BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows

First presented at: *27th IEEE International Symposium on High-Performance Computer Architecture, 2021*

Authors: *A. Giray Yağlıkçı[1], Minesh Patel[1], Jeremie S. Kim[1], Roknoddin Azizi[1], Ataberk Olgun[1], Lois Oros[1], Hasan Hassan[1], Jisung Park[1], Konstantinos Kanellopoulos[1], Taha Shahroodi[1], Saugata Ghose[2], Onur Mutlu[1]*

[1]ETH Zürich *[2]University of Illinois at Urbana–Champaign*

Presented by: Sofie Daniëls

Executive summary

Problem:

- Memory density scaling of DRAM chips causes increasing vulnerability to RowHammer, but most solutions can't scale accordingly
- Current solutions often require knowledge of or modification to DRAM internals

Goal:

- Find scalable and efficient way to prevent RowHammer without modifying DRAM chip

Key idea:

- Selectively throttle memory accesses that can cause bit-flips

Mechanism:

- Tracking all row activations and throttling RowHammer unsafe row accesses
- Identifying and throttling potential attacker threads

Results:

- Hardware complexity: scalable
- Performance & energy consumption: efficient & scalable

Overview

BACKGROUND, PROBLEM & GOAL

MECHANISMS & IMPLEMENTATION

RESULTS

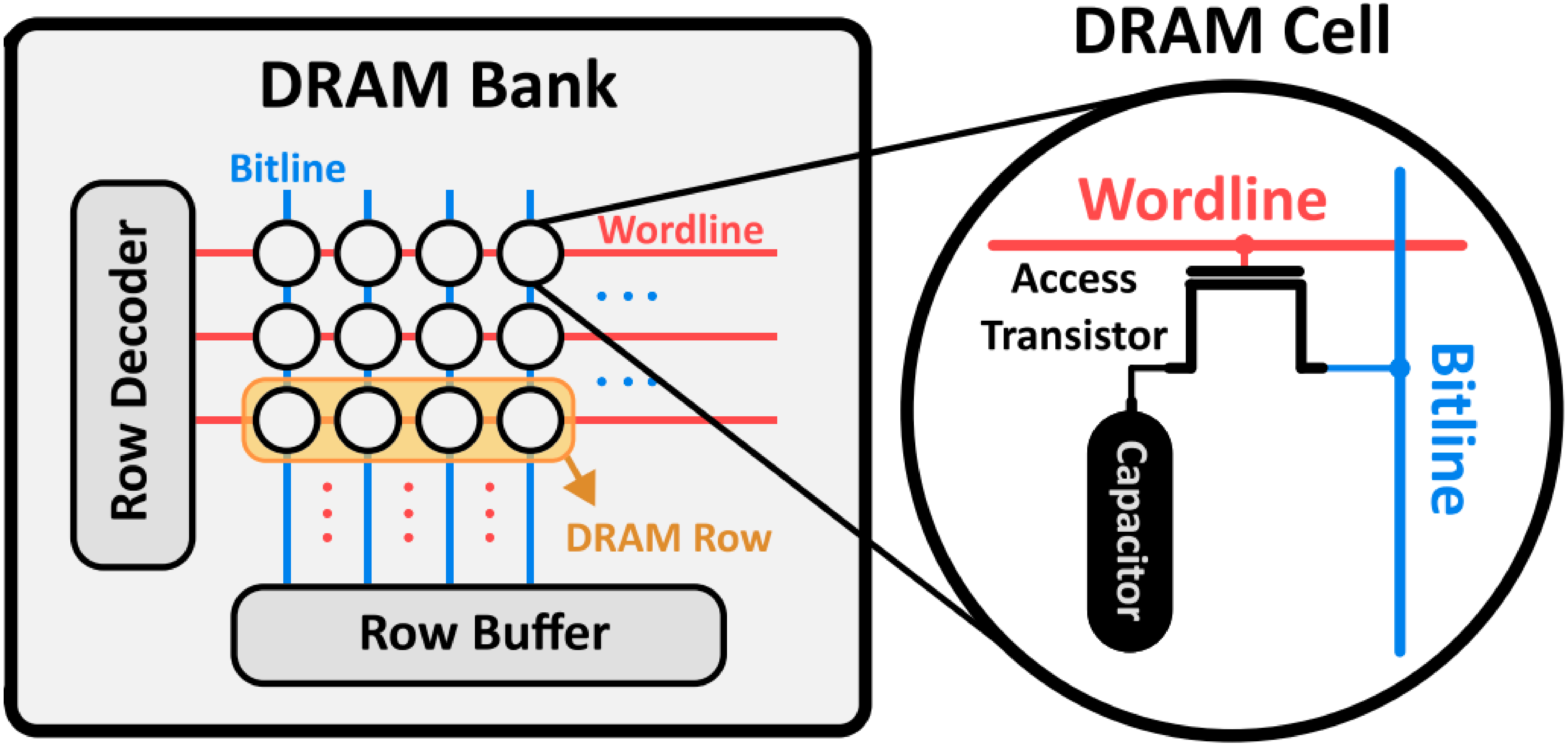
SUMMARY

STRENGTHS & WEAKNESSES

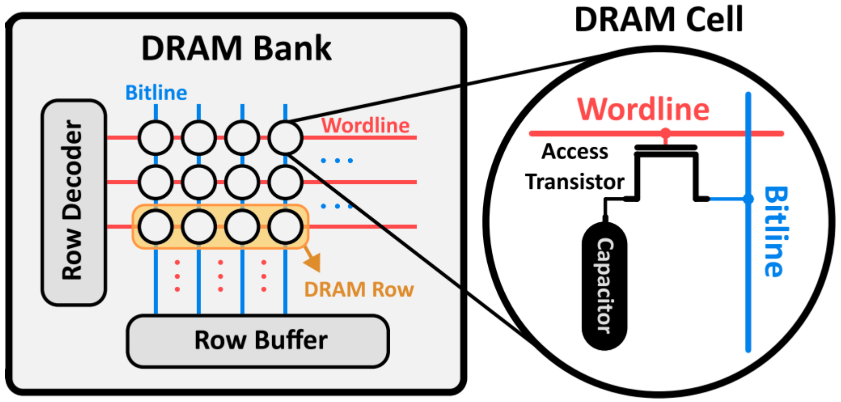
DISCUSSION

Background, Problem & Goal

Recap: DRAM


DRAM Bank
Bitline
Wordline
Row Decoder
DRAM Row
Row Buffer
DRAM Cell
Wordline
Access Transistor
Capacitor
Bitline

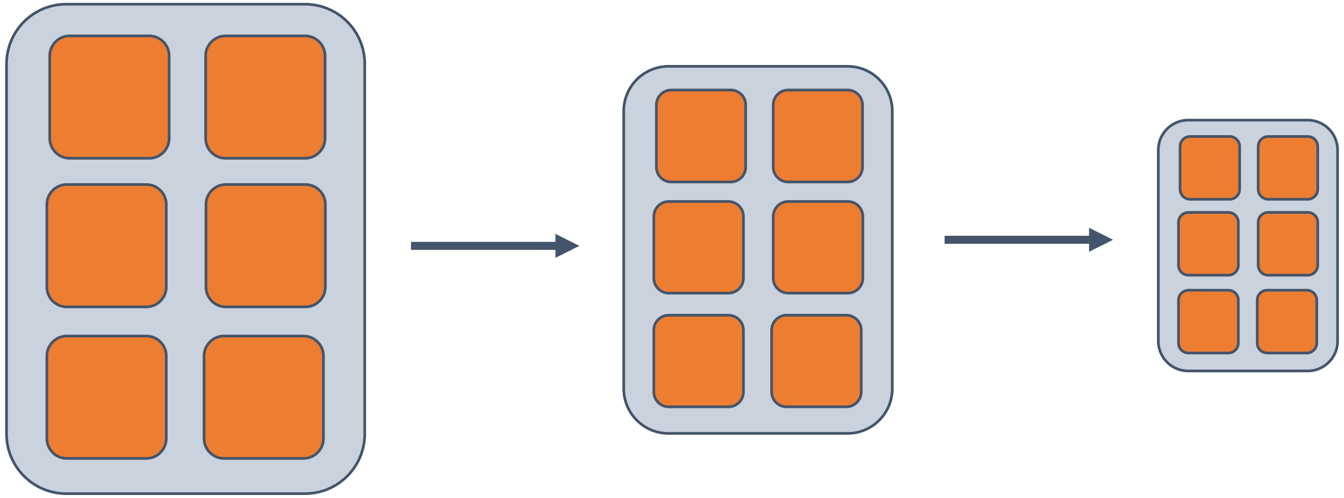
DRAM & RowHammer



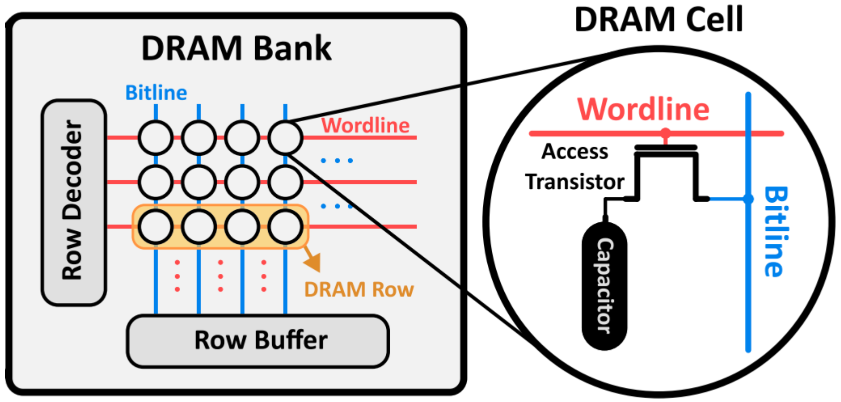
Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing



DRAM & RowHammer



Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing

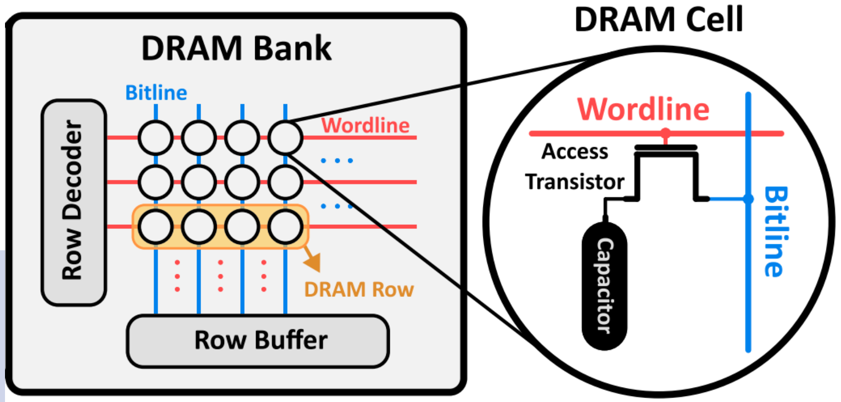
RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

DRAM & RowHammer

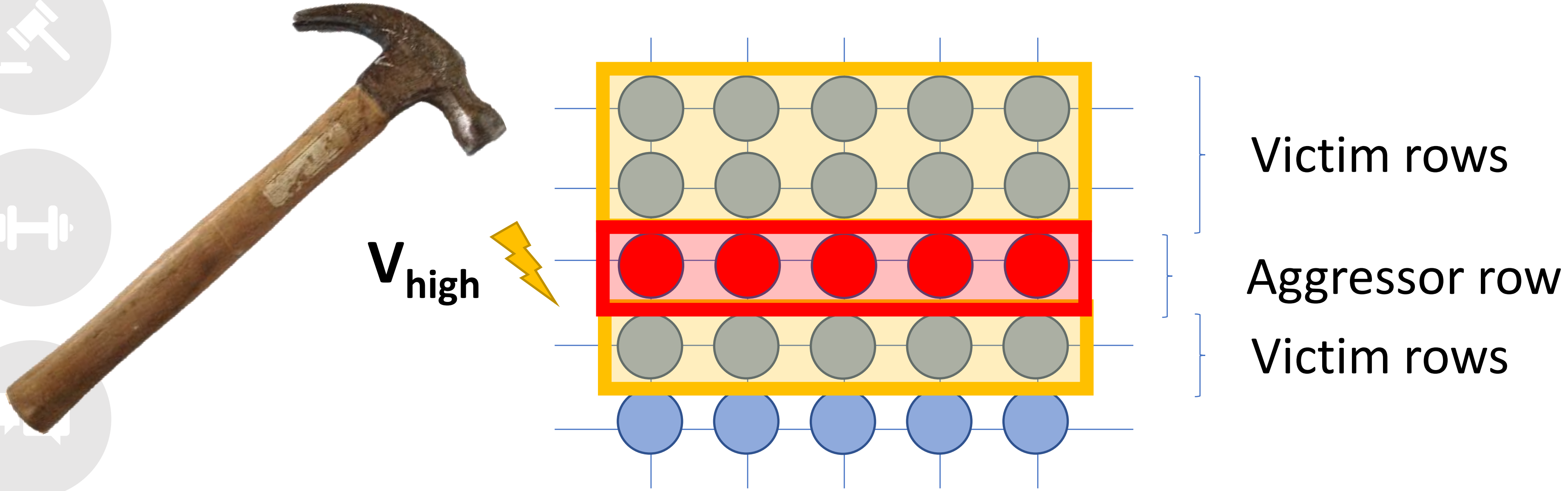
Cause: memory density scaling

↓ DRAM cell size

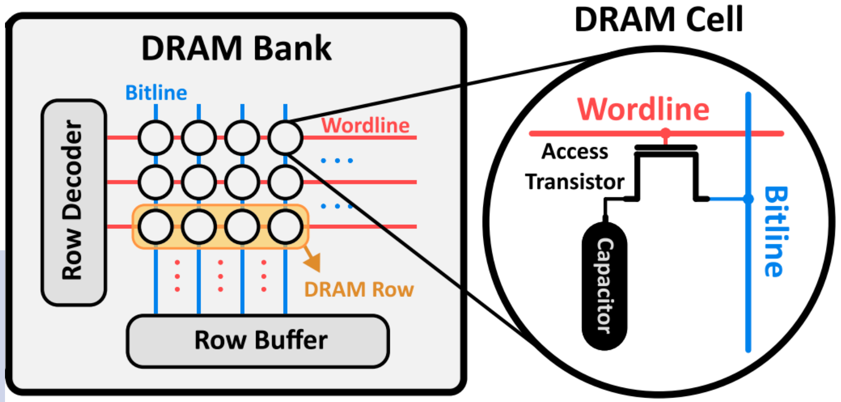
↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows



DRAM & RowHammer

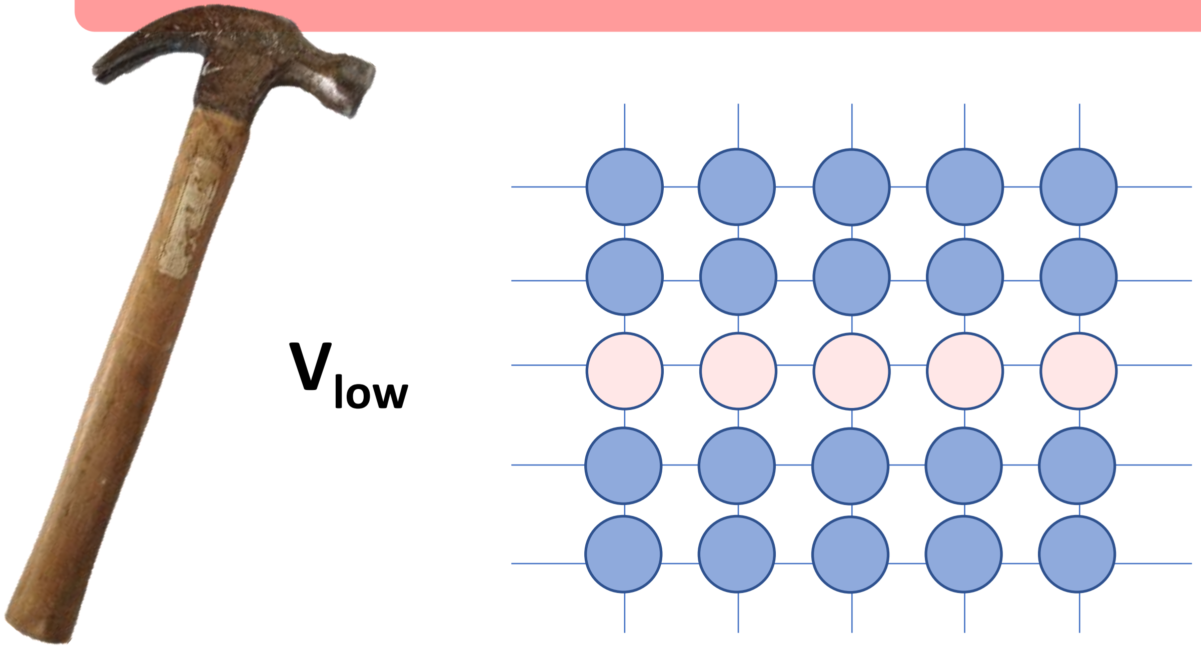


Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing

RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

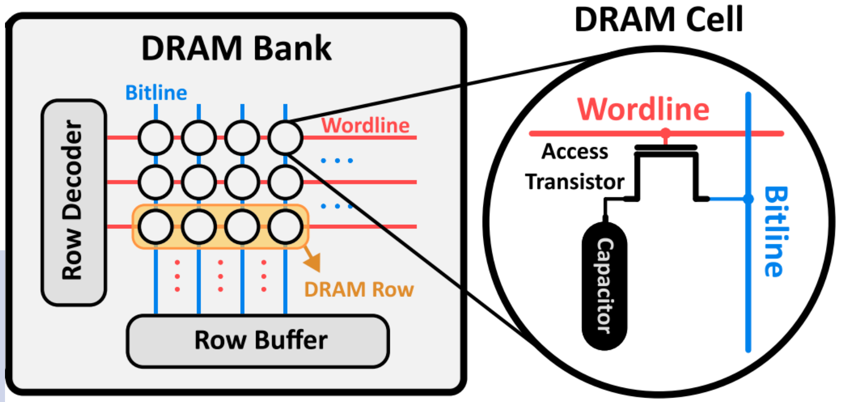


DRAM & RowHammer

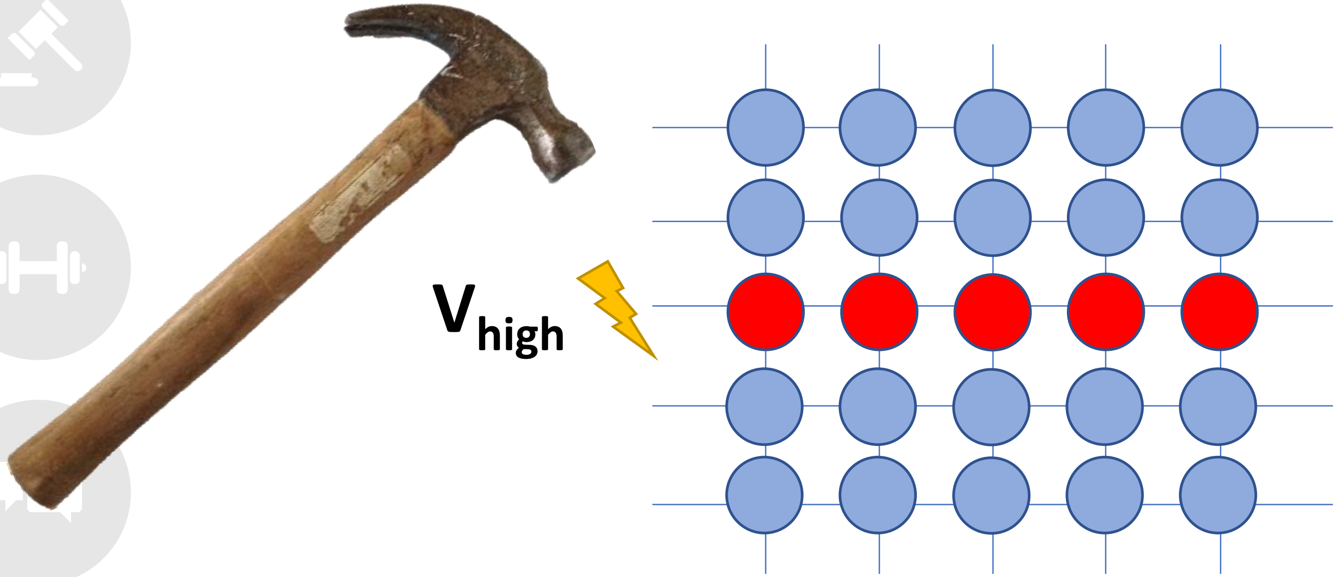
Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

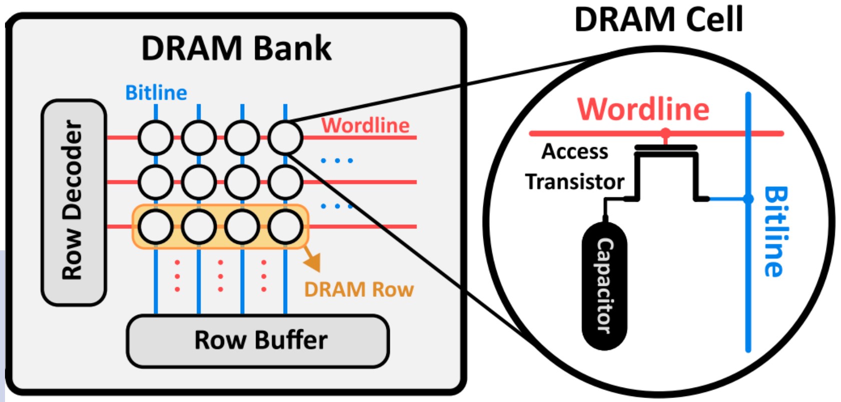


DRAM & RowHammer

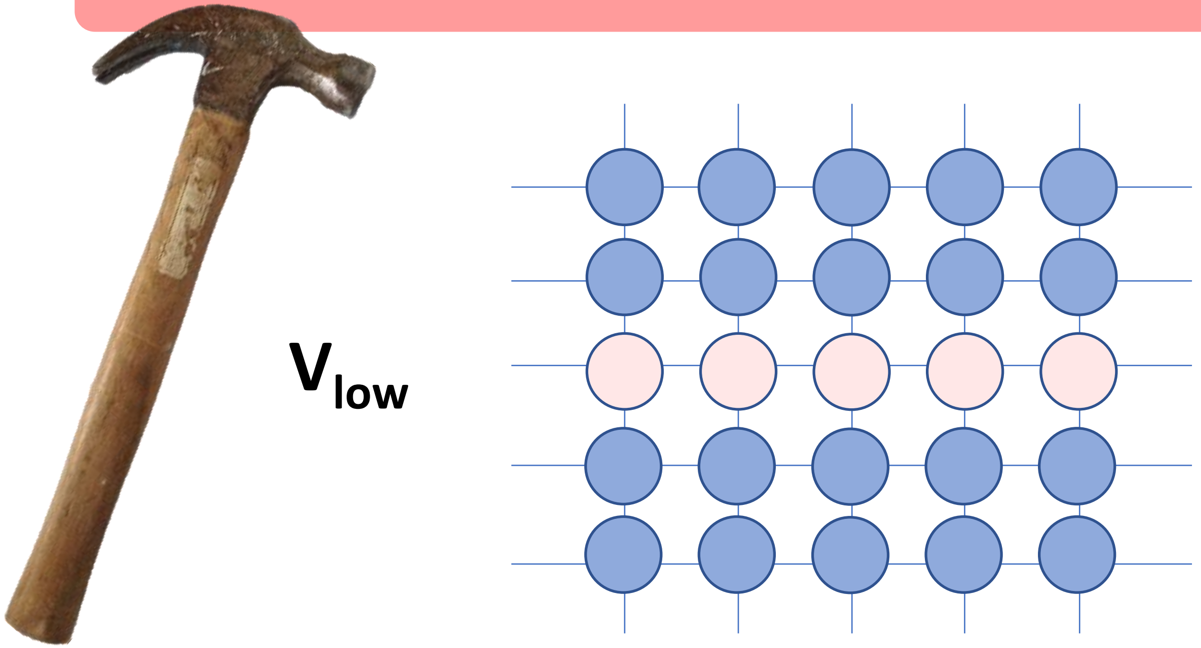
Cause: memory density scaling

↓ DRAM cell size

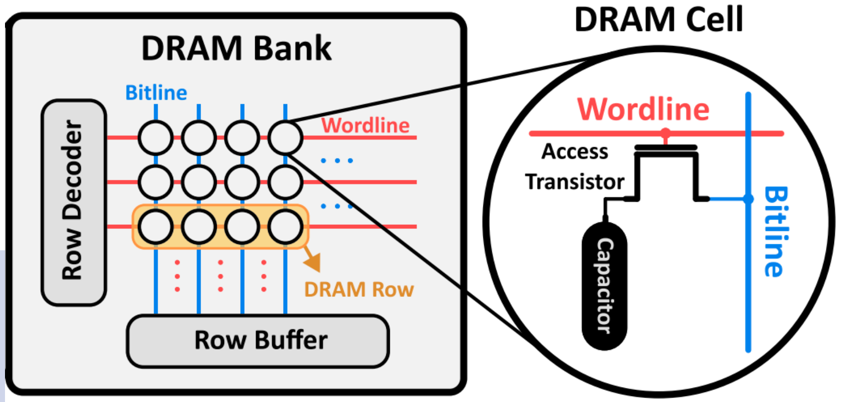
↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows



