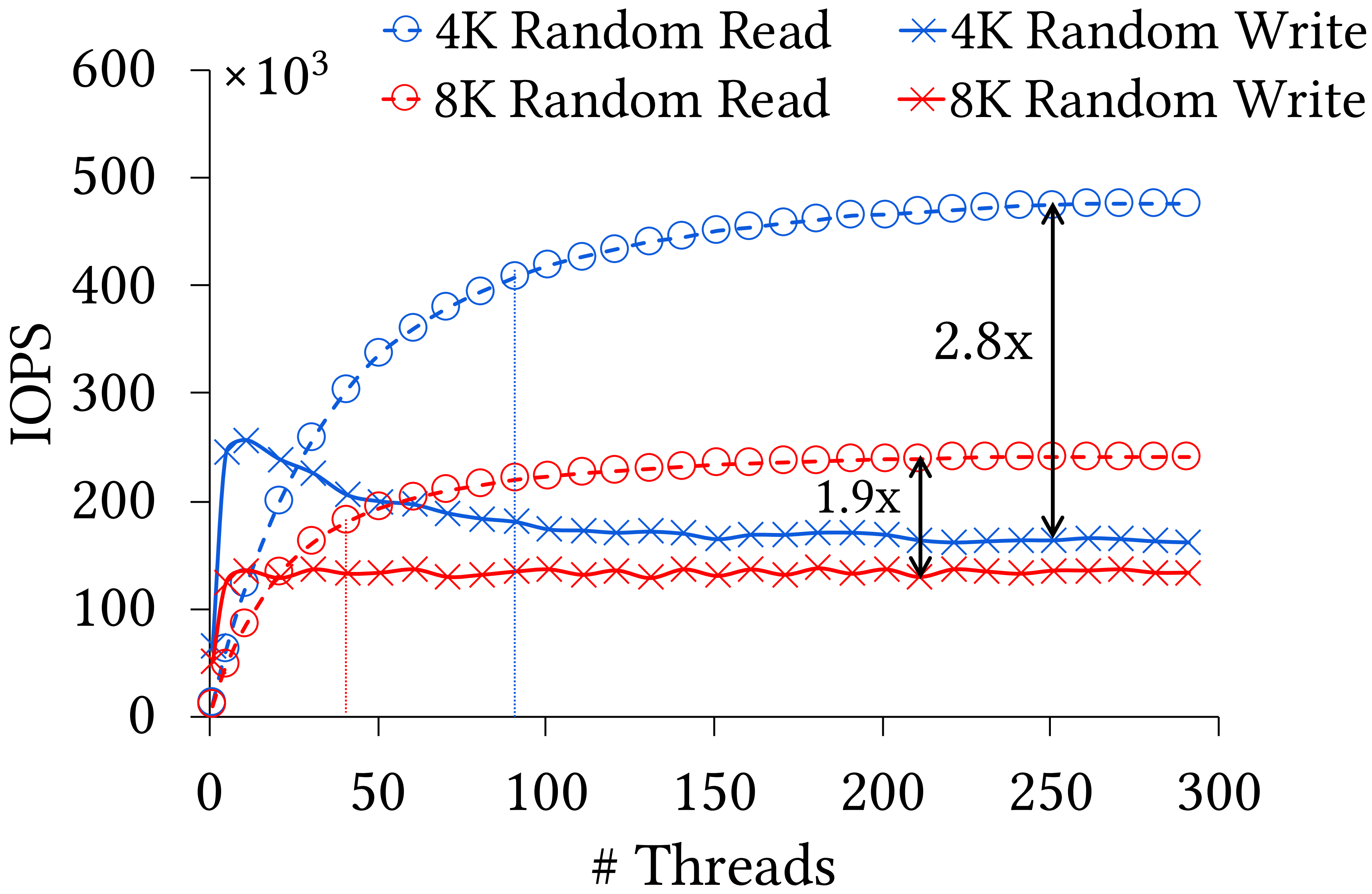
Measuring Asymmetry & Concurrency

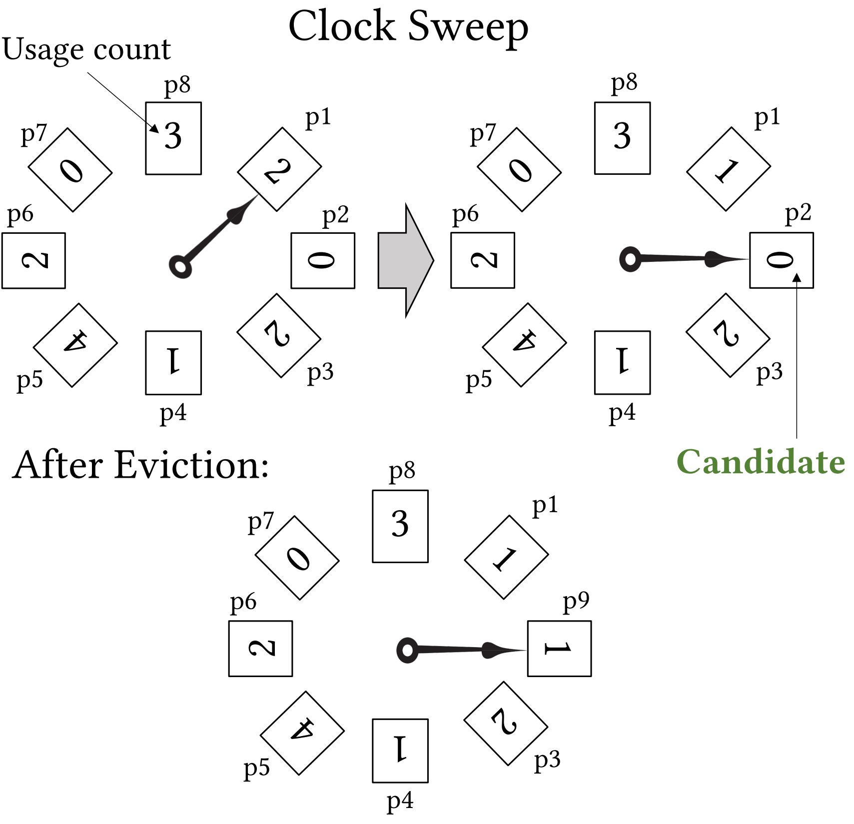
Device
PCIe SSD - P4510 (1TB)