DRAM & RowHammer

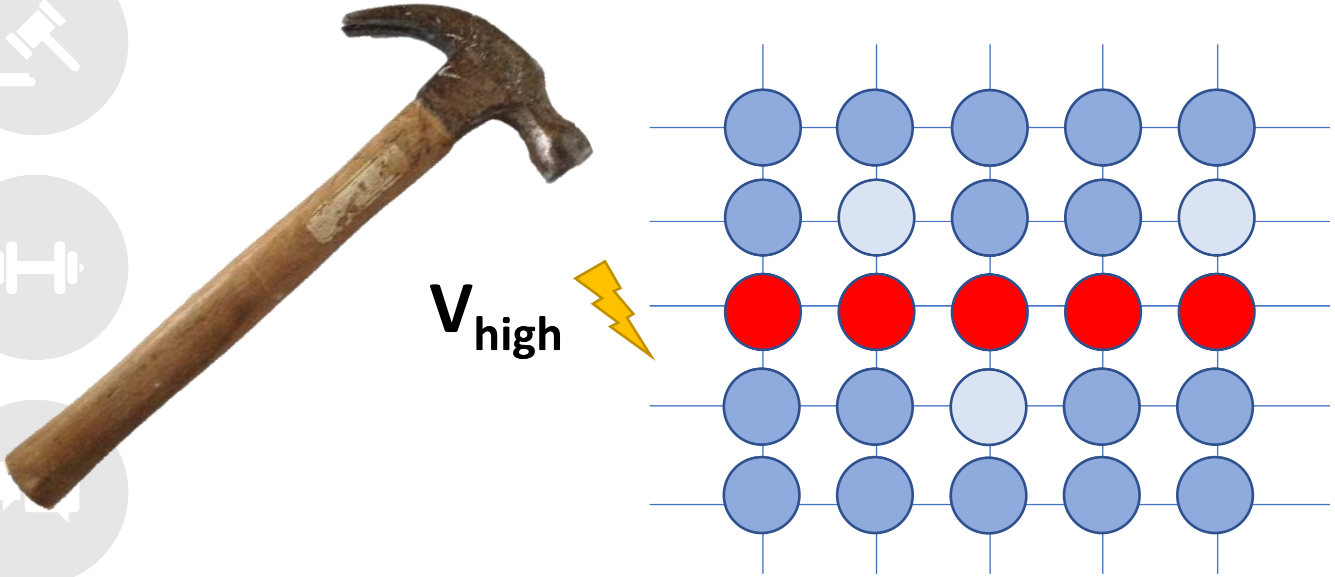


Cause: memory density scaling

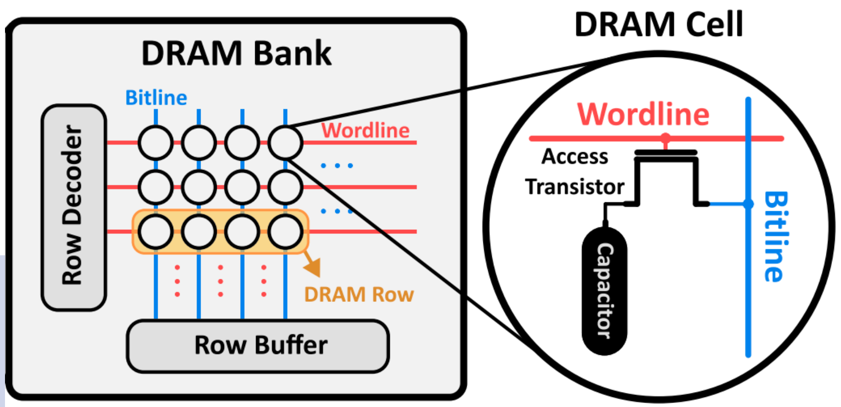
↓ DRAM cell size

↓ cell-to-cell spacing

RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows



DRAM & RowHammer

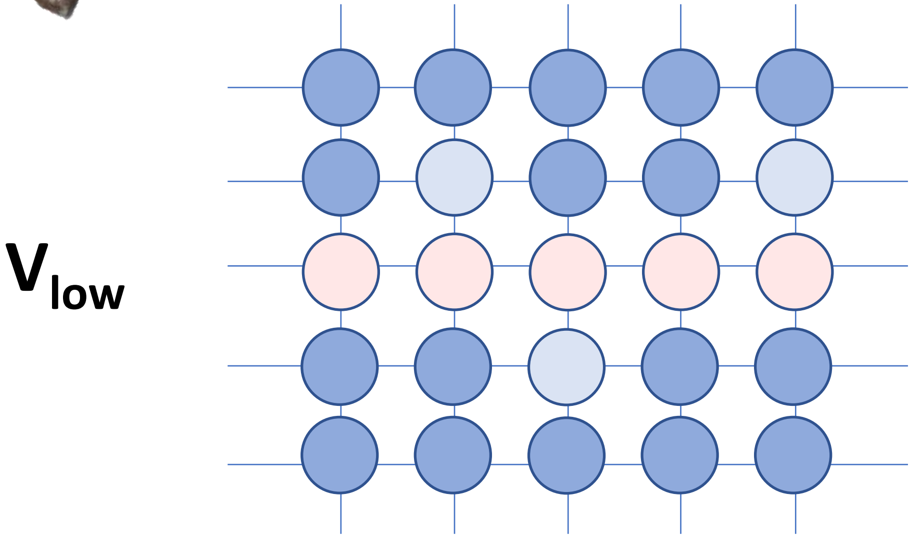


Cause: memory density scaling

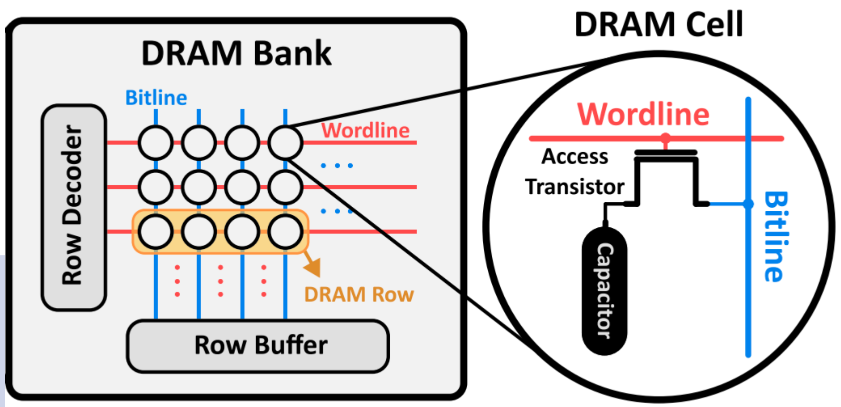
↓ DRAM cell size

↓ cell-to-cell spacing

RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows



DRAM & RowHammer

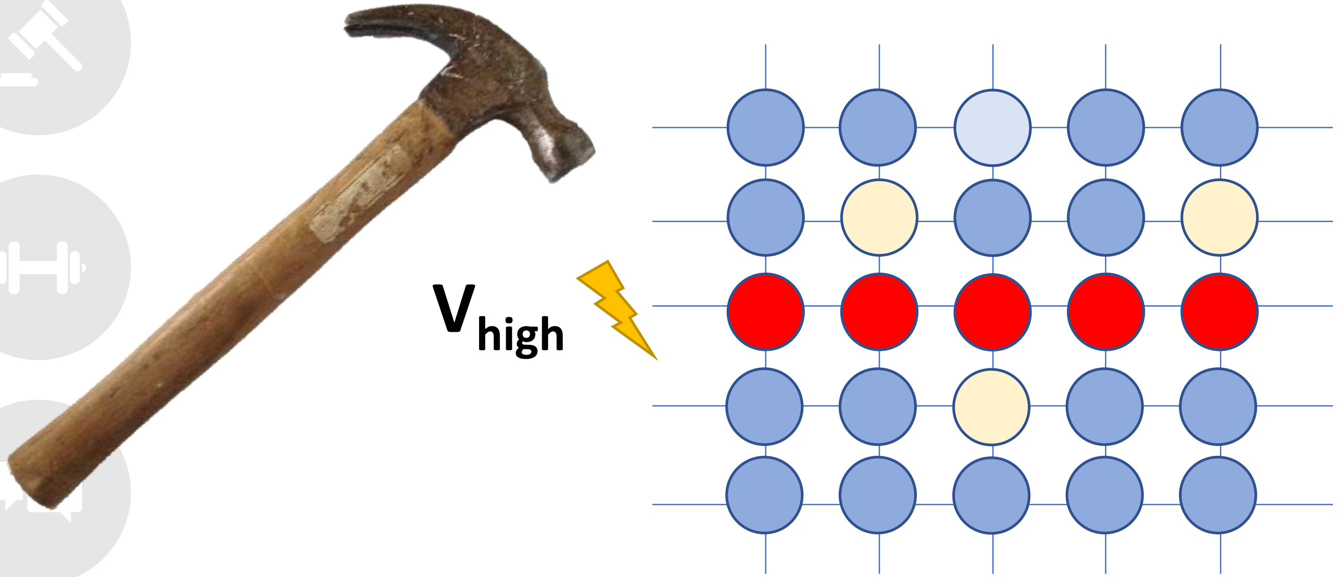


Cause: memory density scaling

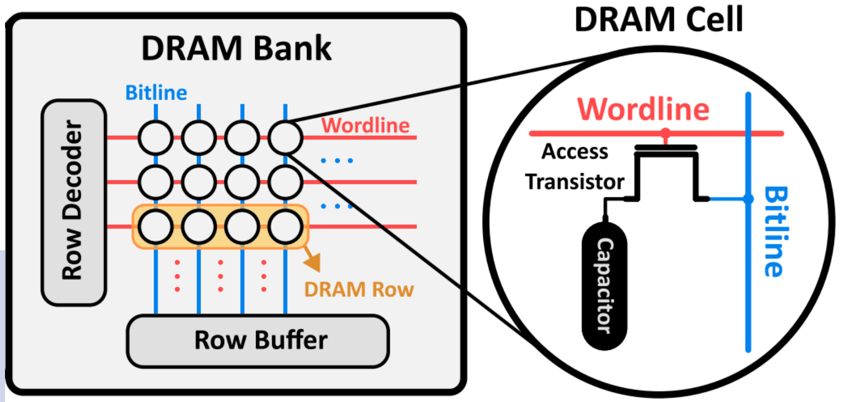
↓ DRAM cell size

↓ cell-to-cell spacing

RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows



DRAM & RowHammer

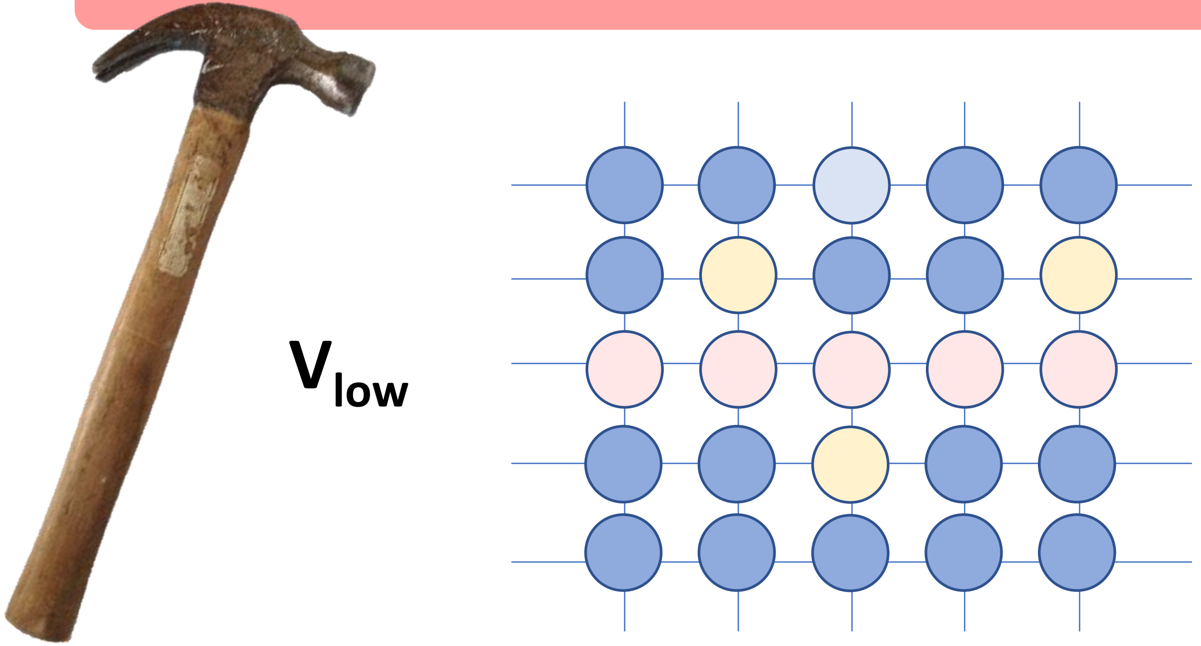


Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing

RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

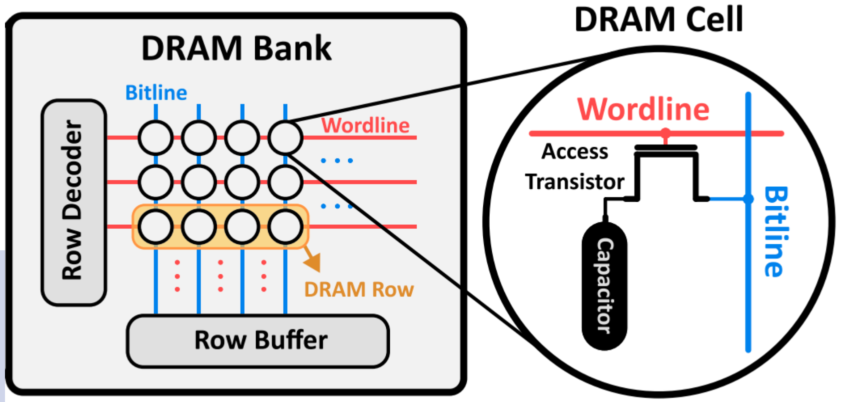


DRAM & RowHammer

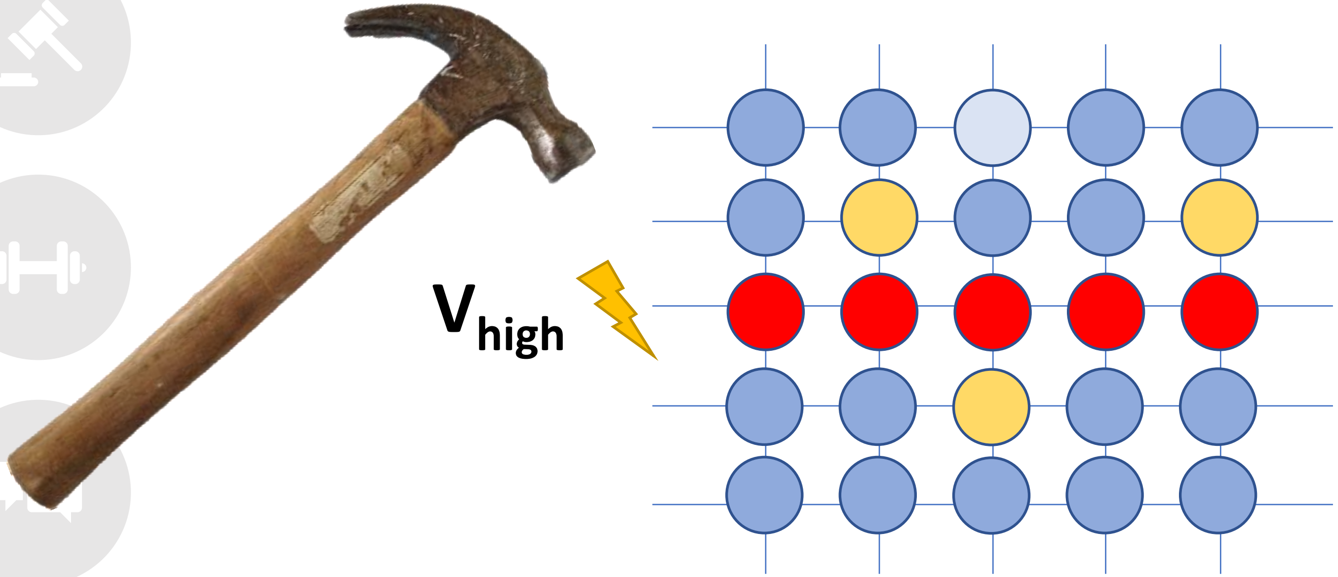
Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

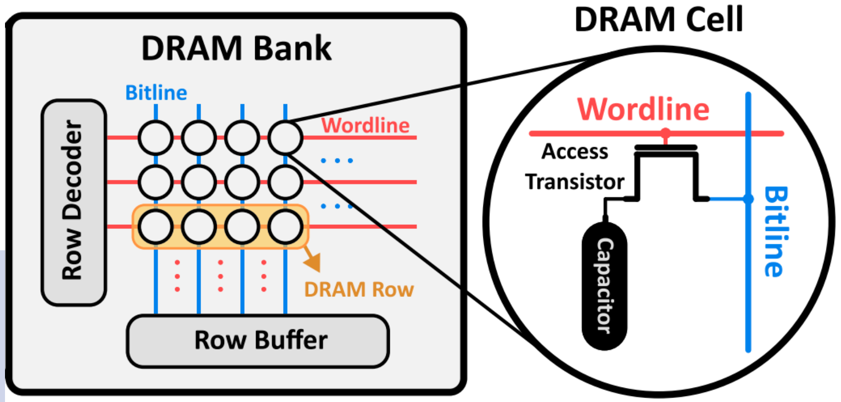


DRAM & RowHammer

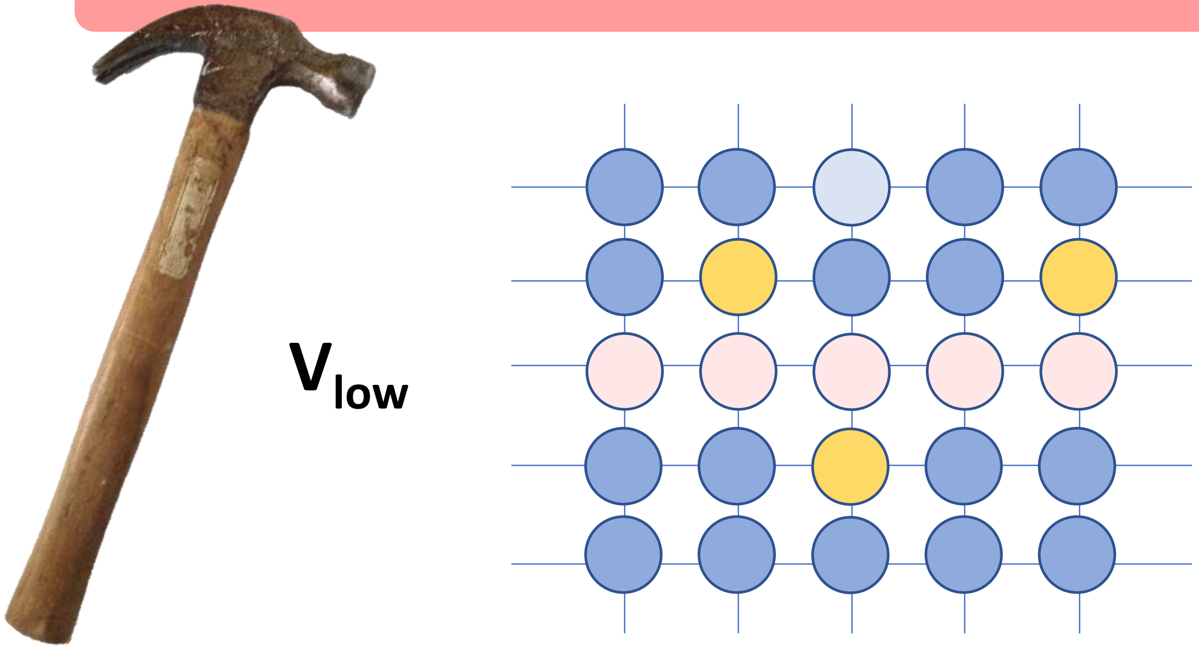
Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

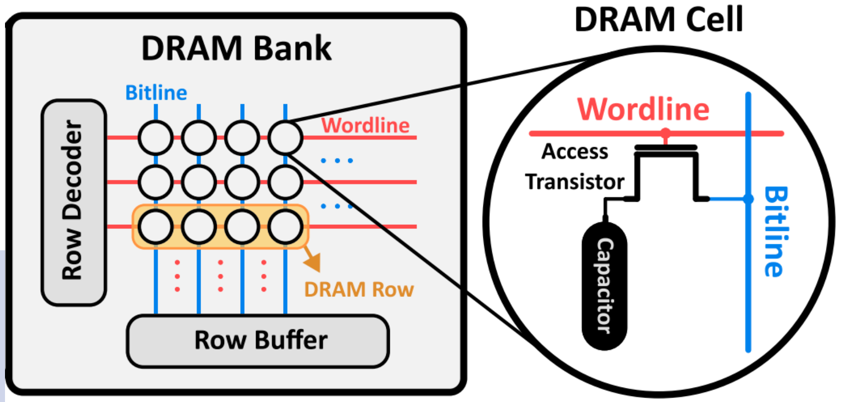


DRAM & RowHammer

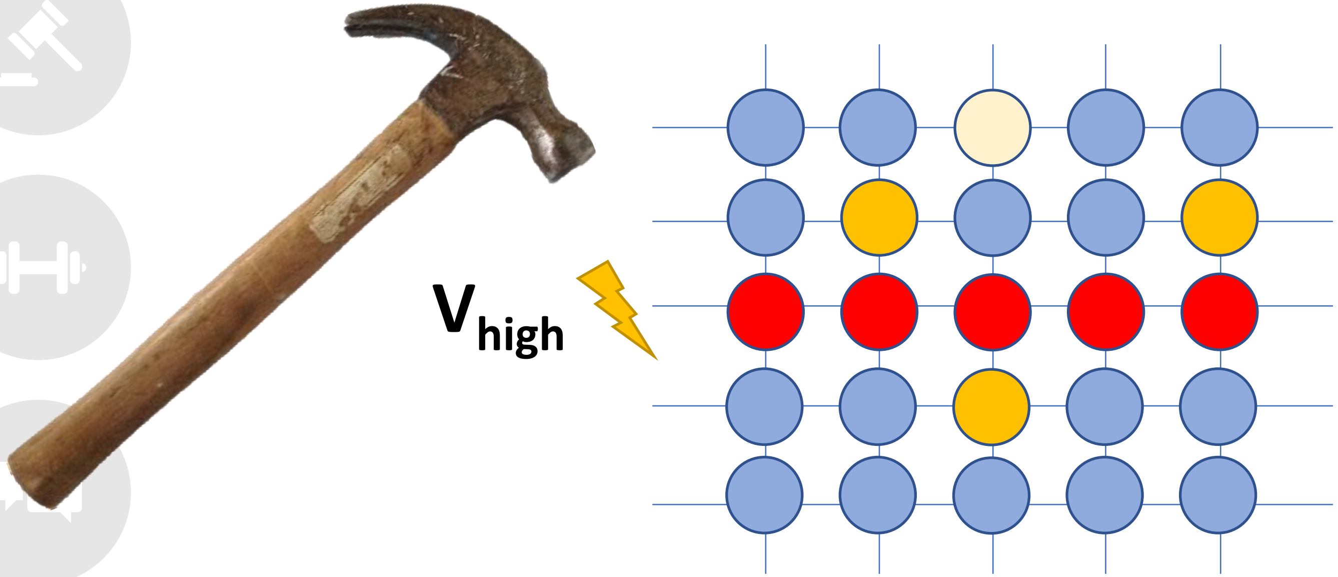
Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

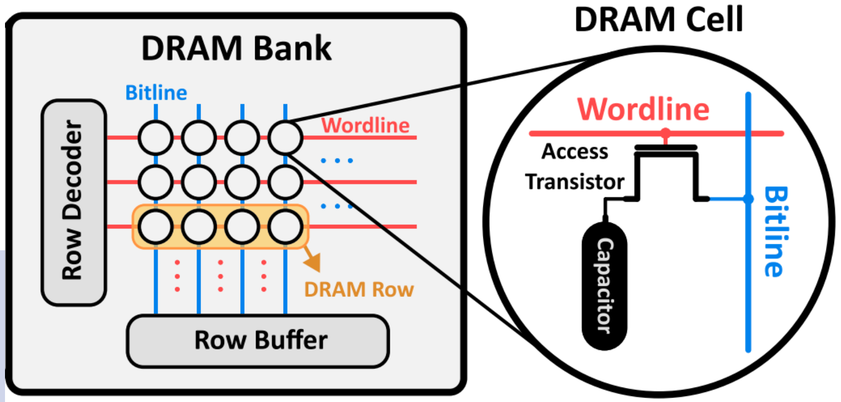


DRAM & RowHammer

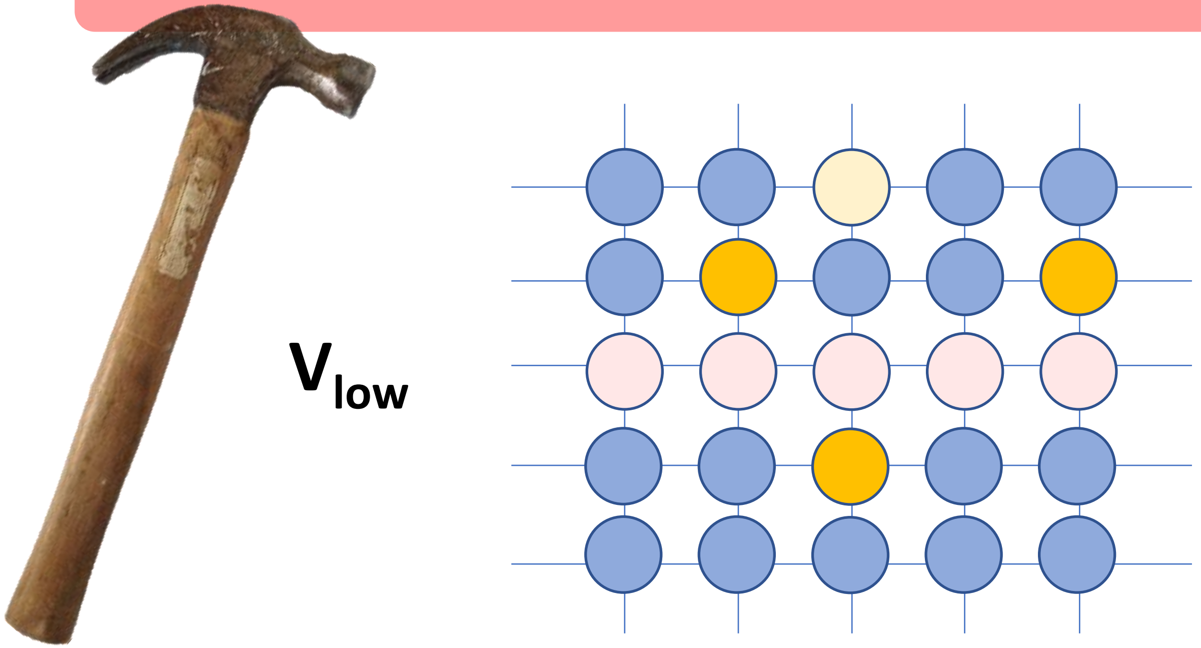
Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

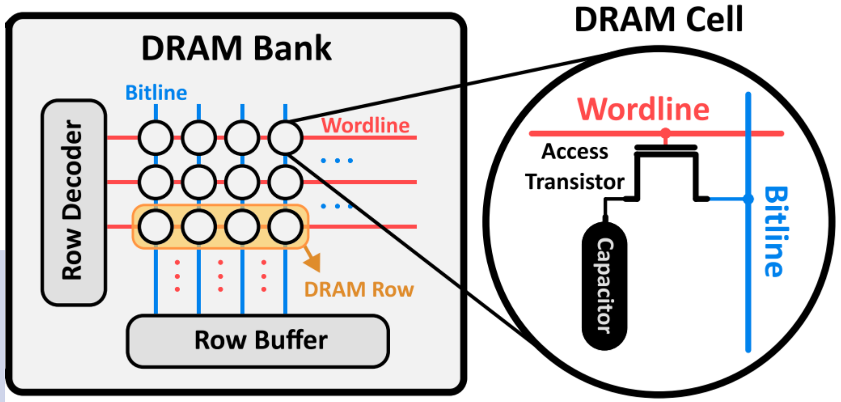


DRAM & RowHammer

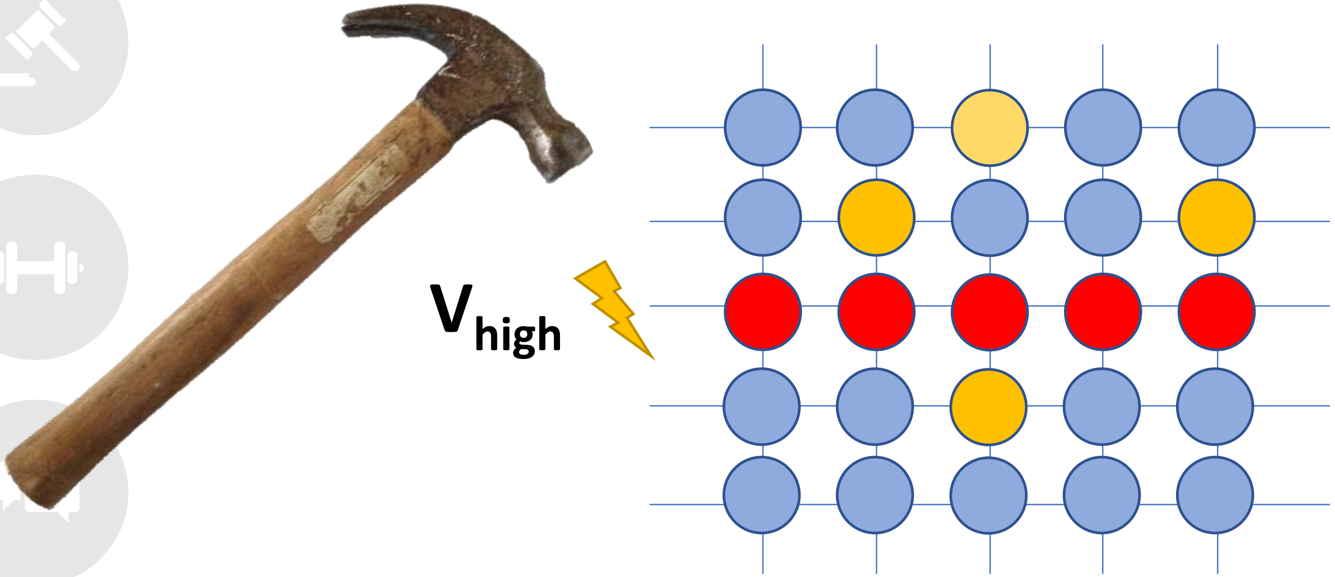
Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

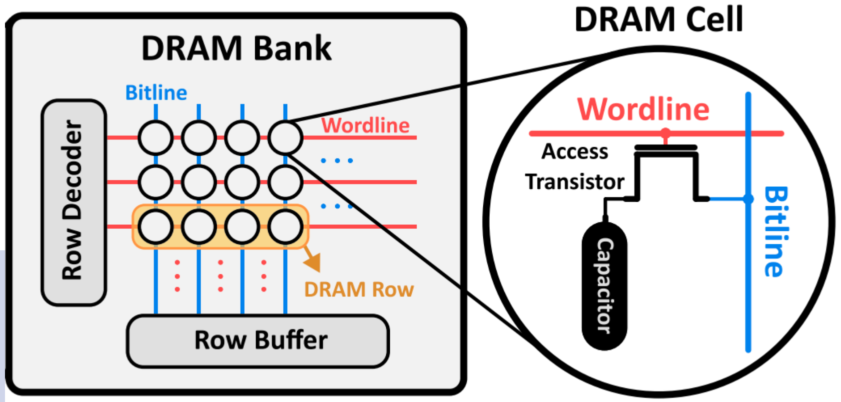


DRAM & RowHammer

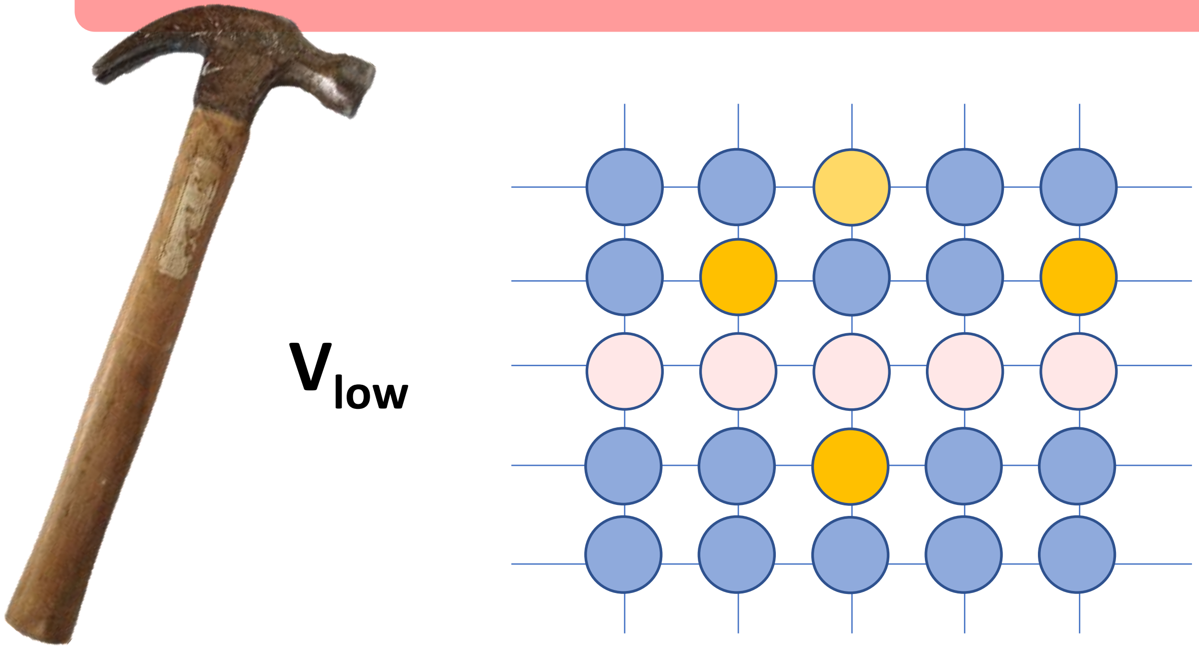
Cause: memory density scaling

↓ DRAM cell size

↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows

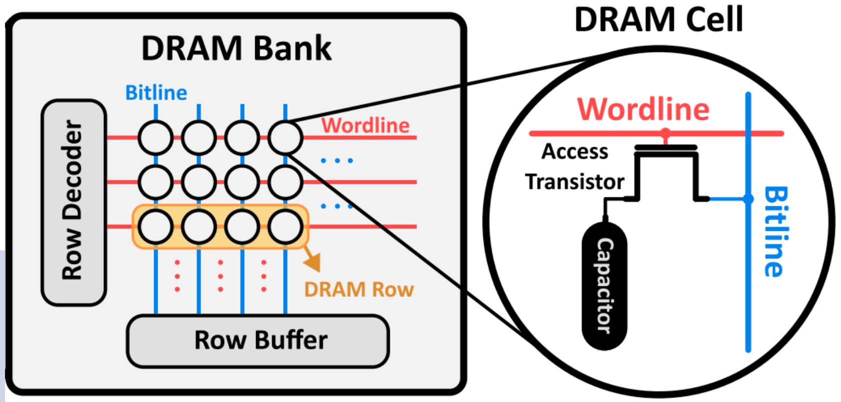


DRAM & RowHammer

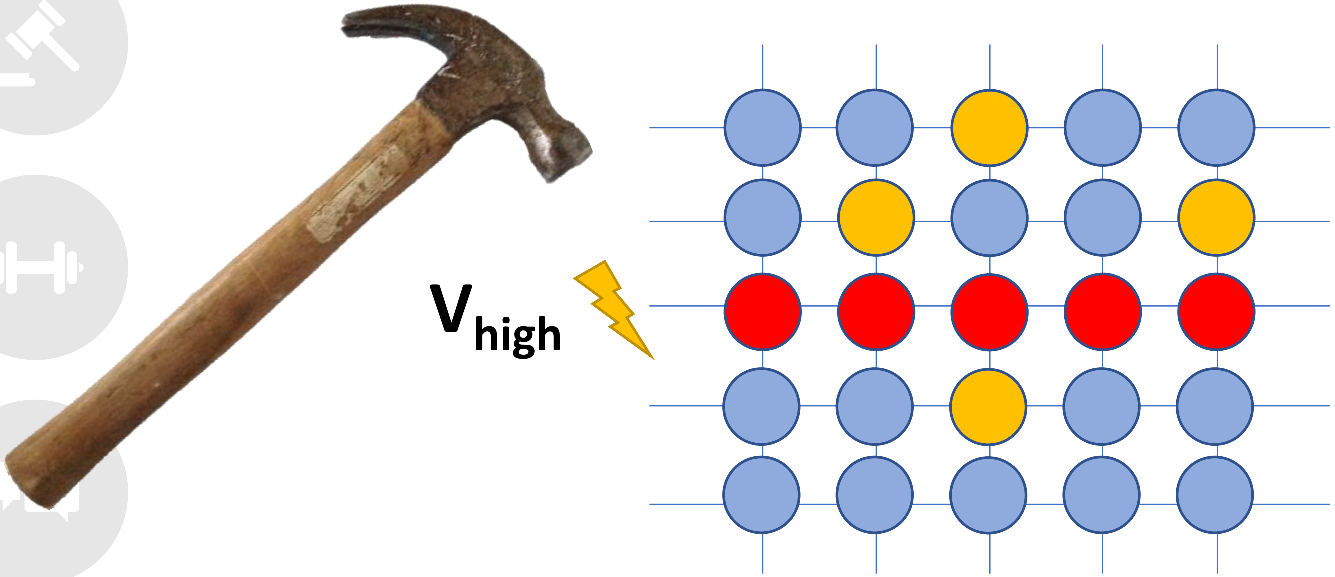
Cause: memory density scaling

↓ DRAM cell size

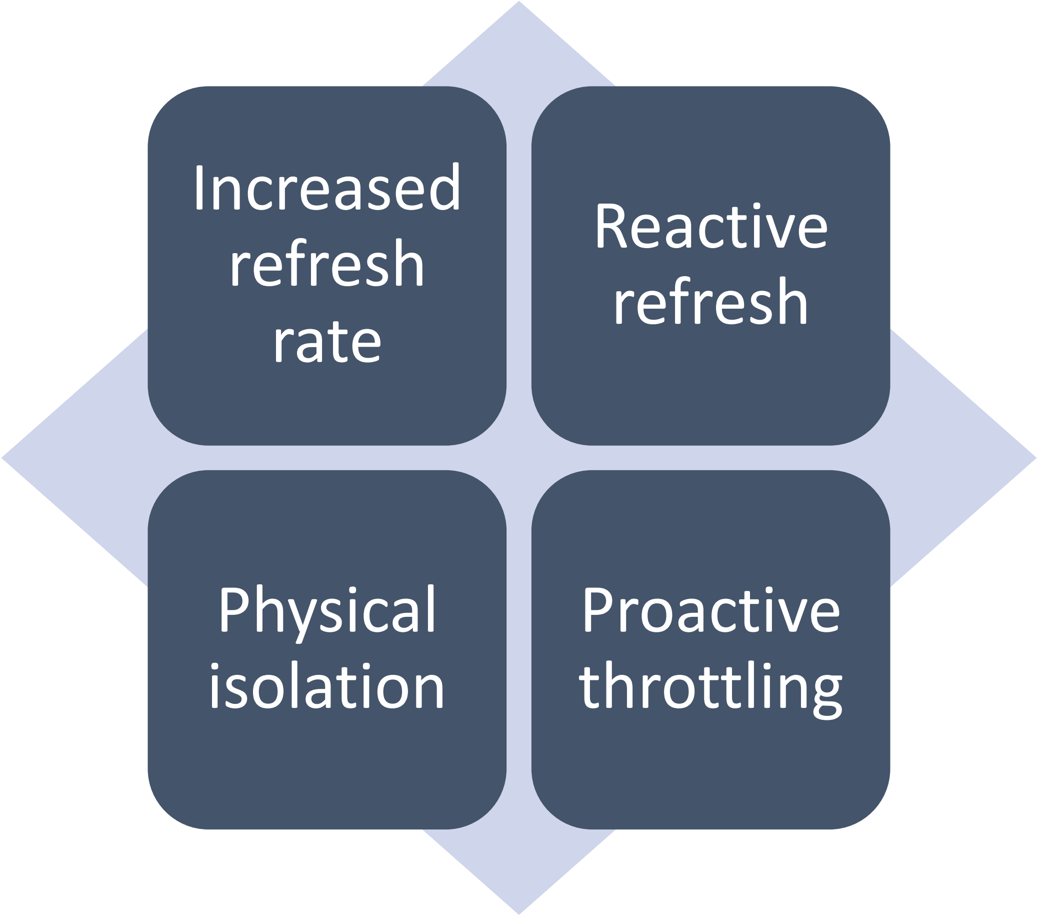
↓ cell-to-cell spacing



RowHammer: rapidly activating (opening) and precharging (closing) DRAM row can cause bit-flips in nearby rows



Current solutions to RowHammer



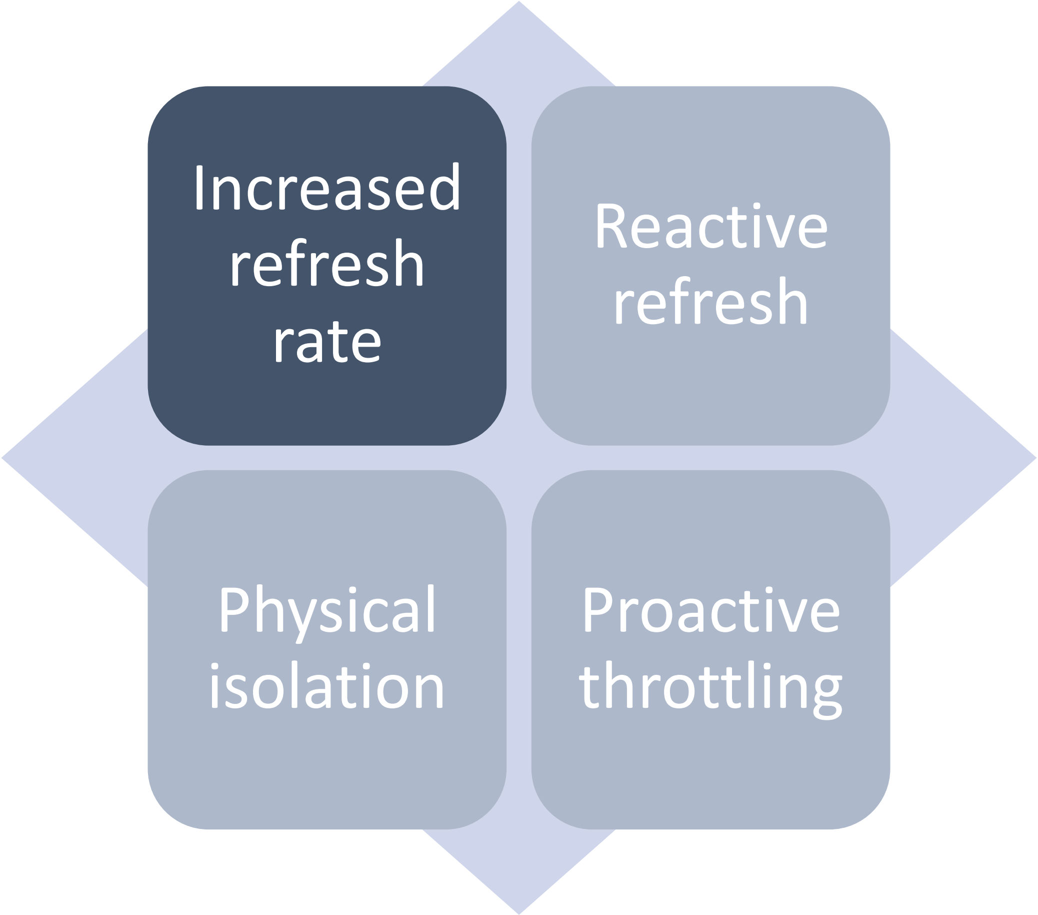
Increased refresh rate

Reactive refresh

Physical isolation

Proactive throttling

Current solutions to RowHammer



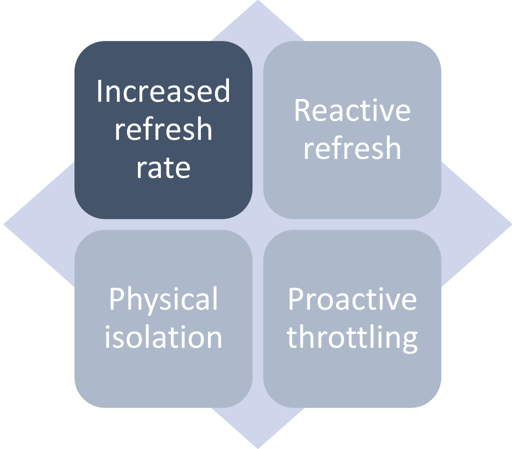
Increased refresh rate

Reactive refresh

Physical isolation

Proactive throttling

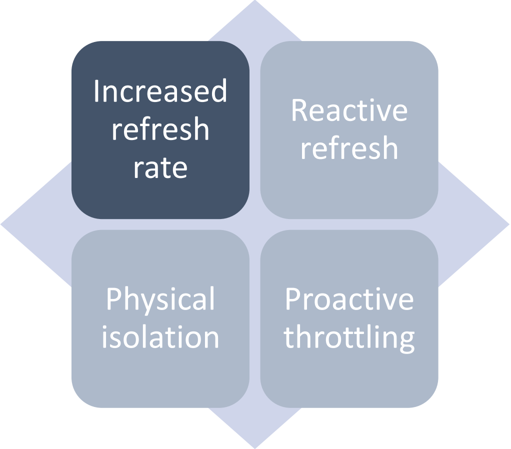
Increased refresh rate

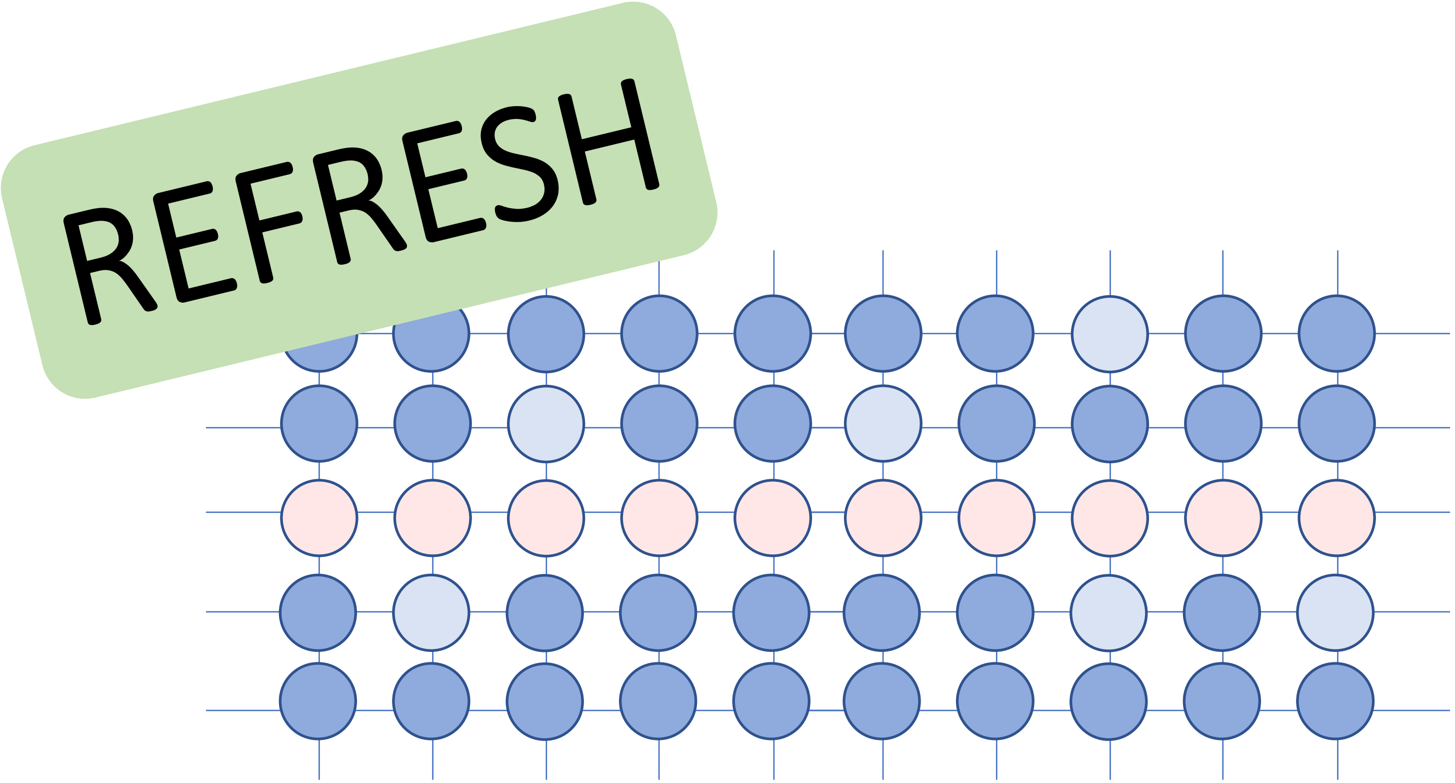


What: refresh (all!) DRAM rows more often to reduce probability of successful bitflip