For 4K random read,

Asymmetry: 2.8

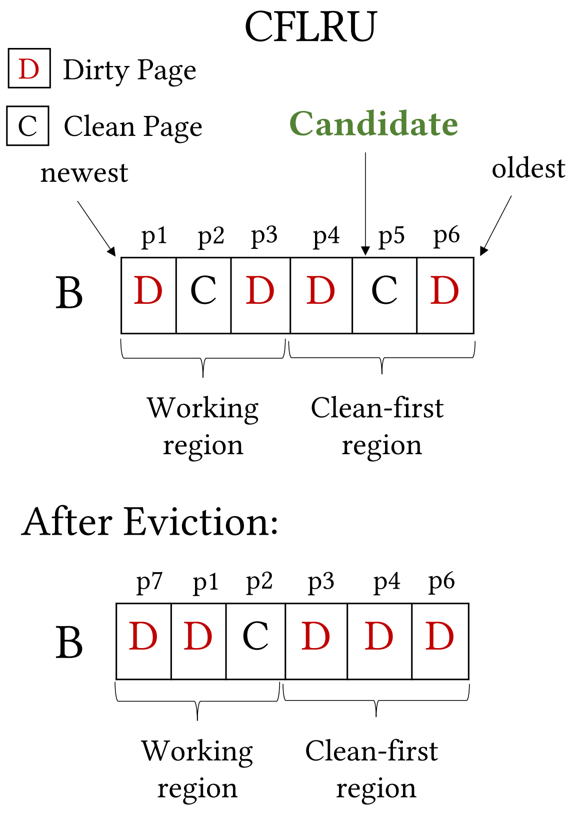
Concurrency: 80





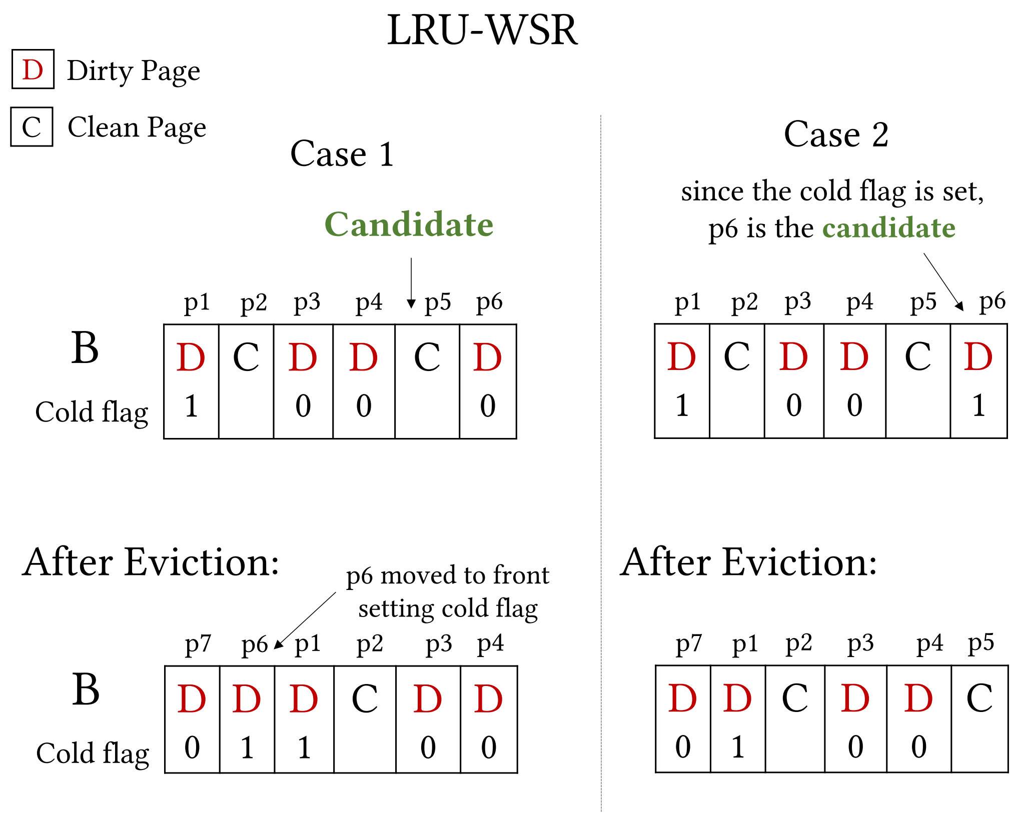
Clock Sweep

(A)



CFLRU

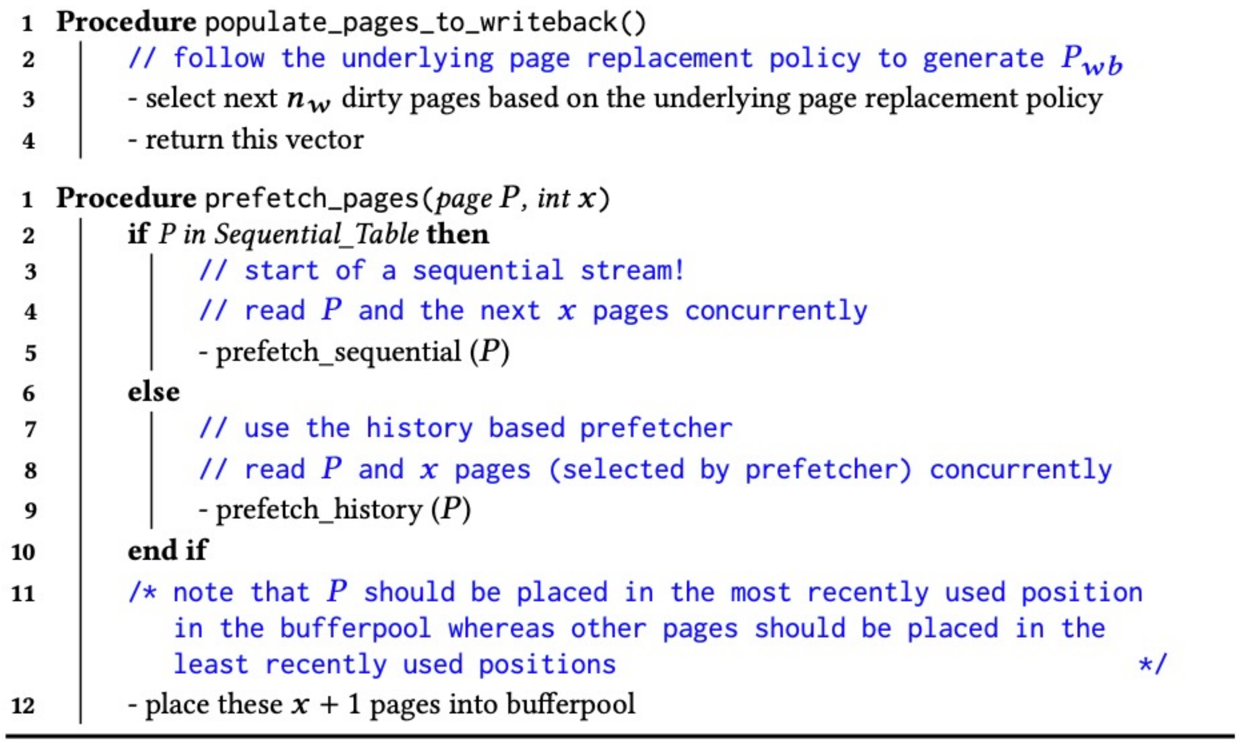
(B)