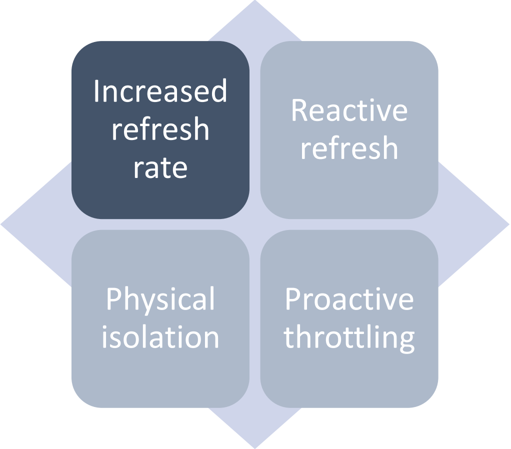
RowHammer (RH) is getting worse: cannot prevent RH without unacceptable performance loss and power consumption increase

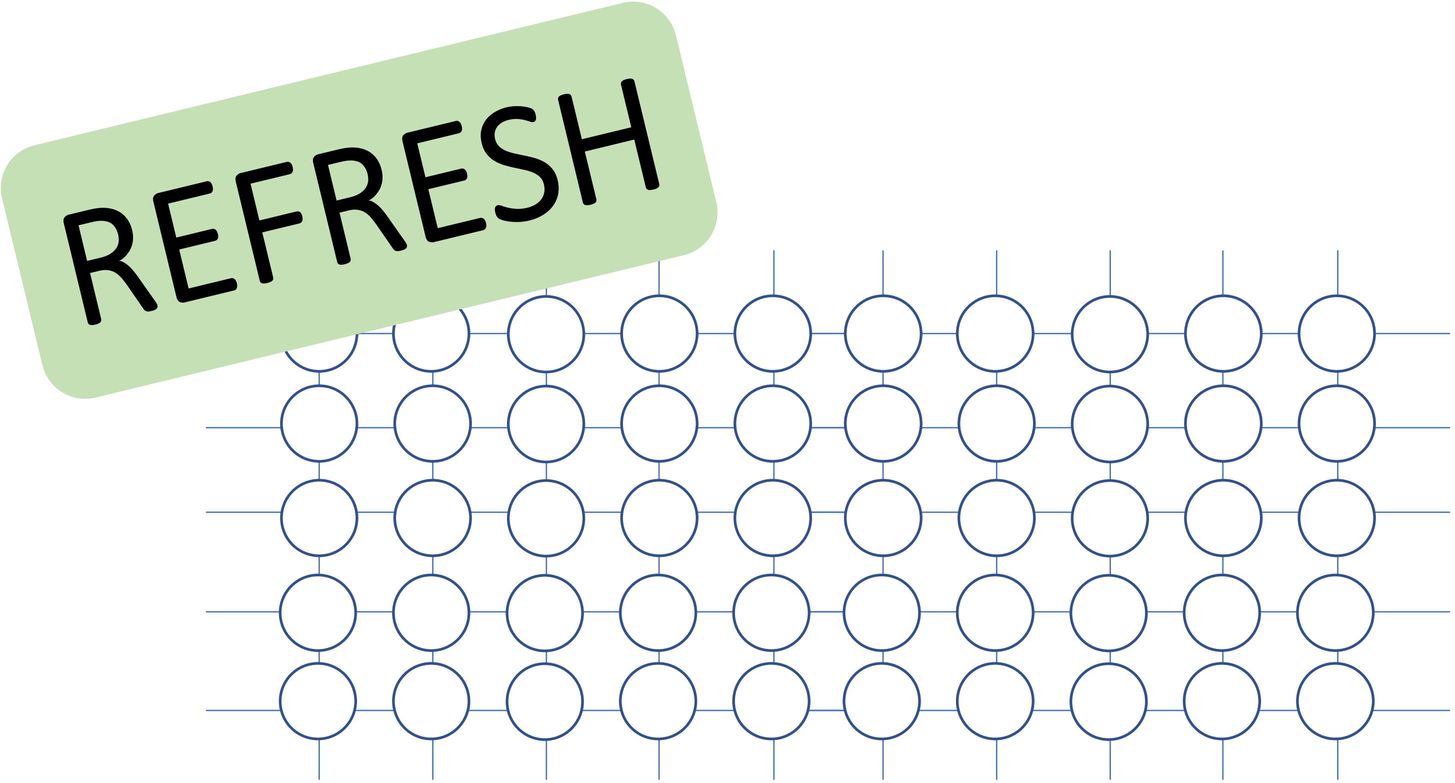
Increased refresh rate



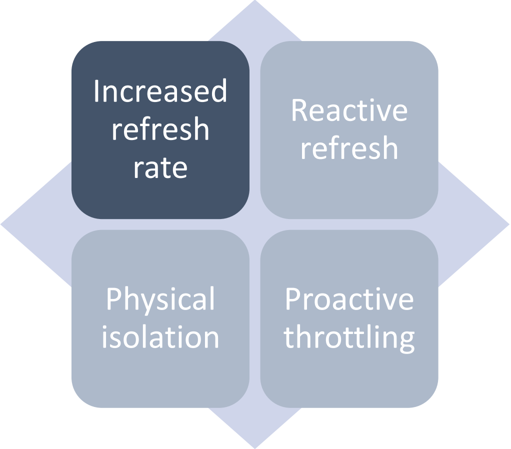


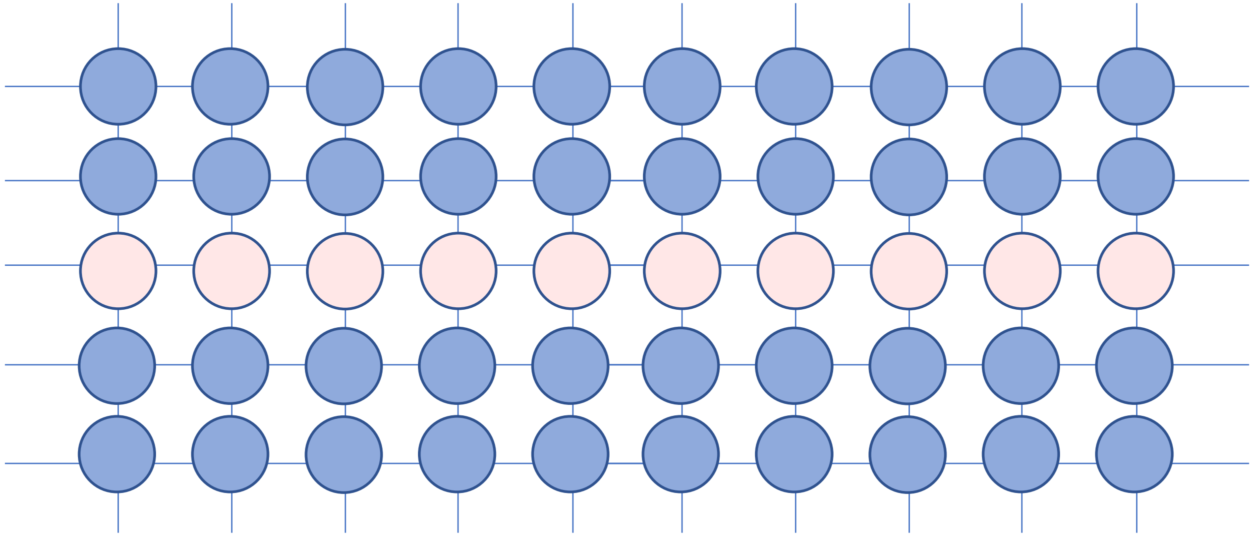
Increased refresh rate



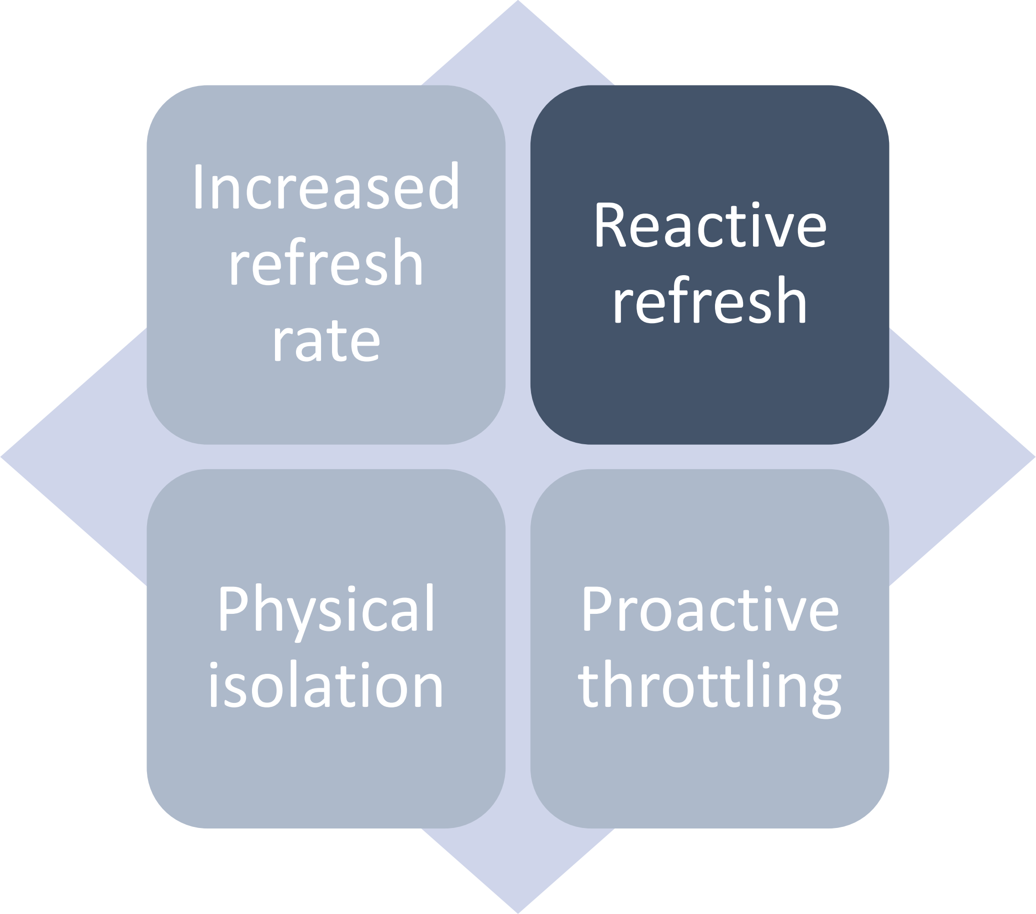


Increased refresh rate





Current solutions to RowHammer



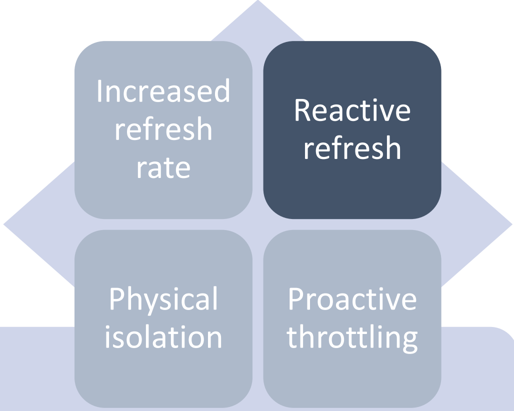
Increased refresh rate

Reactive refresh

Physical isolation

Proactive throttling

Reactive refresh



What: observes activations and reacts by refreshing potential victim rows

e.g., TWiCe, PARA, ProHIT, MRLoc, CAT, CBT, …

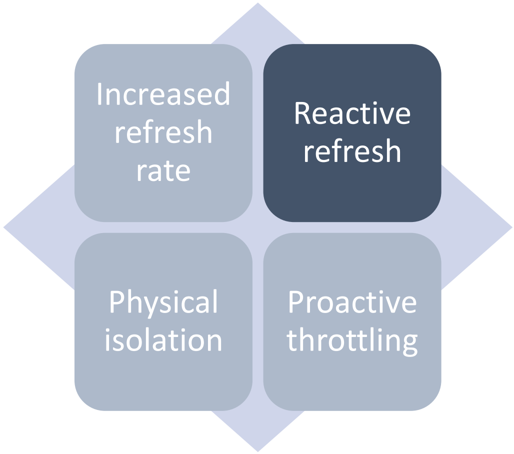
Requires proprietary knowledge on DRAM internals: need to know which rows are adjacent to aggressor rows

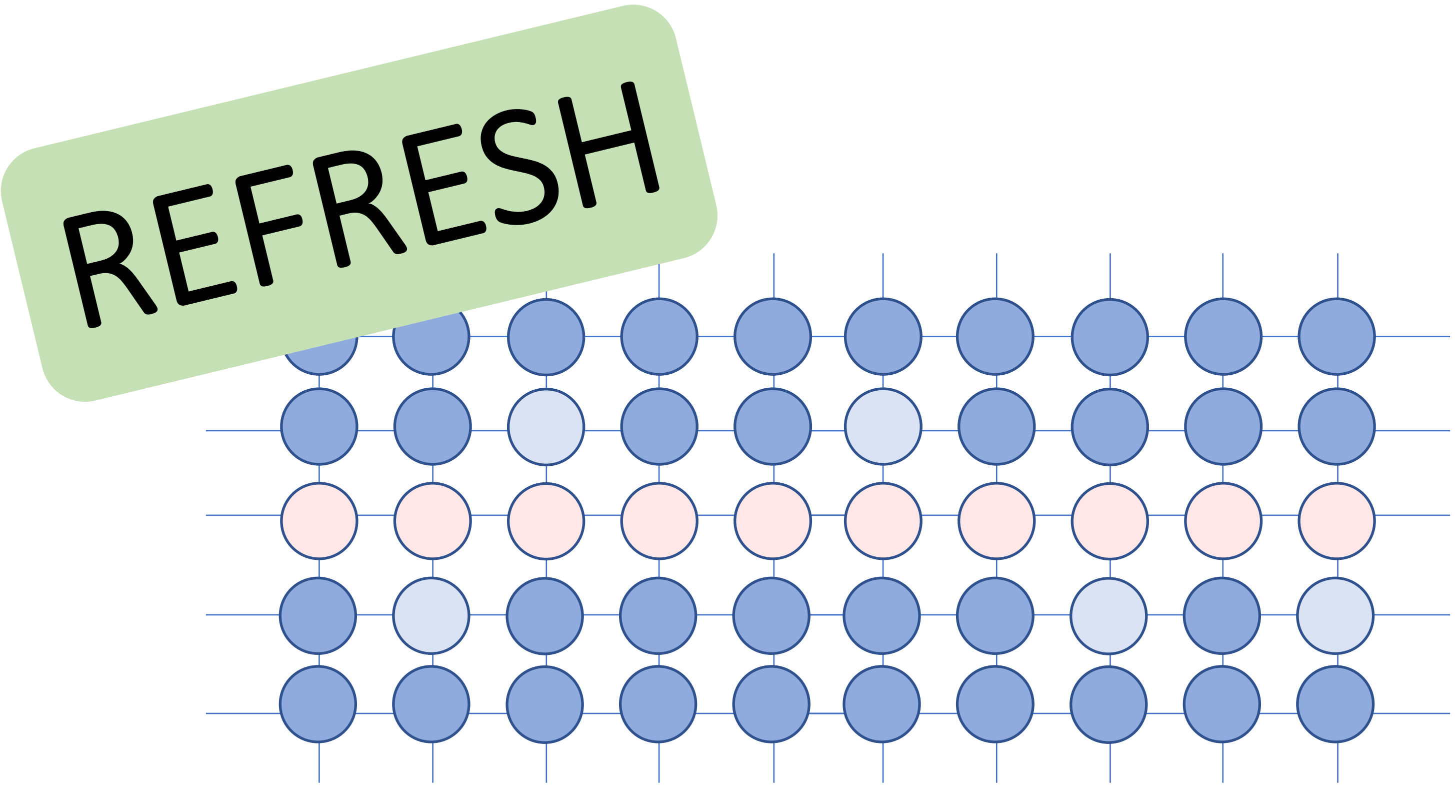
- Faulty rows/cells/columns
- Differences in access latency of fastest & slowest cell

Wang, Minghua, et al. "DRAMDig: a knowledge-assisted tool to uncover DRAM address mapping." *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020.

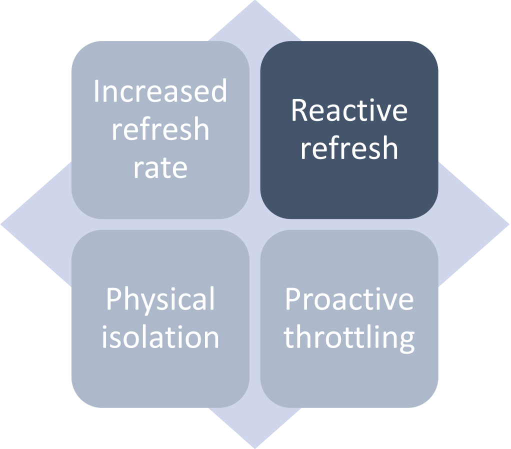
Some are probabilistic methods: do not prevent RowHammer completely