LRU-WSR

(C)

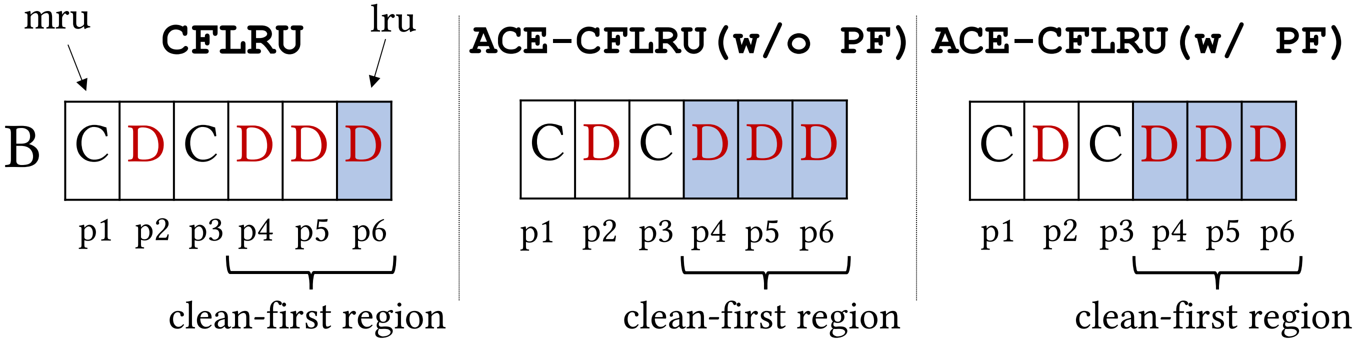
```
Algorithm 1: ACE
Input: $P$, $n_w$, $n_e$ $is\_pf\_enabled$
// $P$ is the accessed page
// $n_w$ is the maximum effective write concurrency ($n_w = k_w$)
// $n_e$ is the number of concurrent reads when prefetching is enabled
// $is\_pf\_enabled$ determines if prefetching is enabled or not
if $P$ in buferpool then
    return $P$
else
    // miss! need to bring P from disk
    if buferpool not full then
        if $is\_pf\_enabled == true$ then
            // reads P and prefetches up to $n_e - 1$ pages from disk
               (depending on available slots)
            - prefetch_pages ($P$, $n_e - 1$)
        else
            - read $P$ from disk
        end if
    else
        $top\_page$ = replacement_policy.get_one_page_to_evict()
        if $top\_page$ is clean then
            // follow classical approach if page is clean
            - drop $top\_page$ from bufferpool
            - read $P$ from disk
        else
            // $top\_page$ is dirty. concurrently write $n_w$ dirty pages
            // $P_{wb}$ is a vector containing the candidate dirty pages
            - $P_{wb}$ = populate_pages_to_writeback()
            - issue ||length($P_{wb}$)|| concurrent writes, $\forall p \in P_{wb}$
            - mark ||length($P_{wb}$)|| pages as clean, $\forall p \in P_{wb}$
            if $is\_pf\_enabled == true$ then
                // evict $n_e$ pages
                // pages written and to be evicted can be different
                // $P_{ev}$ is a vector containing the pages to evict
                $P_{ev}$ = replacement_policy.get_n_pages_to_evict()
                - drop ||length($P_{ev}$)|| pages from bufferpool, $\forall p \in P_{ev}$
                // Now, prefetch
                - prefetch_pages ($P$, $n_e - 1$)
                - empty $P_{ev}$
            else
                // evict 1 page
                - drop $top\_page$ from bufferpool
                - read $P$ from disk
            end if
            - empty $P_{wb}$
        end if
    end if
end if
```