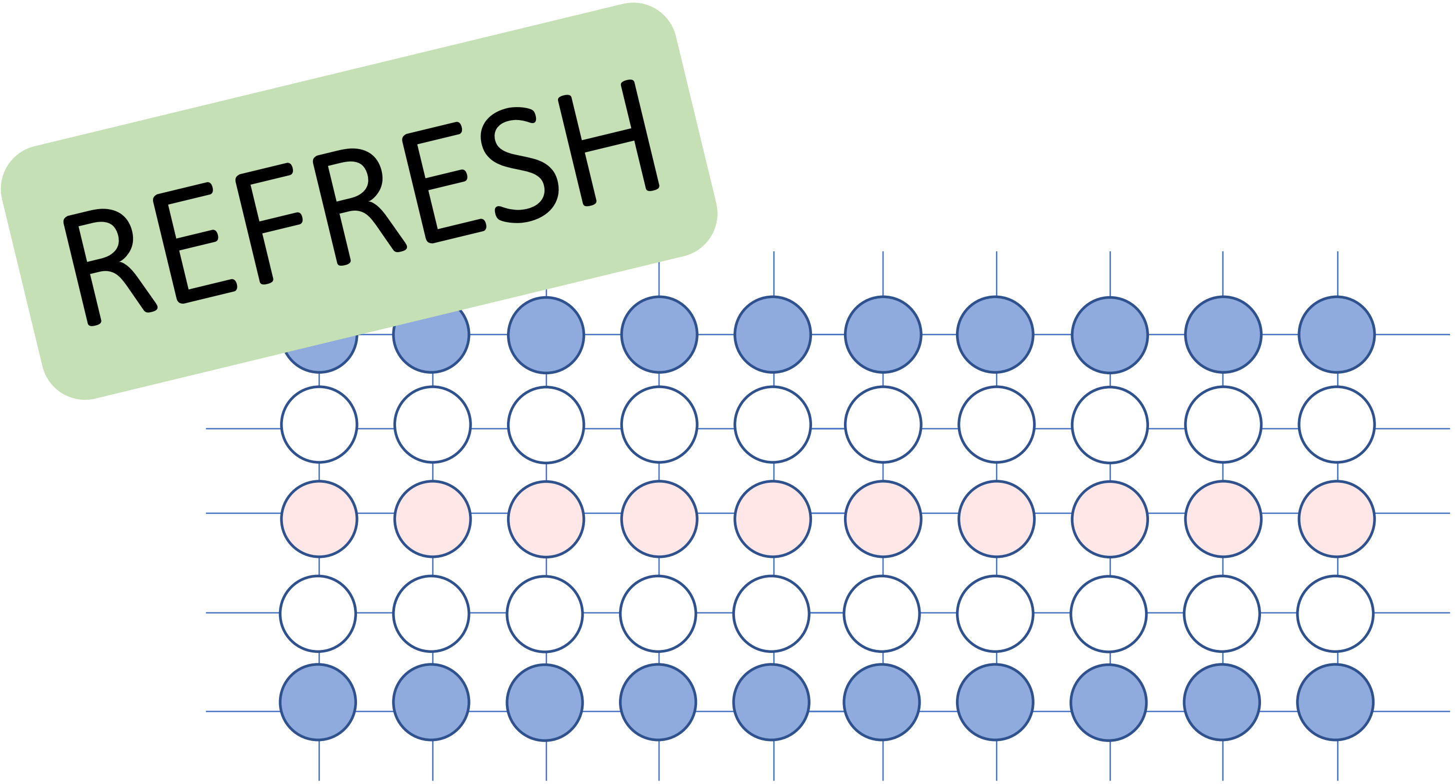
Reactive refresh



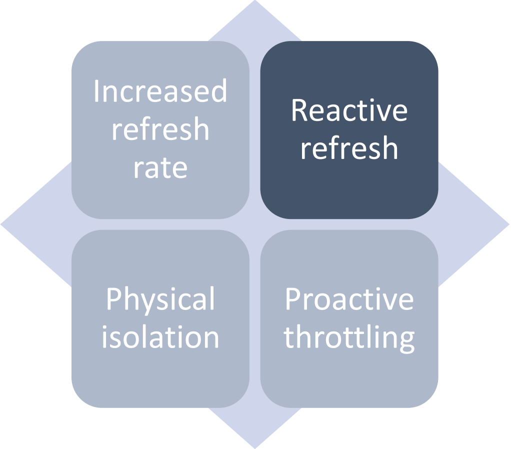


Reactive refresh





Reactive refresh

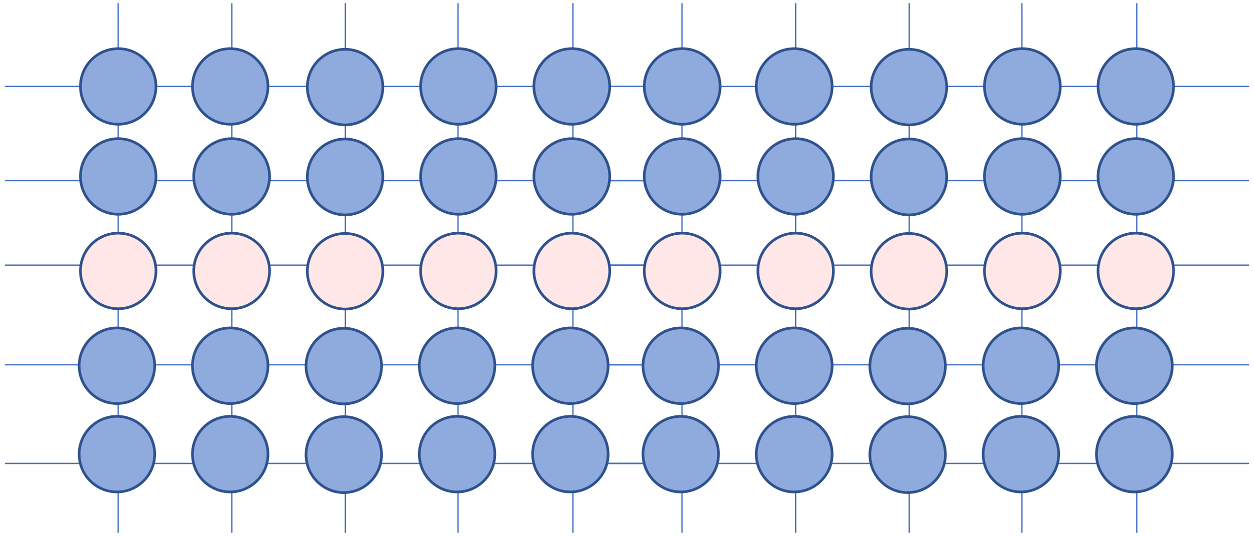


Increased refresh rate

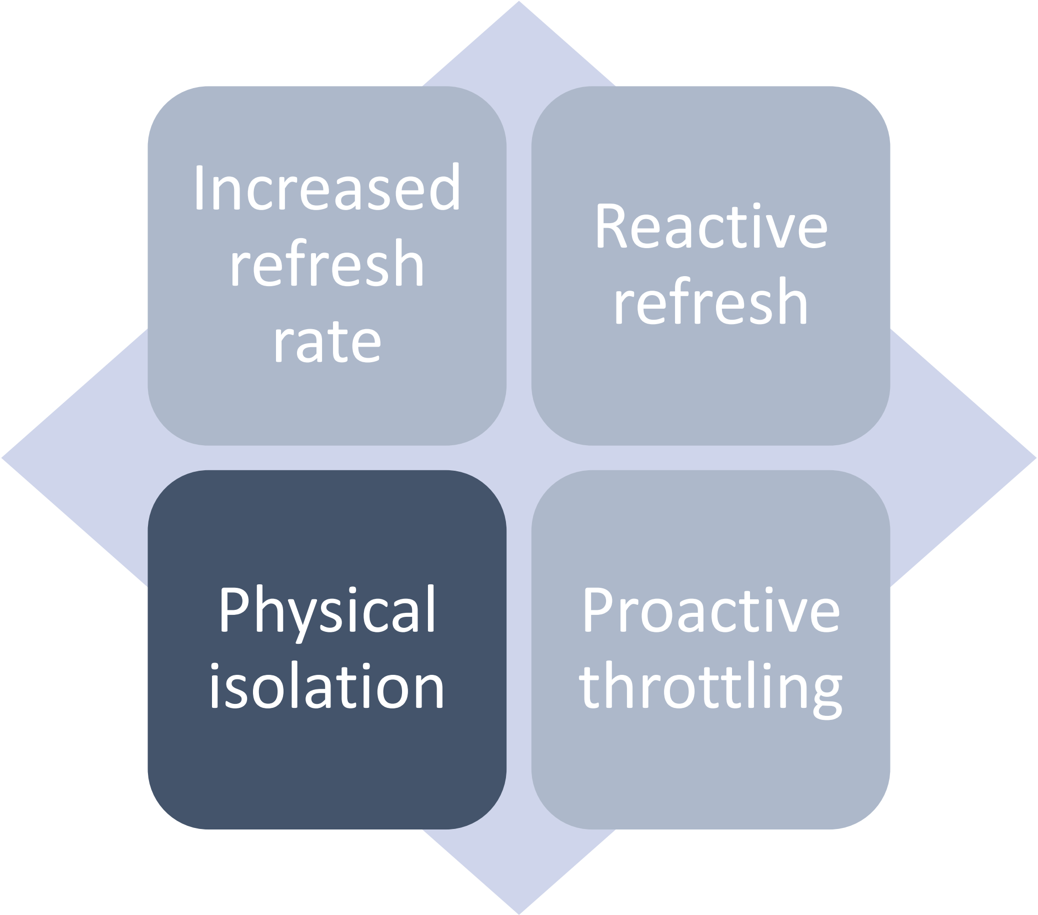
Reactive refresh

Physical isolation

Proactive throttling



Current solutions to RowHammer



Increased refresh rate

Reactive refresh

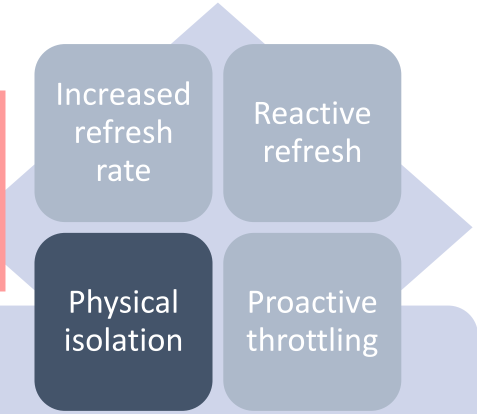
Physical isolation

Proactive throttling

Physical isolation

Already defeated!

PTHammer, opcode flipping, …



What: separates physically sensitive da

e.g., by adding buffer r betw een (ZebRAM)

e.g., by separating me ory ws of user and kernel mode (CATT)

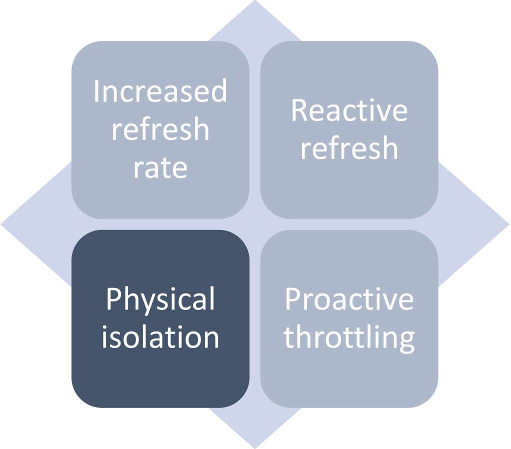
RowHammer is getting worse: we need to provide greater isolation

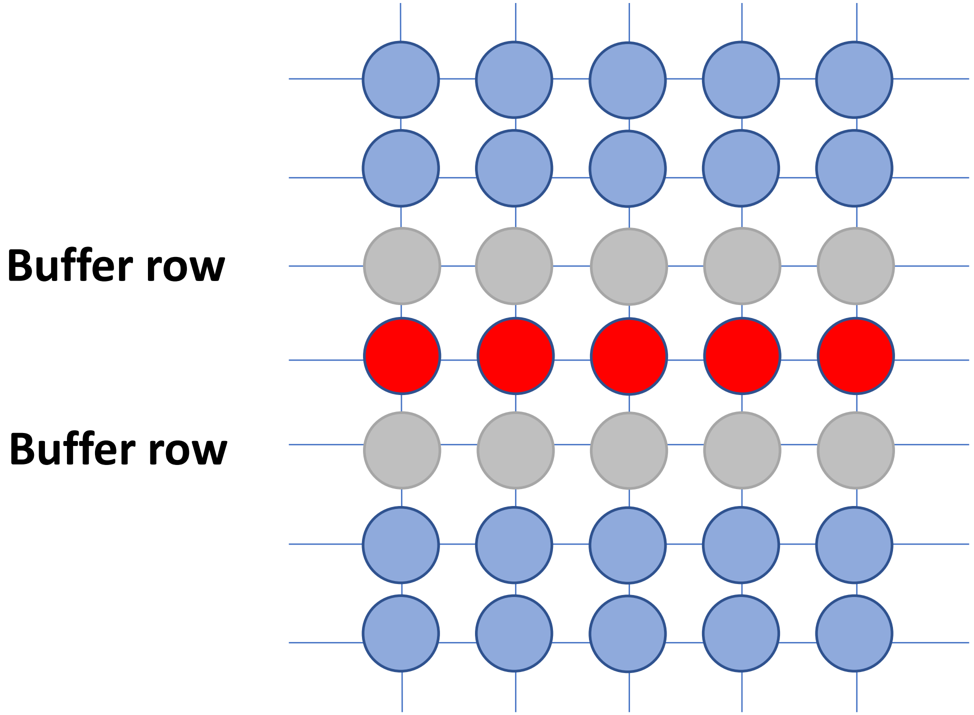
- wastes memory capacity
- reduces fraction of cells we can protect from RH

Requires proprietary knowledge on DRAM internals: need to know which rows are adjacent to aggressor rows

- Faulty rows/cells/columns
- Differences in access latency of fastest & slowest cell

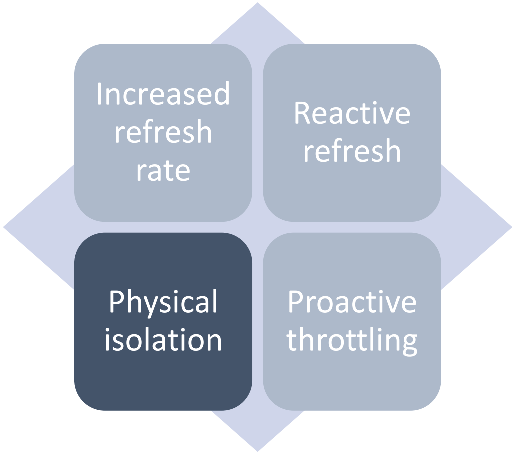
Physical isolation





Buffer row or guard row or isolation row or …

Physical isolation

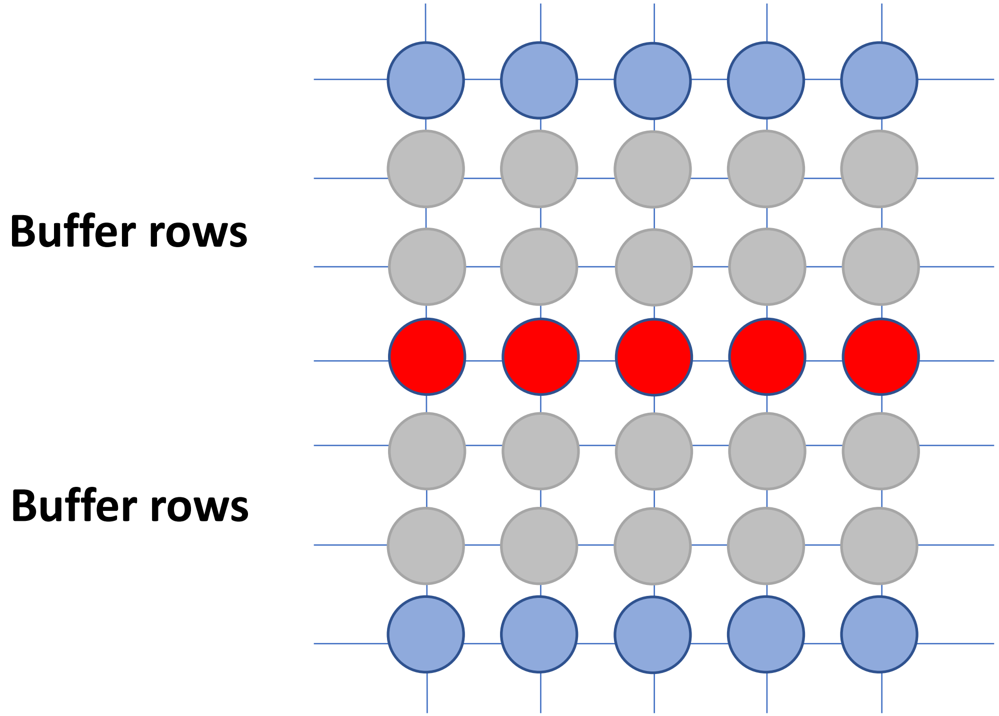


Increased refresh rate

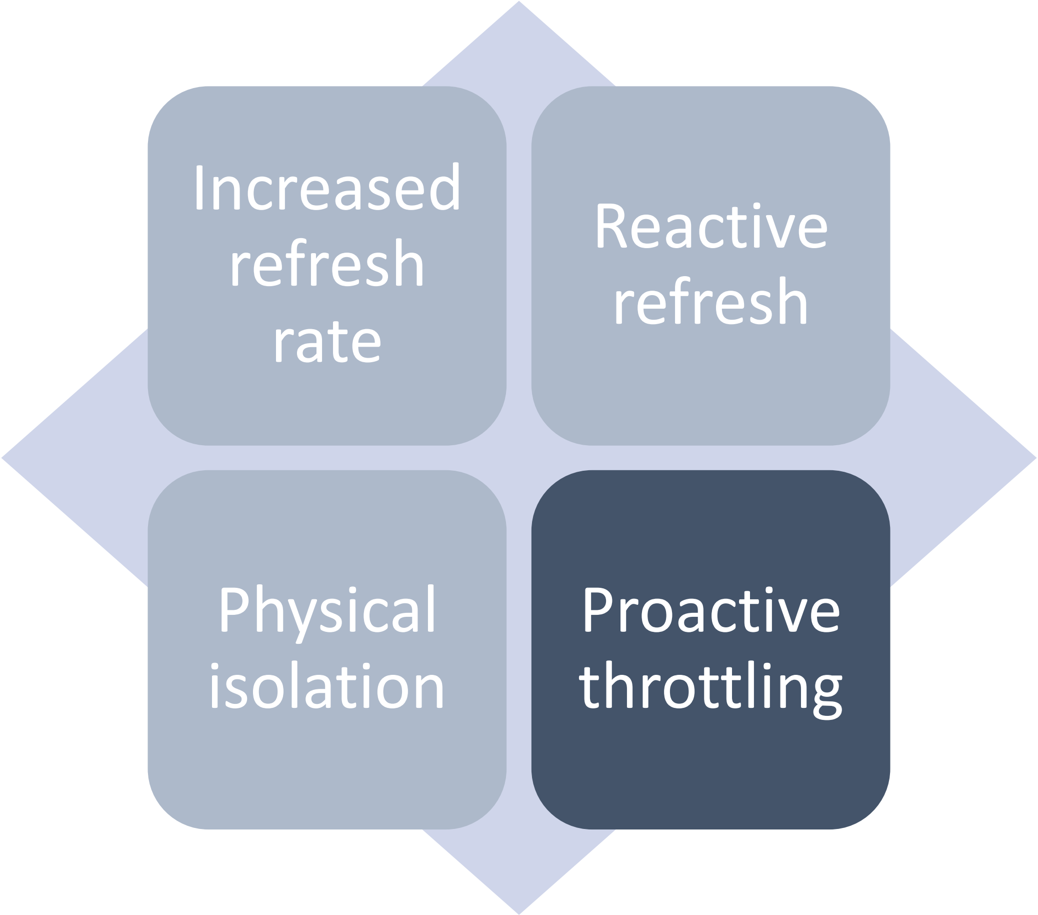
Reactive refresh

Physical isolation

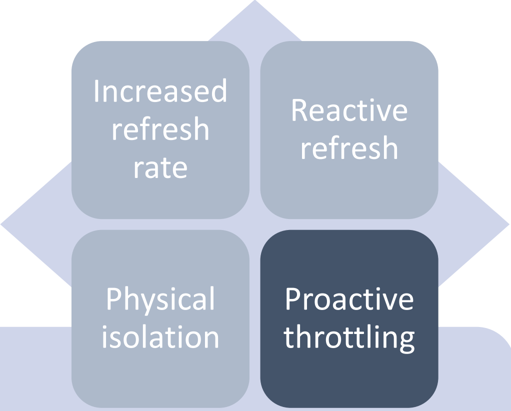
Proactive throttling



Current solutions to RowHammer



Proactive throttling


Increased refresh rate

Reactive refresh

Physical isolation

Proactive throttling

What: limit repeated access to the same row

e.g., by setting a minimum access delay

e.g., by limiting number of accesses to a row within refresh window

Challenge: performance overhead

Will we delay every access?

Challenge: area overhead

How do you track the number of row activations?

Proactive throttling

Increased refresh rate

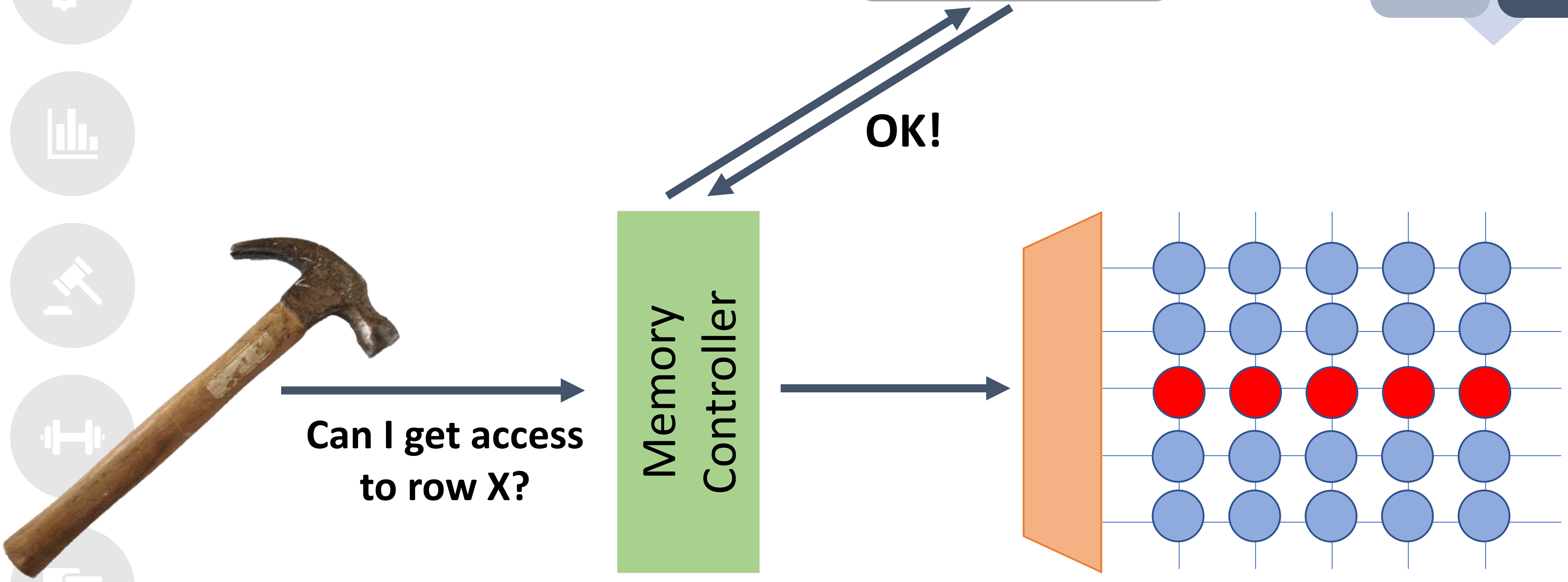
Reactive refresh

Physical isolation

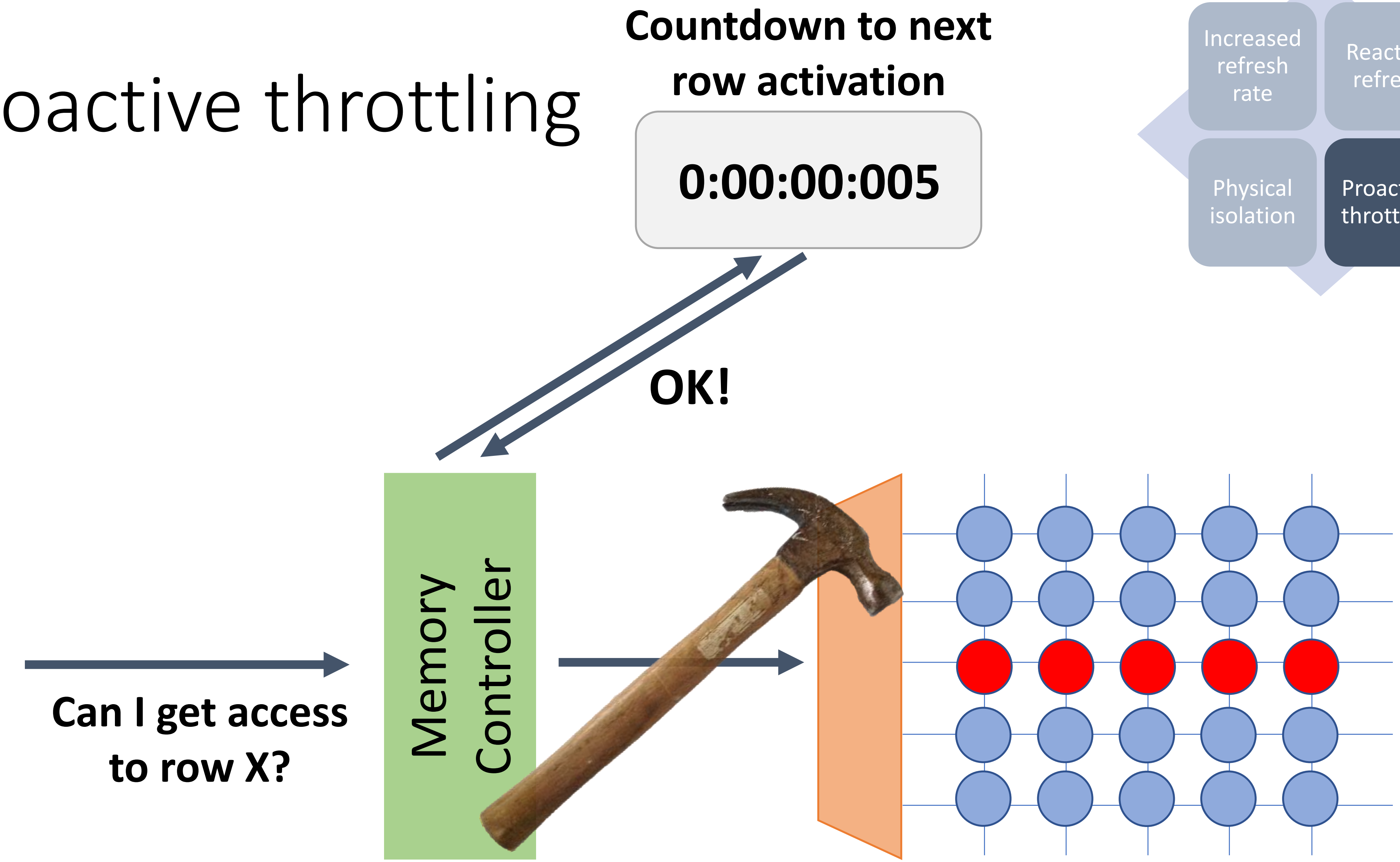
Proactive throttling

Countdown to next row activation

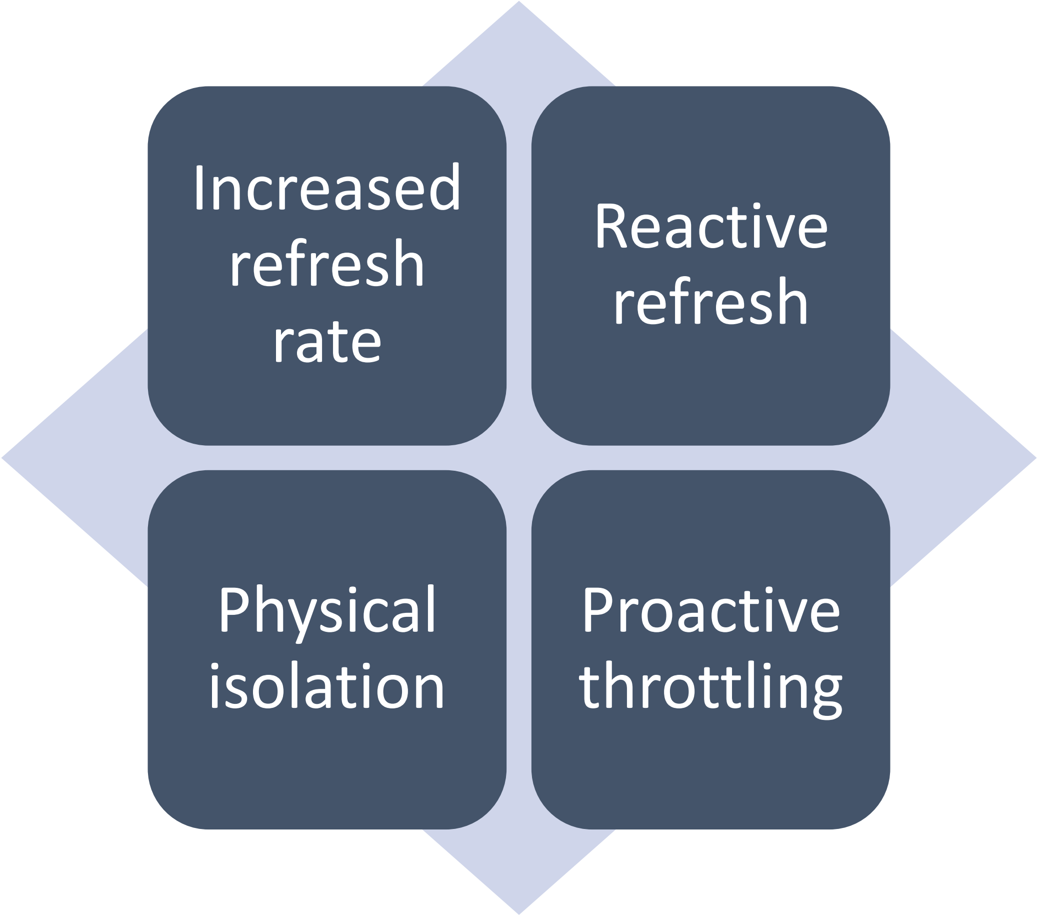
0:00:00:000



Proactive throttling



Current solutions to RowHammer



Increased refresh rate

Reactive refresh

Physical isolation

Proactive throttling

In search of a better solution

Efficient: low performance/area overhead

Scalable: we want things to work in the future

Implemented **without knowledge** of **or modification** to DRAM chip

Key idea: selectively throttle RowHammer-like memory accesses by

Tracking activation rates of all rows in an area-efficient way

Using tracking data to **throttle** RowHammer unsafe activations

Identifying and **limiting** row activation rates of potential attacker threads *(minimizes performance degradation of benign threads)*