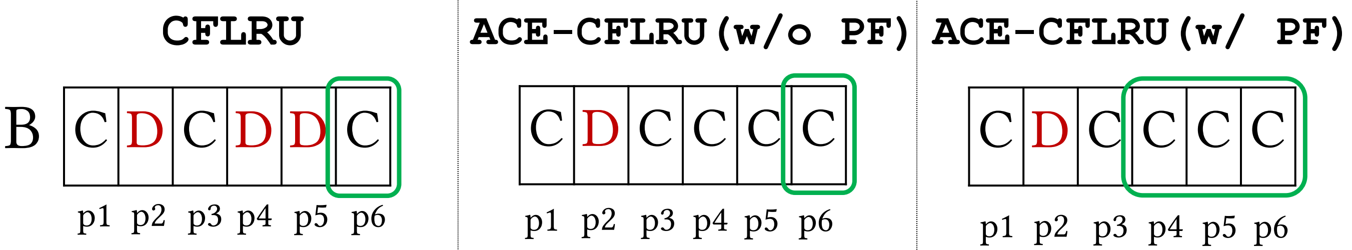
```
Procedure populate_pages_to_writeback()
    // follow the underlying page replacement policy to generate $P_{wb}$
    - select next $n_w$ dirty pages based on the underlying page replacement policy
    - return this vector

Procedure prefetch_pages(page $P$, int $x$)
    if $P$ in Sequential_Table then
        // start of a sequential stream!
        // read $P$ and the next $x$ pages concurrently
        - prefetch_sequential ($P$)
    else
        // use the history based prefetcher
        // read $P$ and $x$ pages (selected by prefetcher) concurrently
        - prefetch_history ($P$)
    end if
    /* note that $P$ should be placed in the most recently used position
       in the bufferpool whereas other pages should be placed in the
       least recently used positions                                   */
    - place these $x + 1$ pages into bufferpool
```

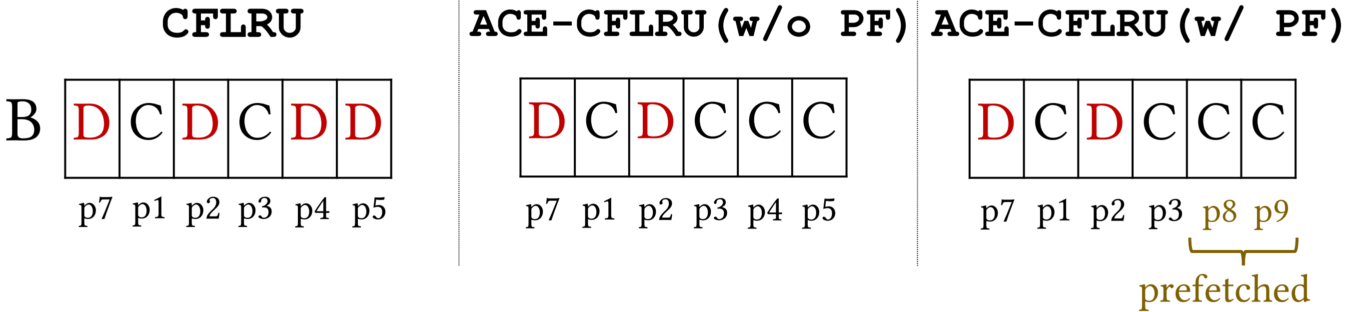
Initial State (Before `write p7`)