Mechanisms & Implementation

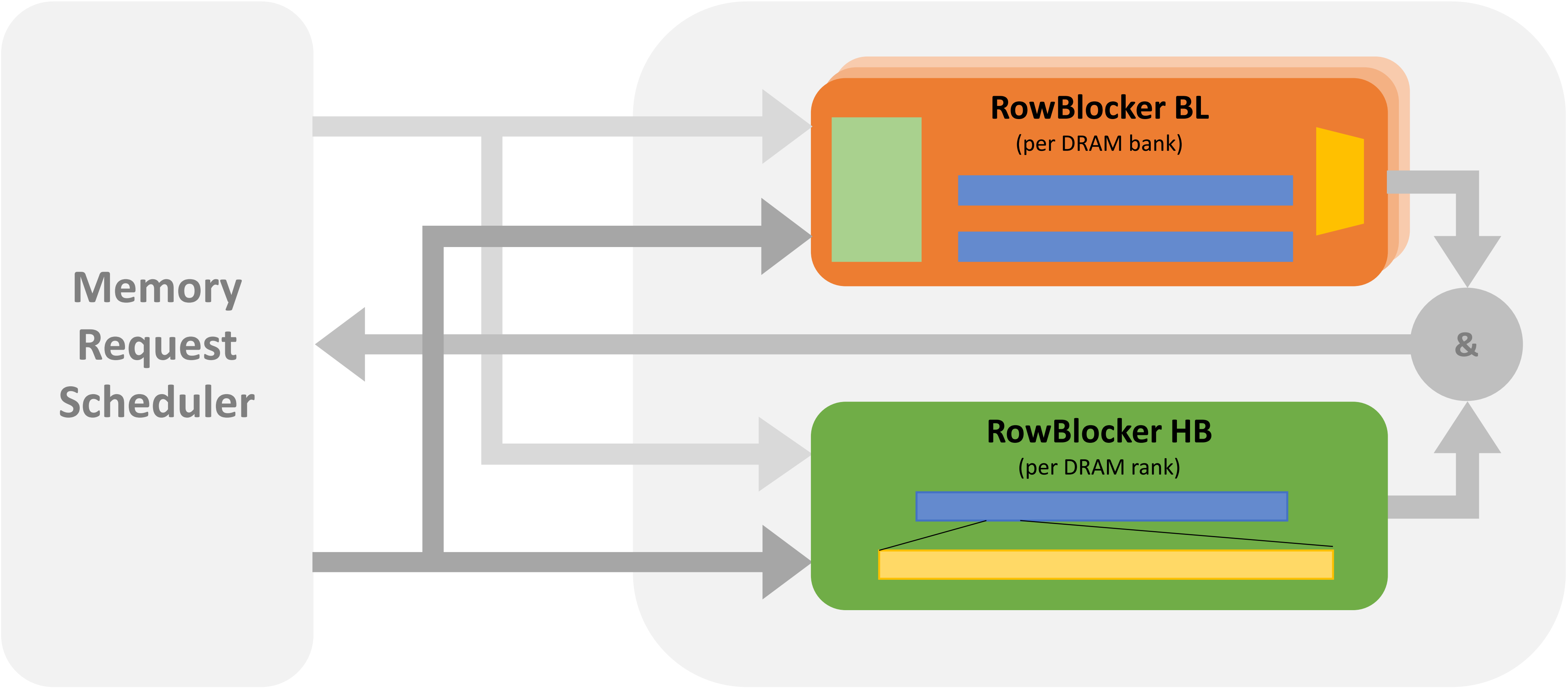
BlockHammer =

RowBlocker

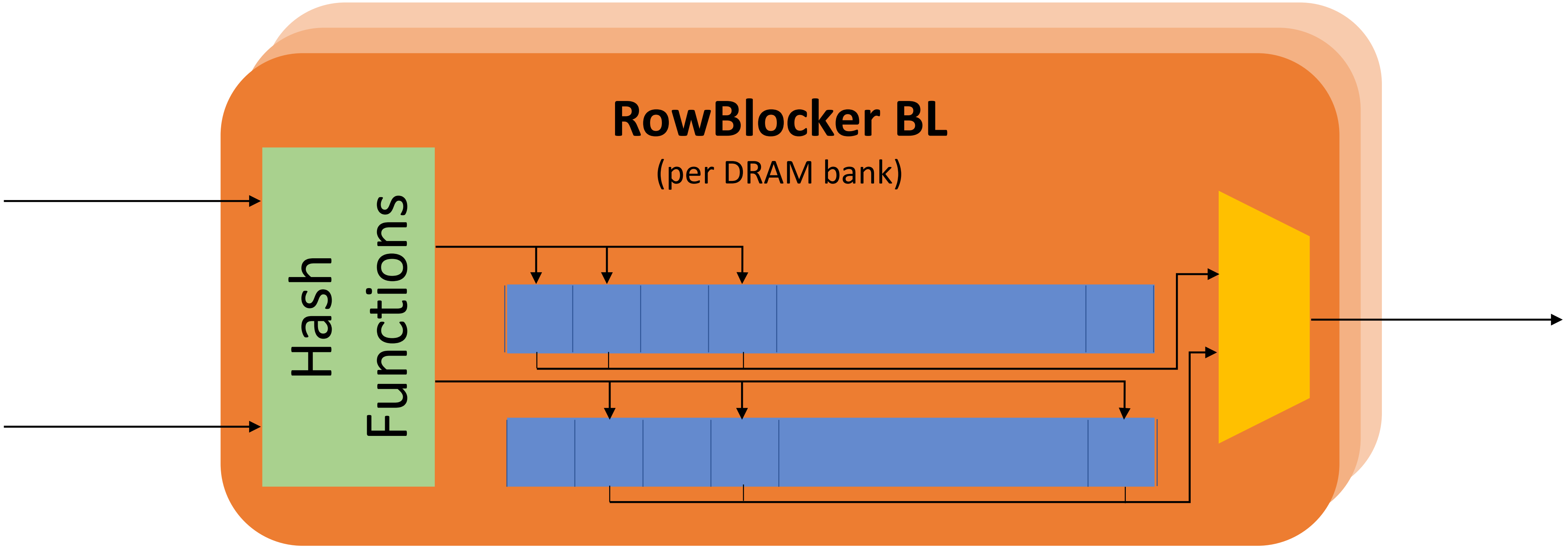
+

AttackThrottler

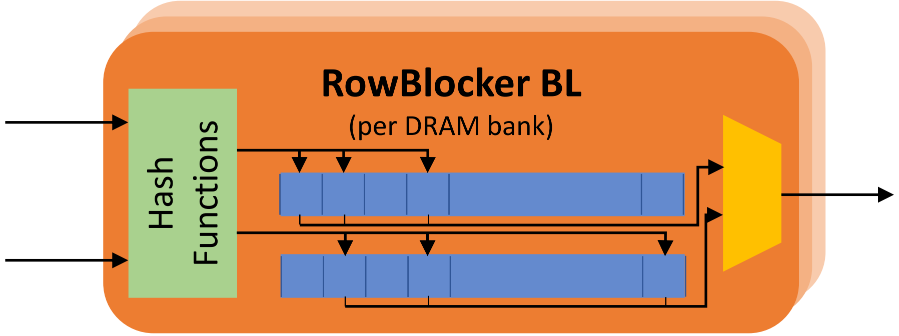
RowBlocker


Memory Request Scheduler
RowBlocker BL
(per DRAM bank)
&
RowBlocker HB
(per DRAM rank)

RowBlocker BL



RowBlocker BL

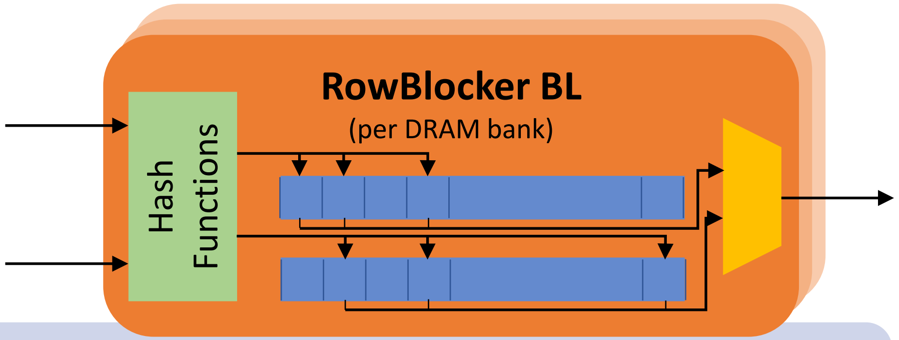


Goal 1: Track which rows have been activated and how many times

Goal 2: Blacklist when activation rate exceeds blacklisting threshold

How can we do this area-efficiently?

Recap: Bloom filter



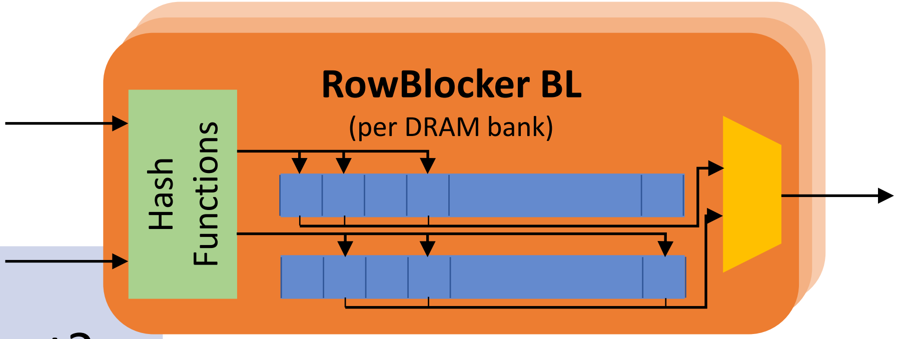
Question: does a set contain a certain element?

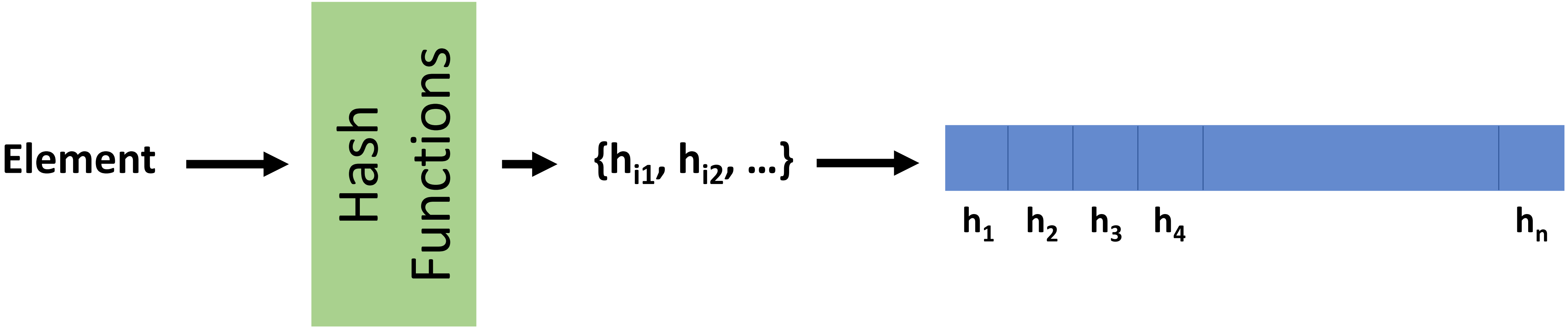
Main components: hash functions + bit array

Operations: insert, test, clear

Recap: Bloom filter

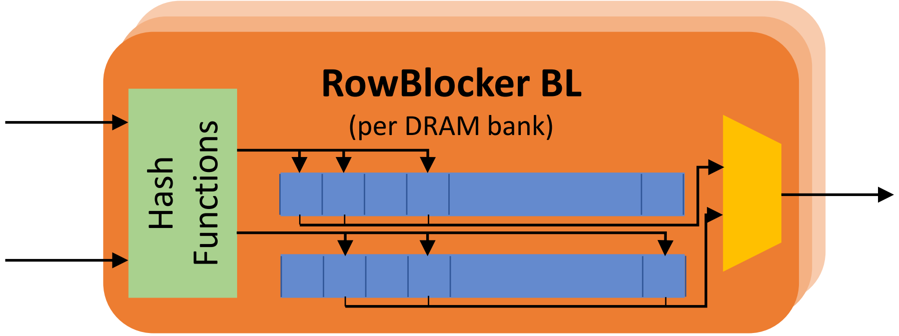
Question: does a set contain a certain element?

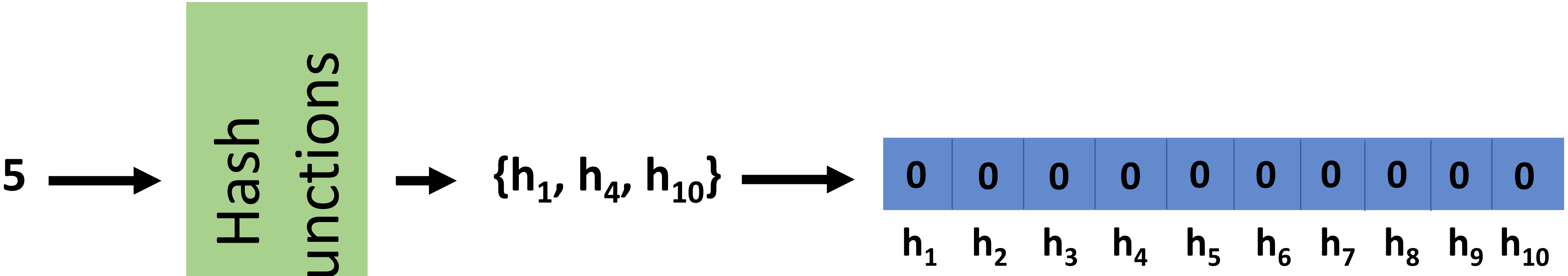




Recap: Bloom filter

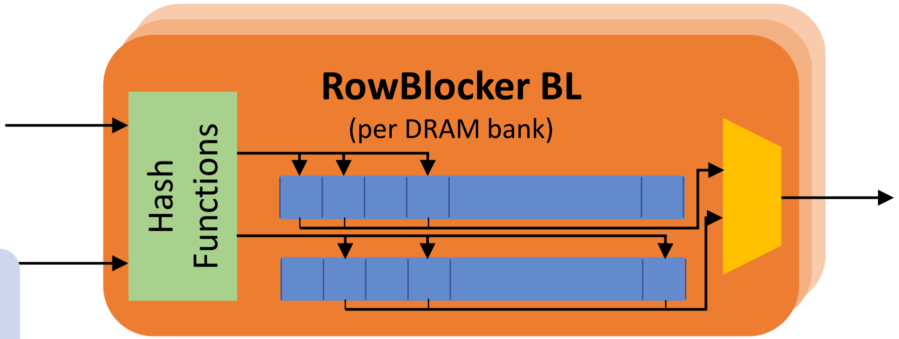
! **Insert 5**





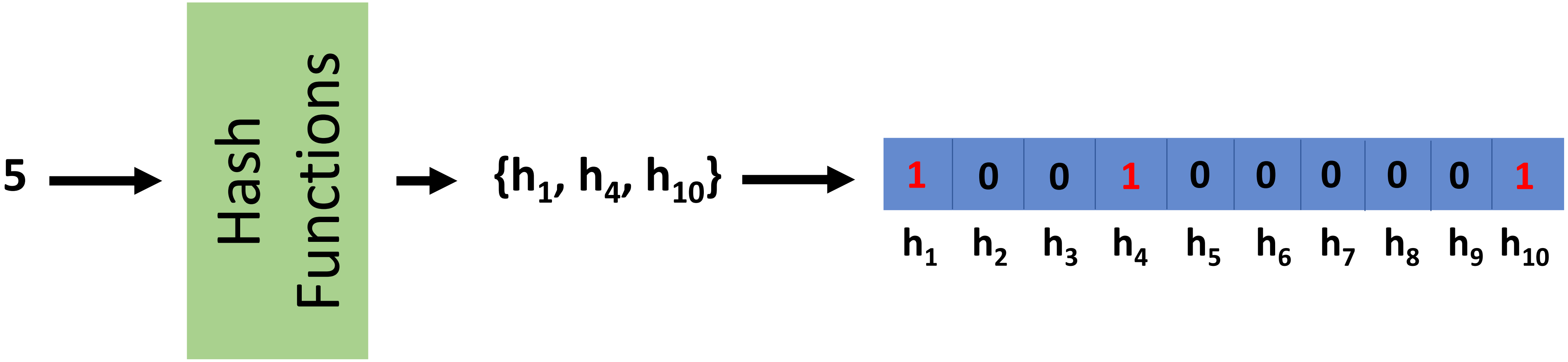
0	0	0	0	0	0	0	0	0	0
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Recap: Bloom filter



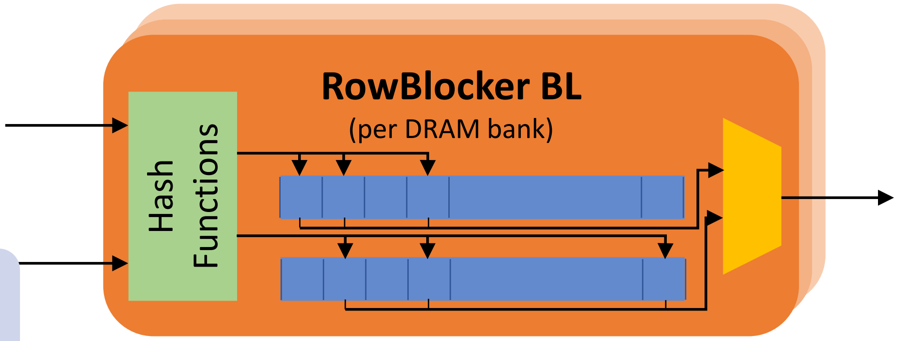
Insert 5

Set = {5}



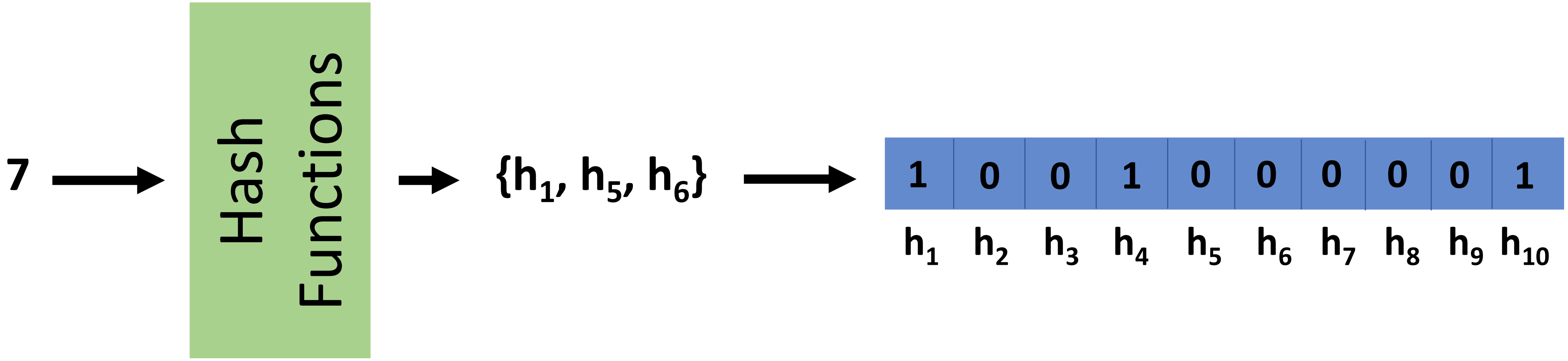
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}
1	0	0	1	0	0	0	0	0	1

Recap: Bloom filter



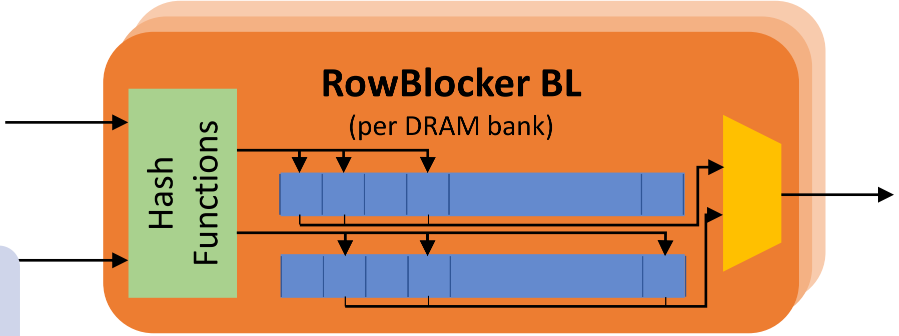
! **Insert 7**

ℹ **Set = {5}**



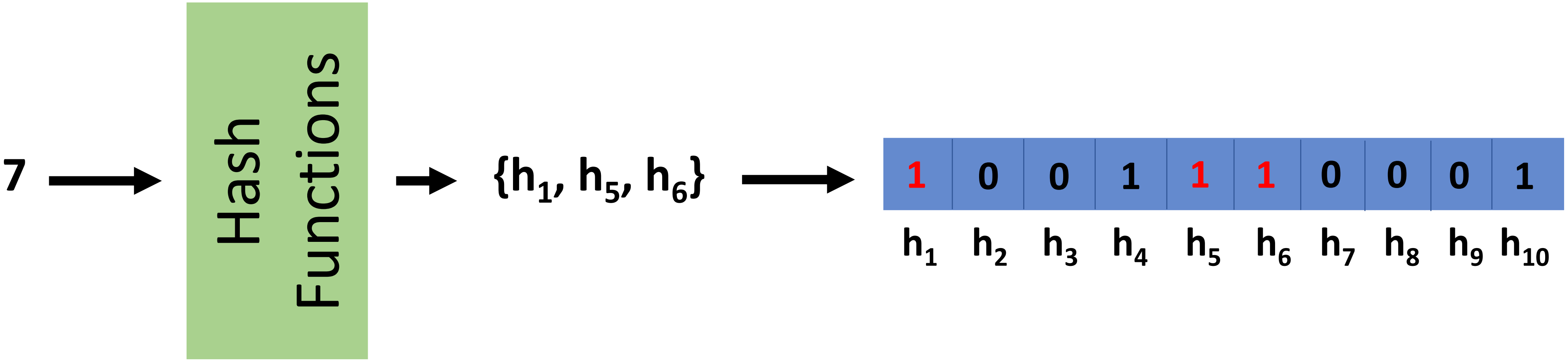
1	0	0	1	0	0	0	0	0	1
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Recap: Bloom filter