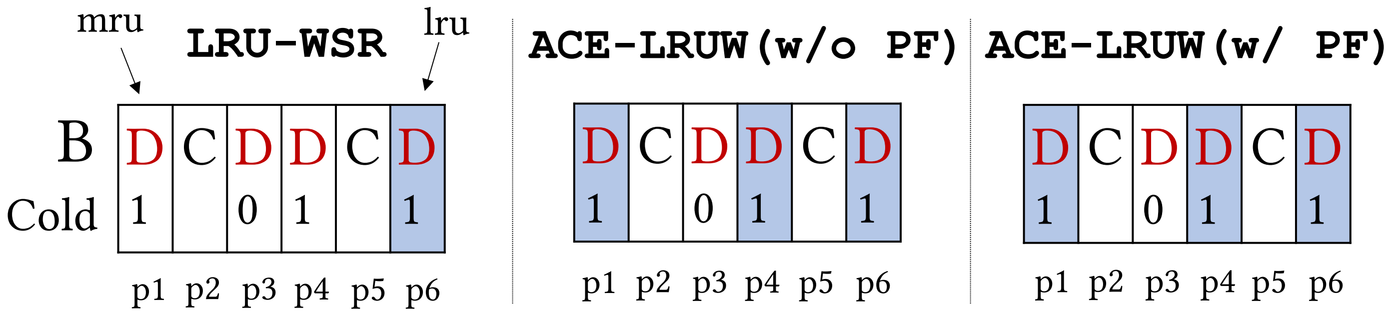
Intermediate State



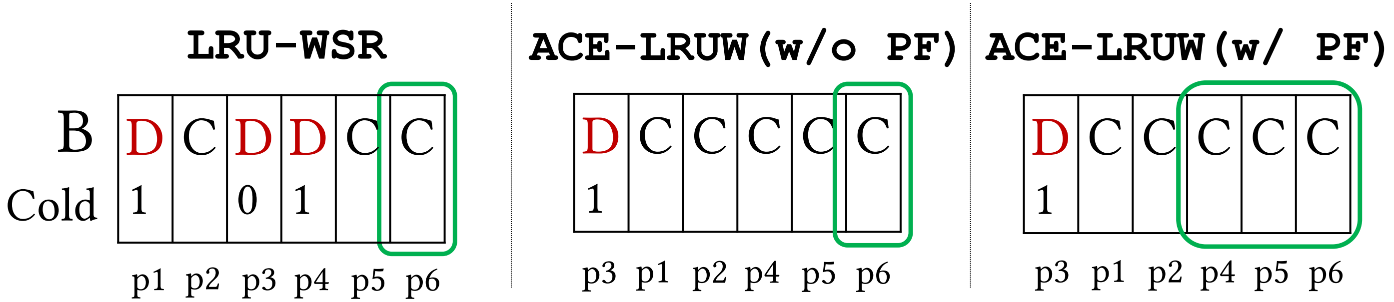
Final State (After `write p7`)



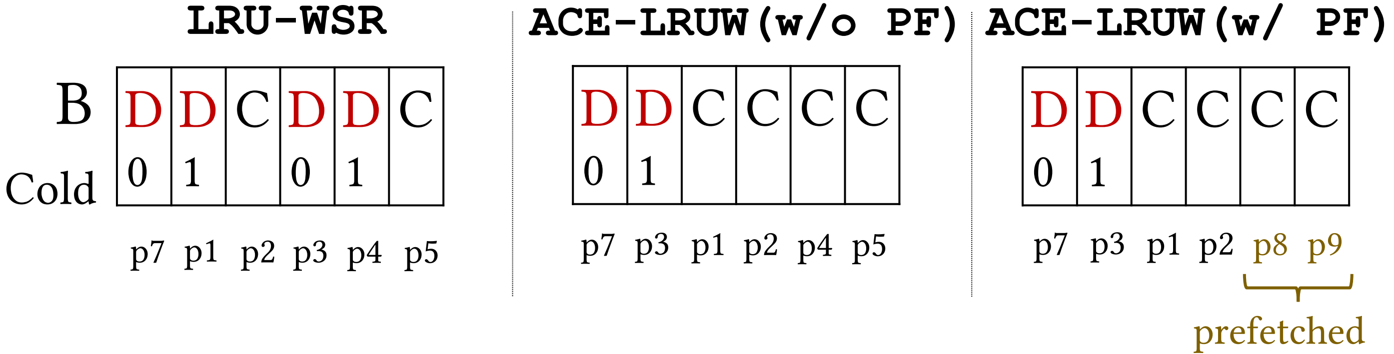
Initial State (Before `write p7`)