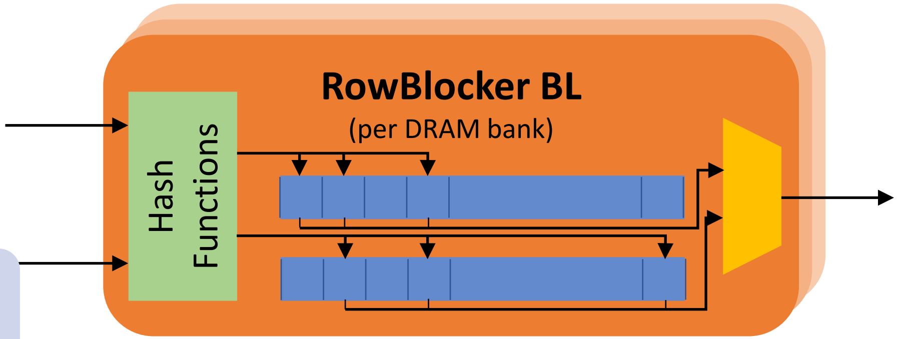
! **Insert 7**

Set = {5, 7}



7 → Hash Functions → $\{h_1, h_5, h_6\}$ →

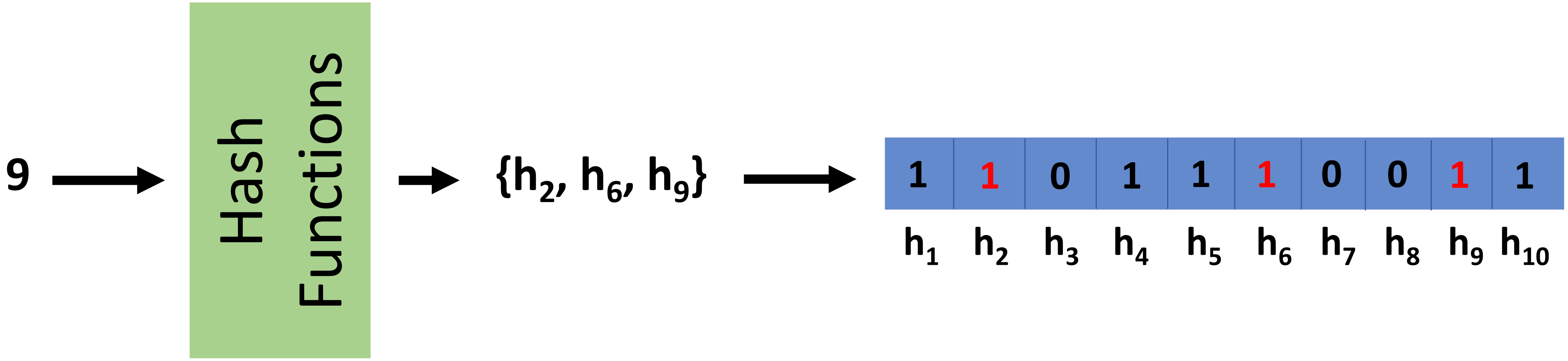
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}
1	0	0	1	1	1	0	0	0	1



Recap: Bloom filter

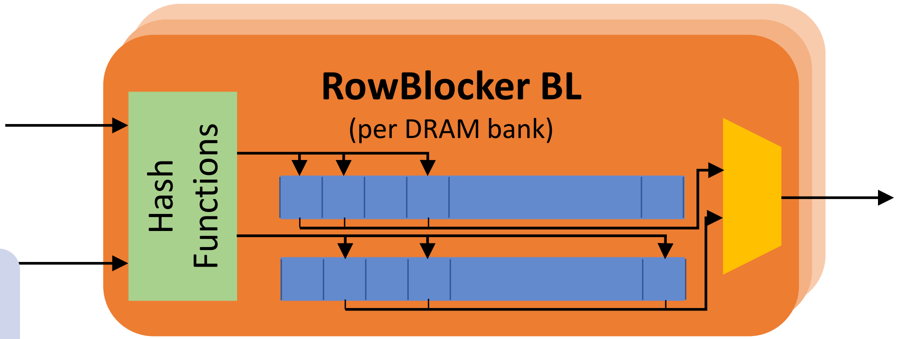
! **Insert 9**

ℹ **Set = {5, 7}**



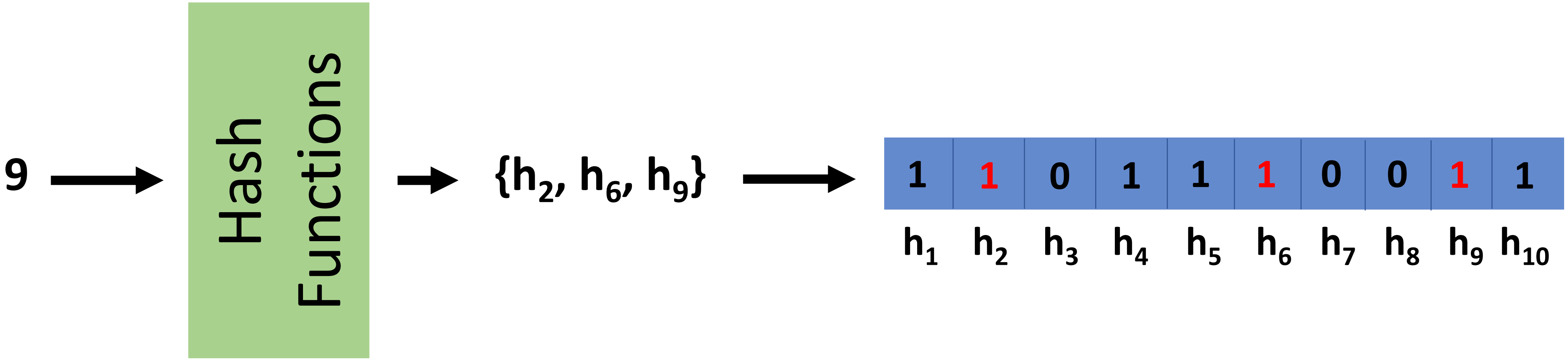
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}
1	1	0	1	1	1	0	0	1	1

Recap: Bloom filter



! **Insert 9**

ℹ **Set = {5, 7, 9}**

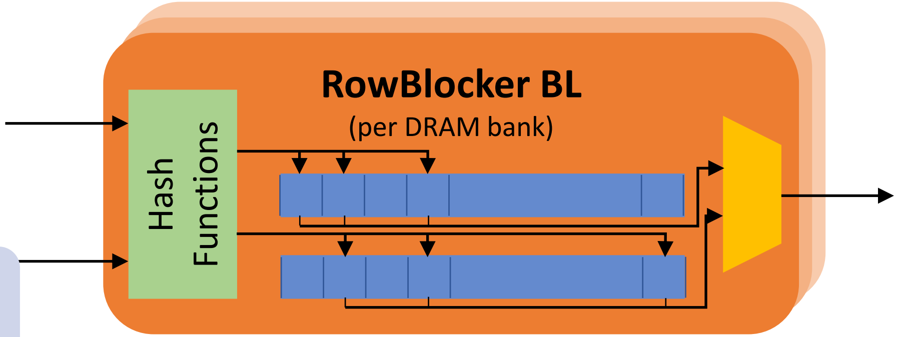


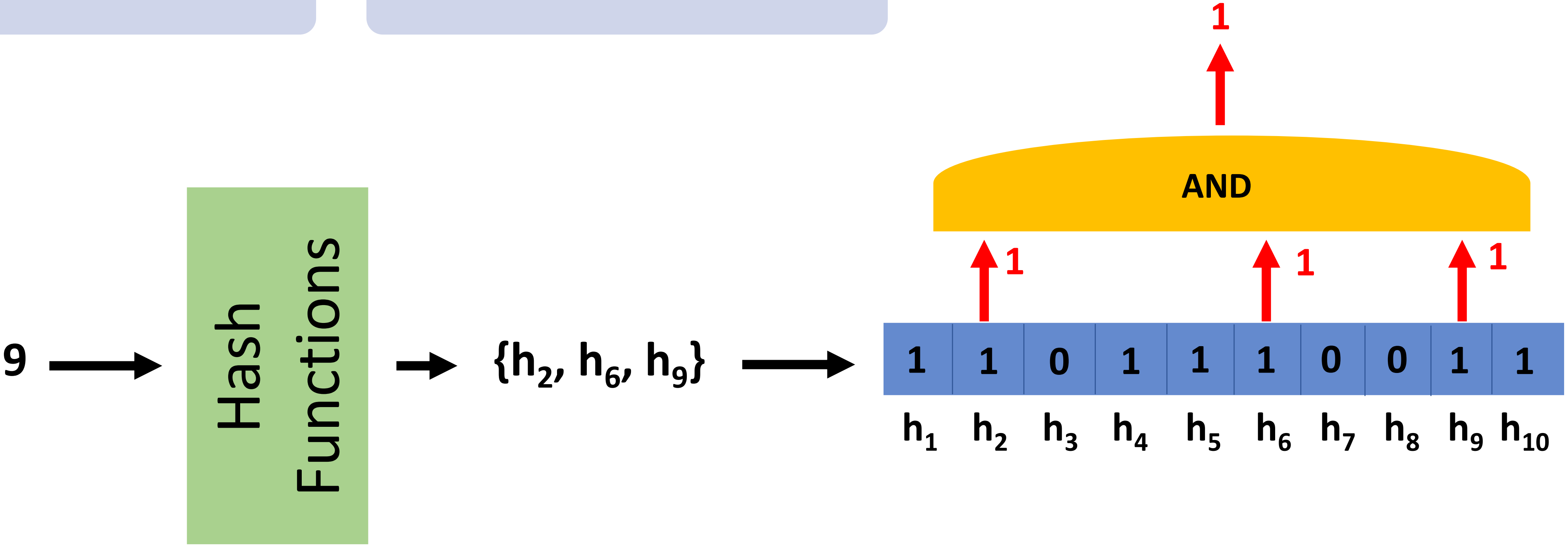
1	1	0	1	1	1	0	0	1	1
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Recap: Bloom filter

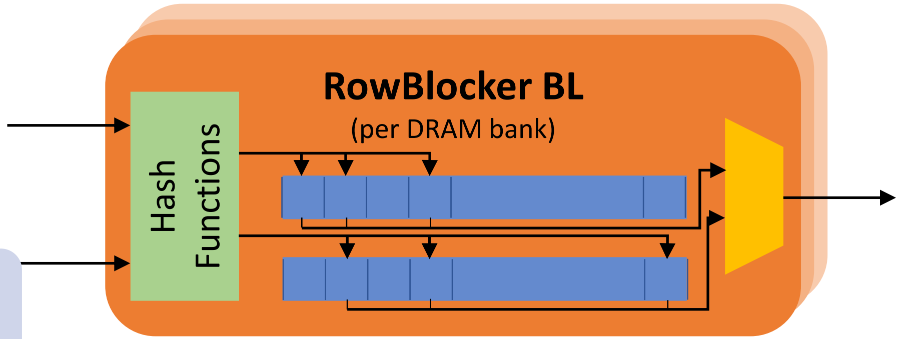
! **Test 9**

i **Set = {5, 7, 9}**





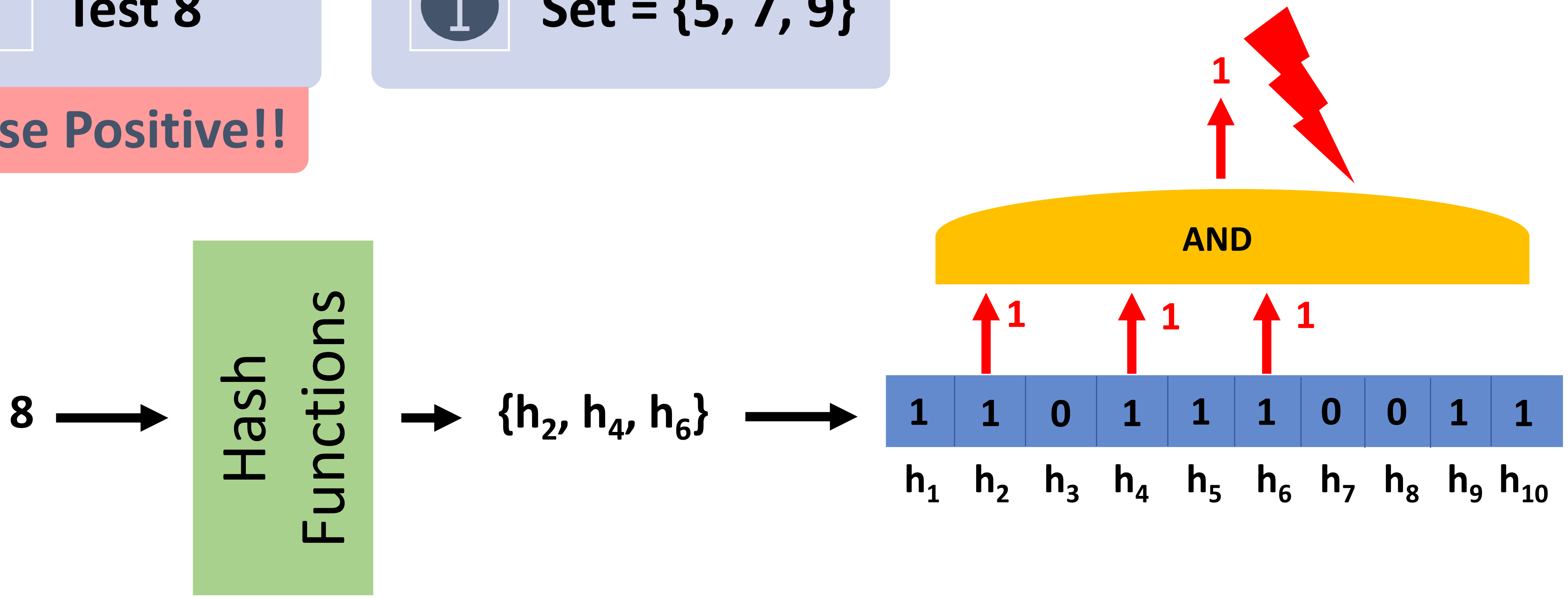
Recap: Bloom filter



! Test 8

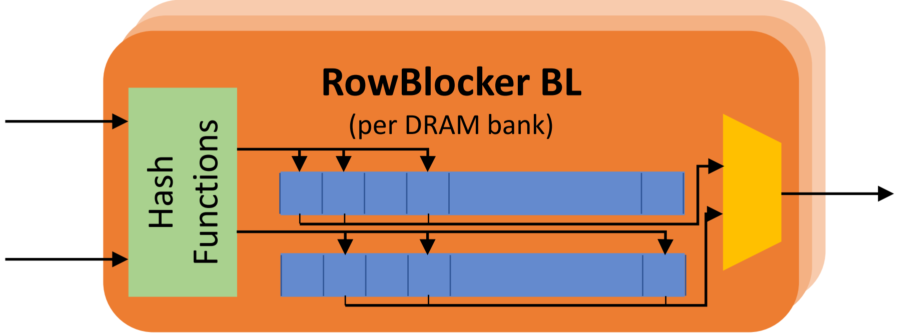
False Positive!!

Set = {5, 7, 9}



1	1	0	1	1	1	0	0	1	1
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

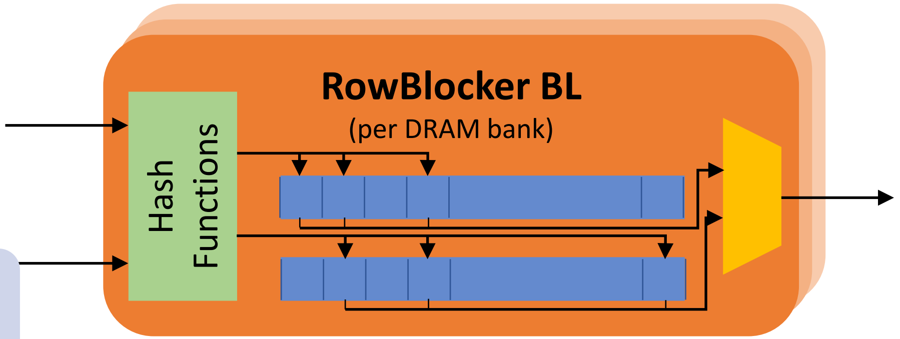
Counting Bloom filter



Remember: we want to know how many times a row is activated
(and blacklist it if activation rate > threshold)

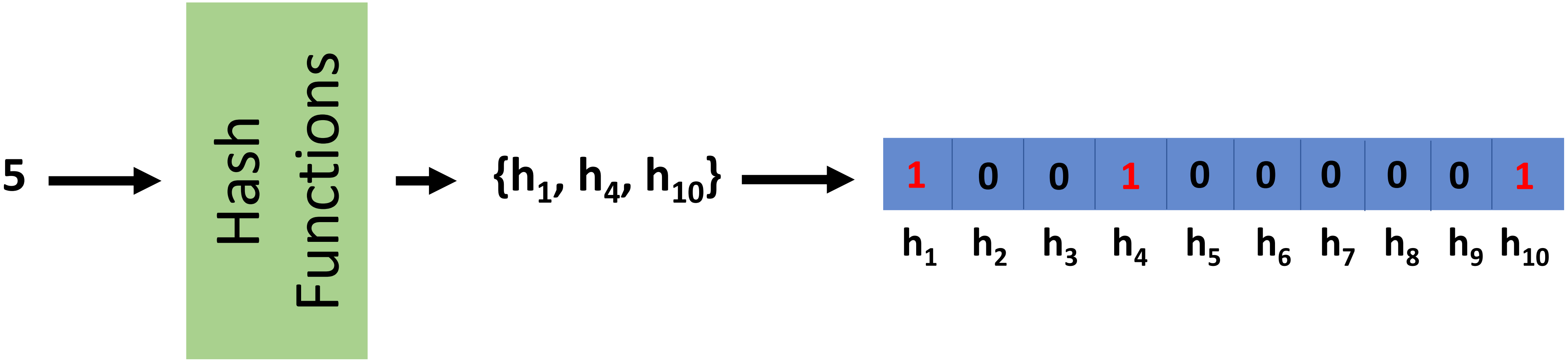
Idea: Counting Bloom filter (CBF)
(tracks number of times an element is inserted into filter)

Counting Bloom filter

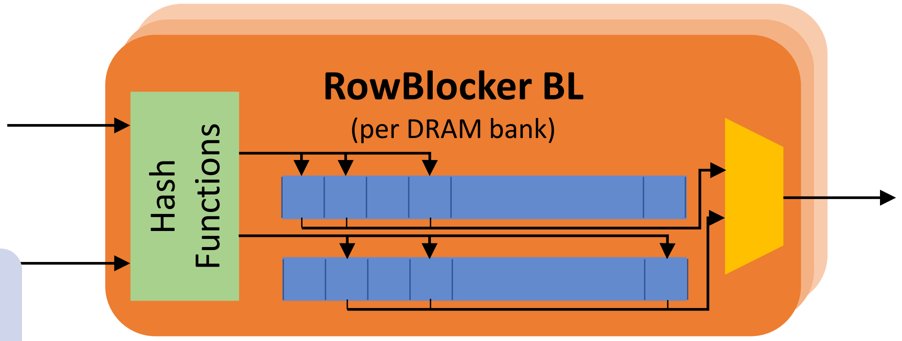


Insert 5

Set = {5}

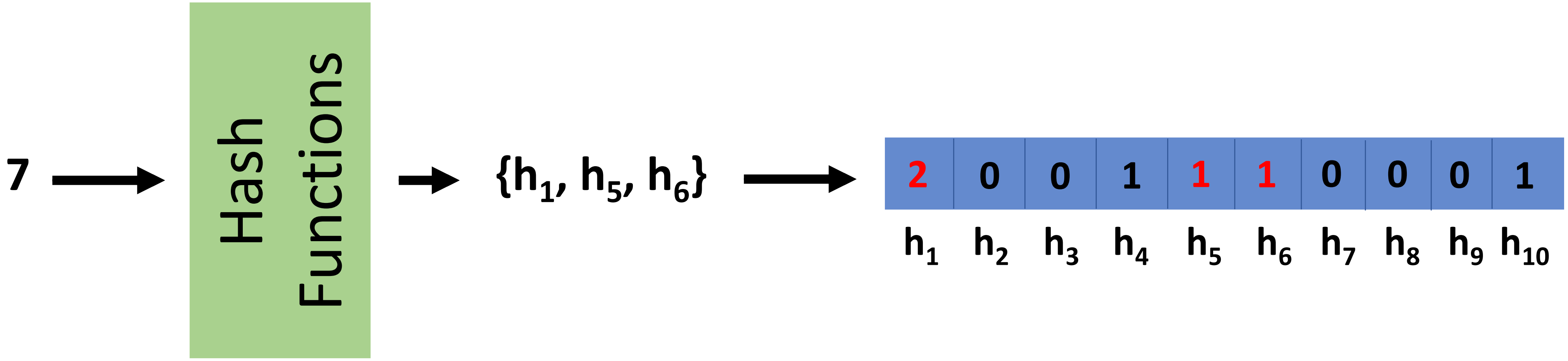


Counting Bloom filter



! **Insert 7**

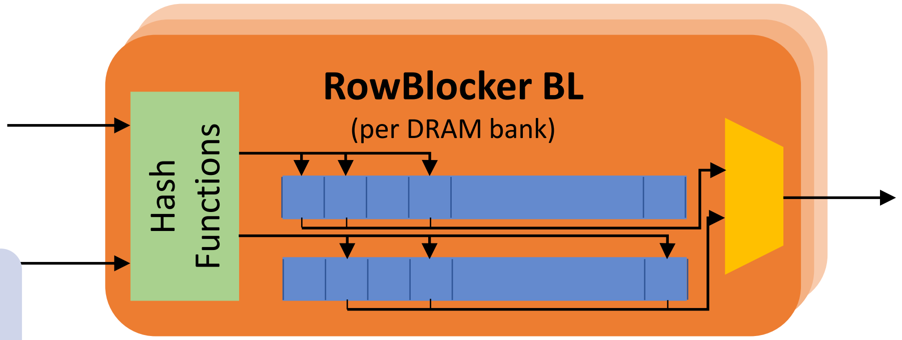
ℹ **Set = {5, 7}**



$\{h_1, h_5, h_6\}$

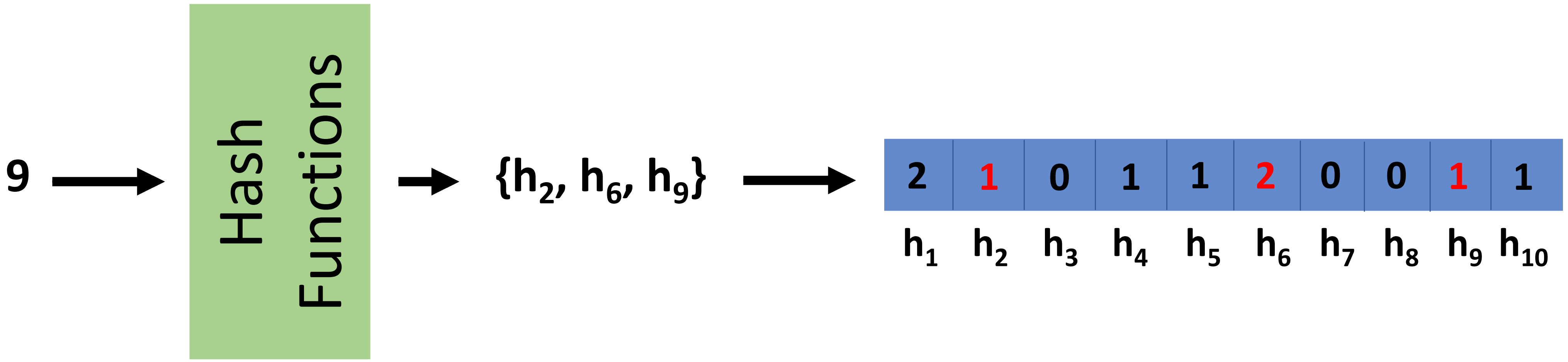
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}
2	0	0	1	1	1	0	0	0	1