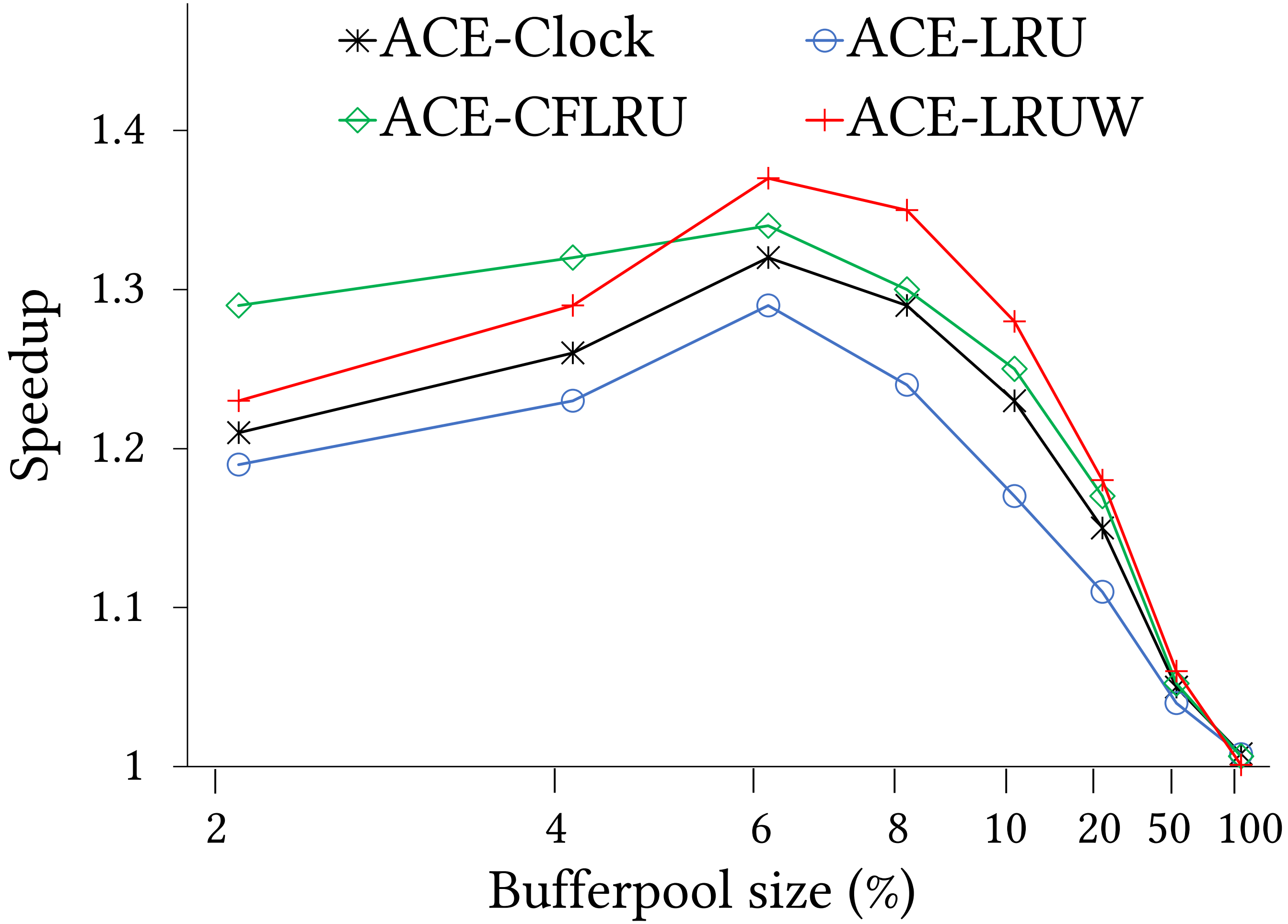
Intermediate State



Final State (After `write p7`)



Experimental Evaluation



ACE performs well under memory pressure

Impact on #writes