Counting Bloom filter



❗ **Insert 9**

ℹ **Set = {5, 7, 9}**

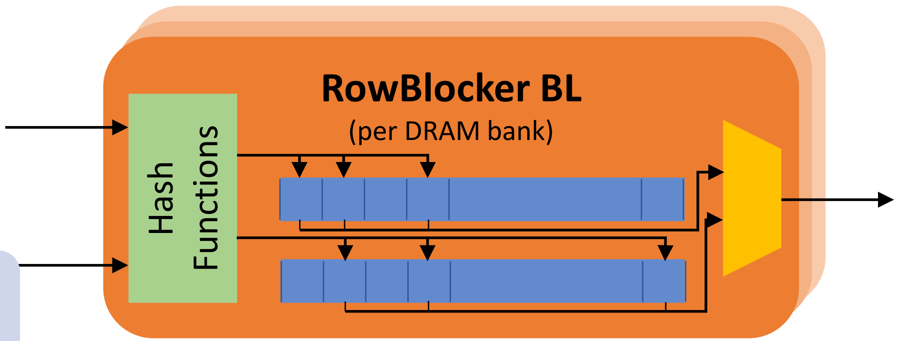


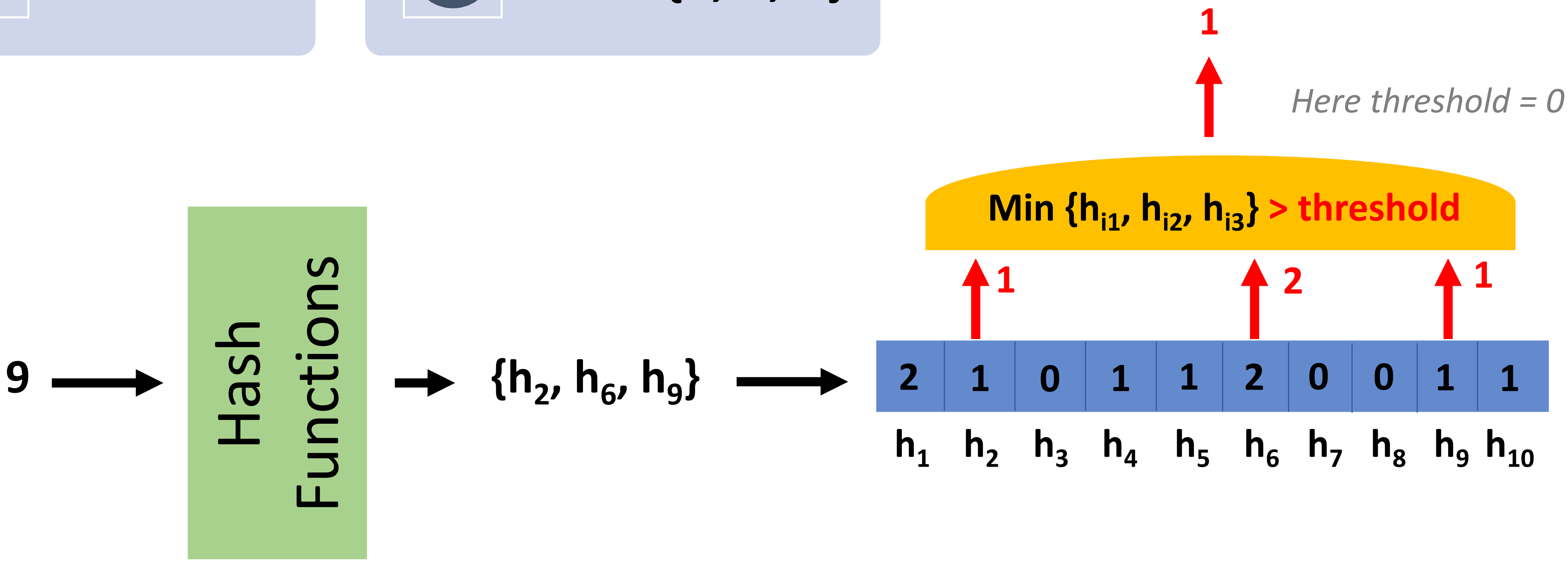
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}
2	1	0	1	1	2	0	0	1	1

Counting Bloom filter

Test 9

Set = {5, 7, 9}

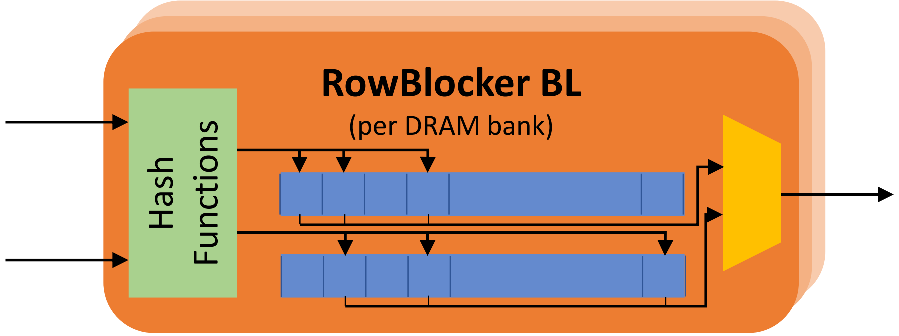




1
Min $\{h_{i1}, h_{i2}, h_{i3}\}$ > threshold
1
2
1
9
Hash Functions
$\{h_2, h_6, h_9\}$
2 1 0 1 1 2 0 0 1 1
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9 h_{10}

Here threshold = 0

Counting Bloom filter

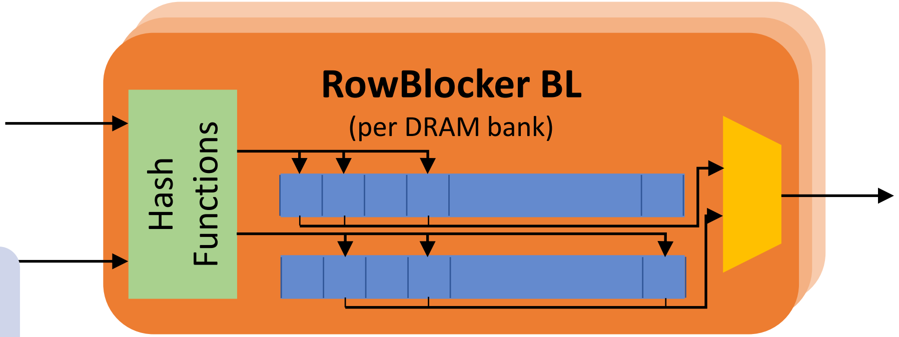


Idea: Counting Bloom filter (CBF)

(tracks number of times an element is inserted into filter)

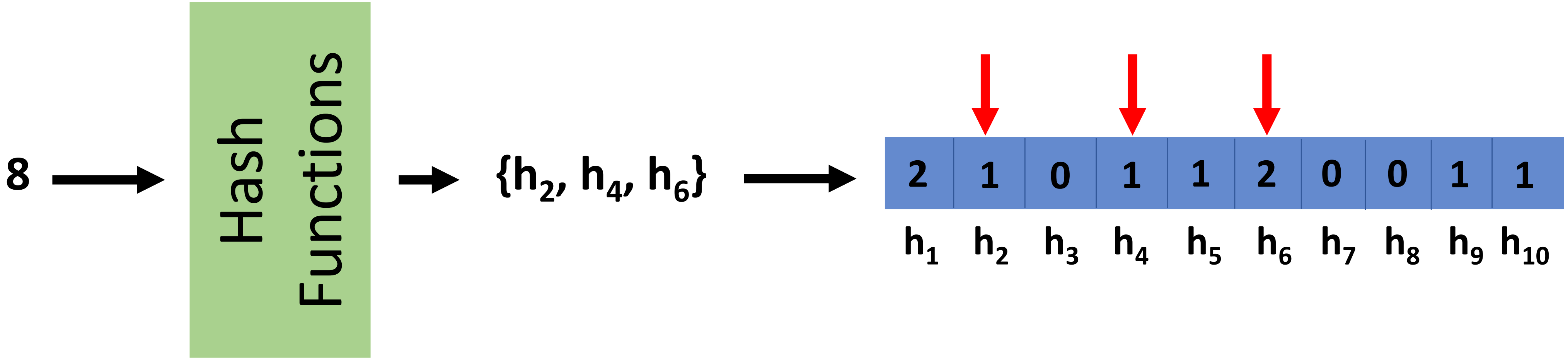
But Bloom filter is getting **saturated**

Counting Bloom filter

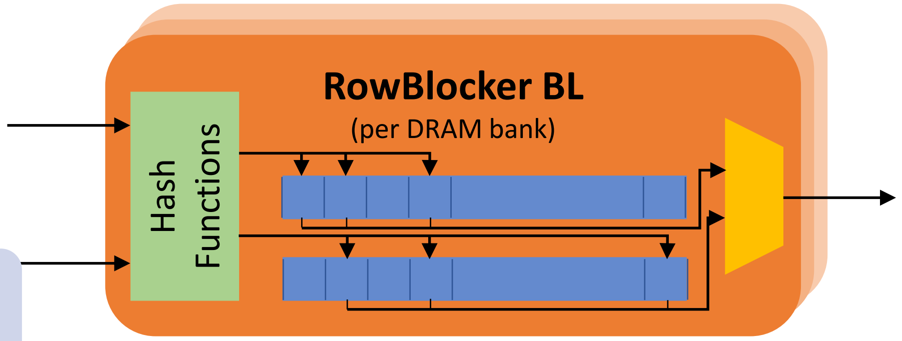


! **Delete 8**

i **Set = {5, 7, 9}**


8
Hash Functions
$\{h_2, h_4, h_6\}$
2 1 0 1 1 2 0 0 1 1
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9 h_{10}

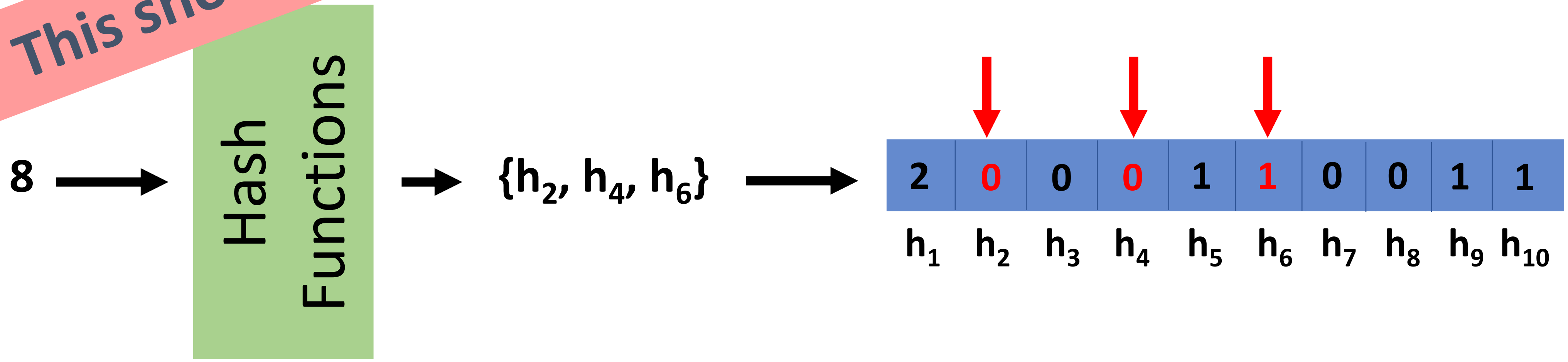
Counting Bloom filter



! **Delete 8**

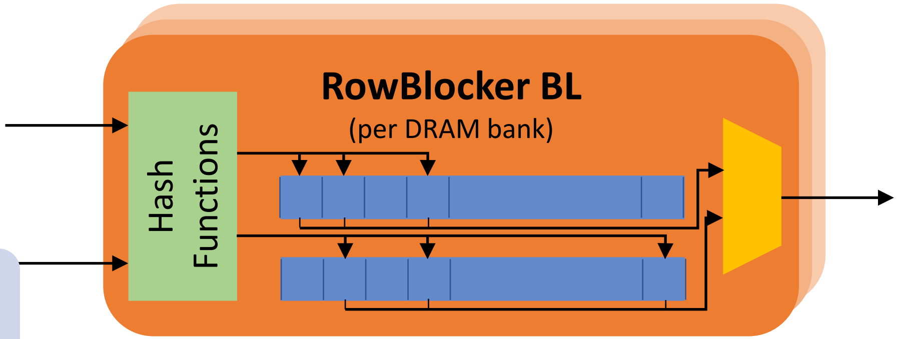
i **Set = { 9}**

This shouldn't be possible!



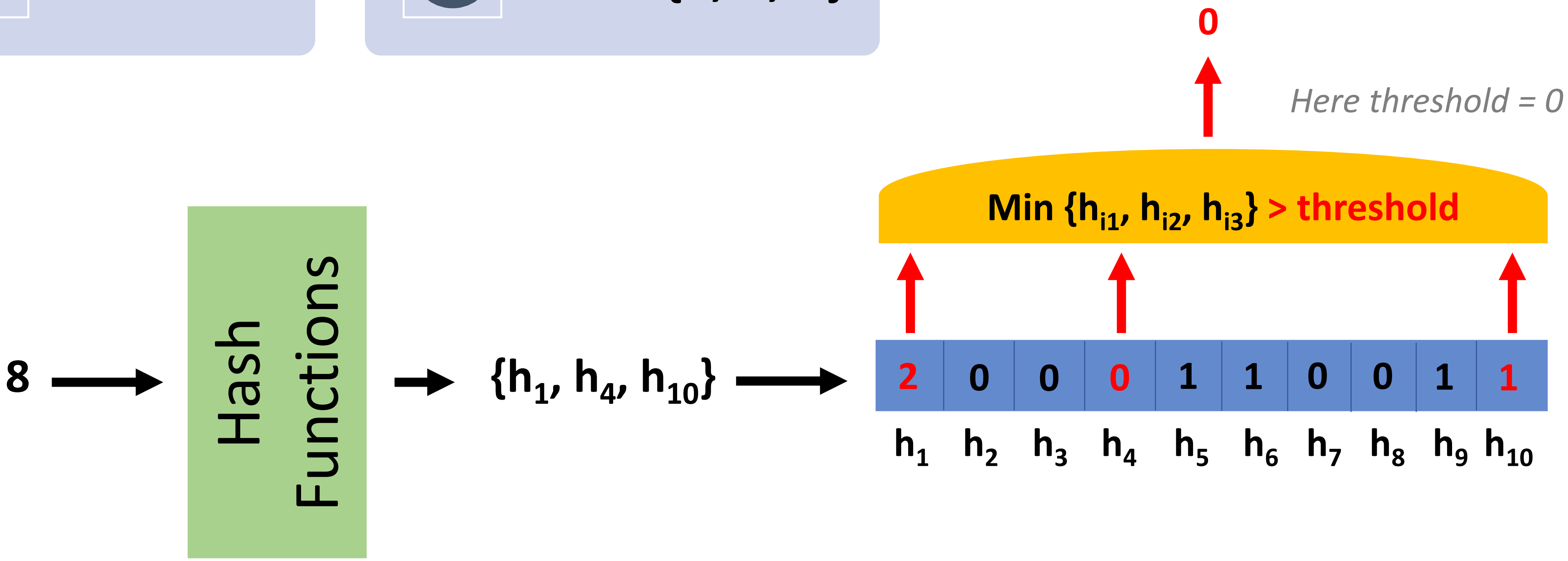
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}
2	0	0	0	1	1	0	0	1	1

Counting Bloom filter

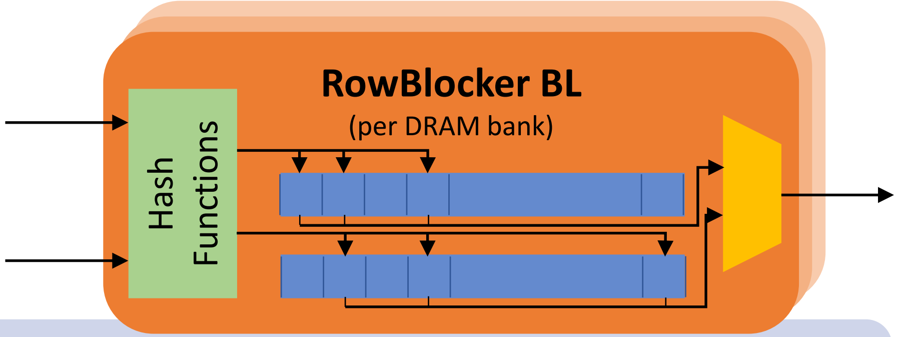


Test 5

Set = {5, 7, 9}



Unified Bloom filter

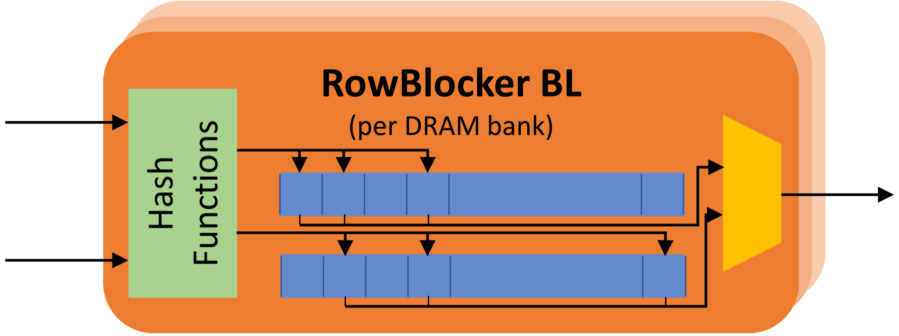


Remember: we want to know how many times a row is activated
(and blacklist it if activation rate > threshold)

But we **can't prevent false negatives**
(without compensating for it in terms of space)

Idea: Unified Bloom filter (UBF)
*(tracks all elements inserted into filter **during specific time window**)*

Unified Bloom filter



Unified Bloom filter: active + passive Bloom filter

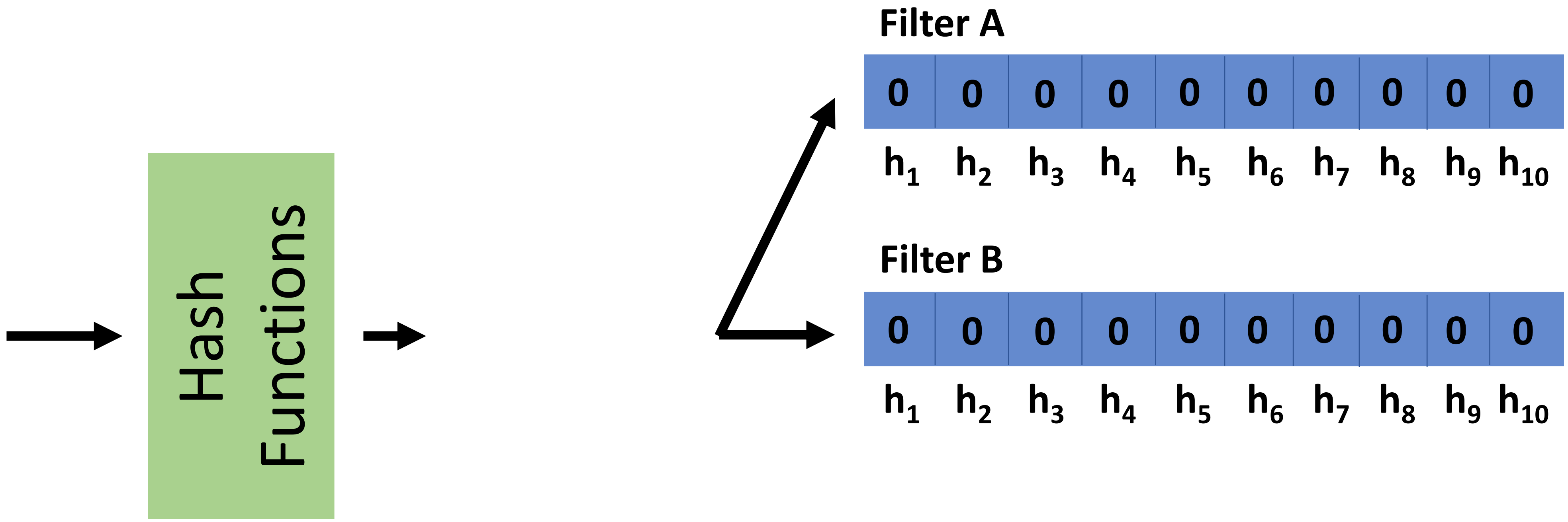
- Both insert all elements into filter
- Only active filter responds to test queries
- Active filter clears array at end of specified time interval (= epoch)
- Switch roles every epoch

Guarantees **no false negatives**
when tested for elements inserted in the last two epochs

Unified Bloom filter

	Epoch 1	Epoch 2	Epoch 3
Filter A			…
Filter B			…

…



Hash Functions

Filter A

0	0	0	0	0	0	0	0	0	0
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Filter B

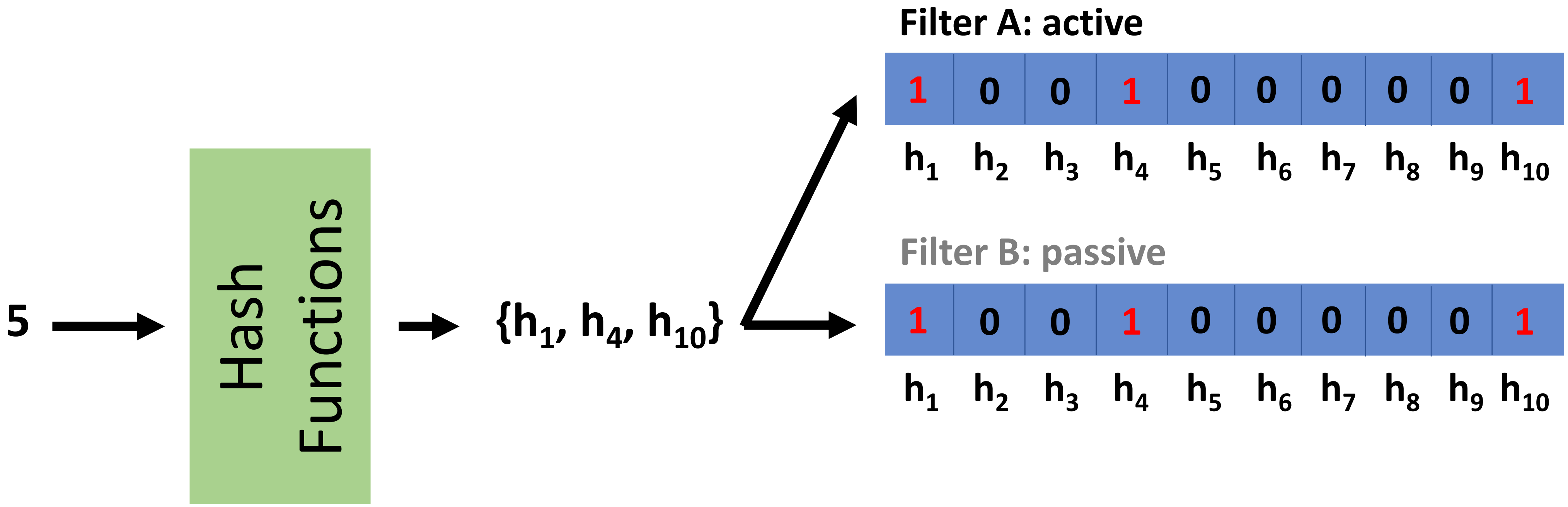
0	0	0	0	0	0	0	0	0	0
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Unified Bloom filter

	Epoch 1	Epoch 2	Epoch 3	
Filter A			…	…
Filter B			…	

Insert 5

Set = {5} $\text{Set}_A = \{5\} = \text{Set}_B$



Filter A: active

h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}
1	0	0	1	0	0	0	0	0	1

Filter B: passive

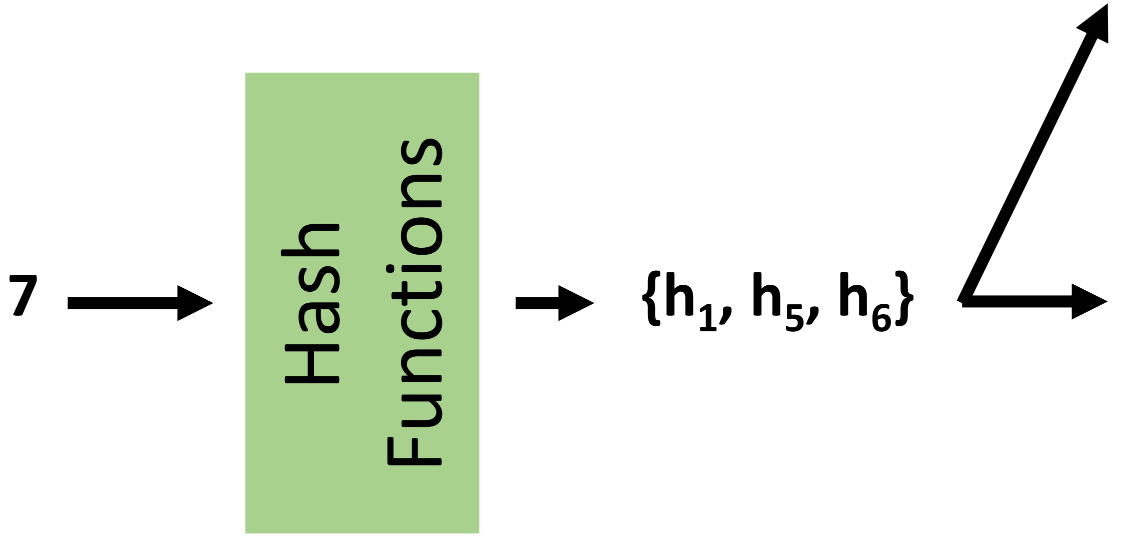
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}
1	0	0	1	0	0	0	0	0	1

Unified Bloom filter

	Epoch 1	Epoch 2	Epoch 3	
Filter A	🟦		...	...
Filter B	🟦		...	

Insert 7

Set = {5, 7} **Set_A = {5, 7} = Set_B**



Filter A: active

1	0	0	1	1	1	0	0	0	1
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Filter B: passive

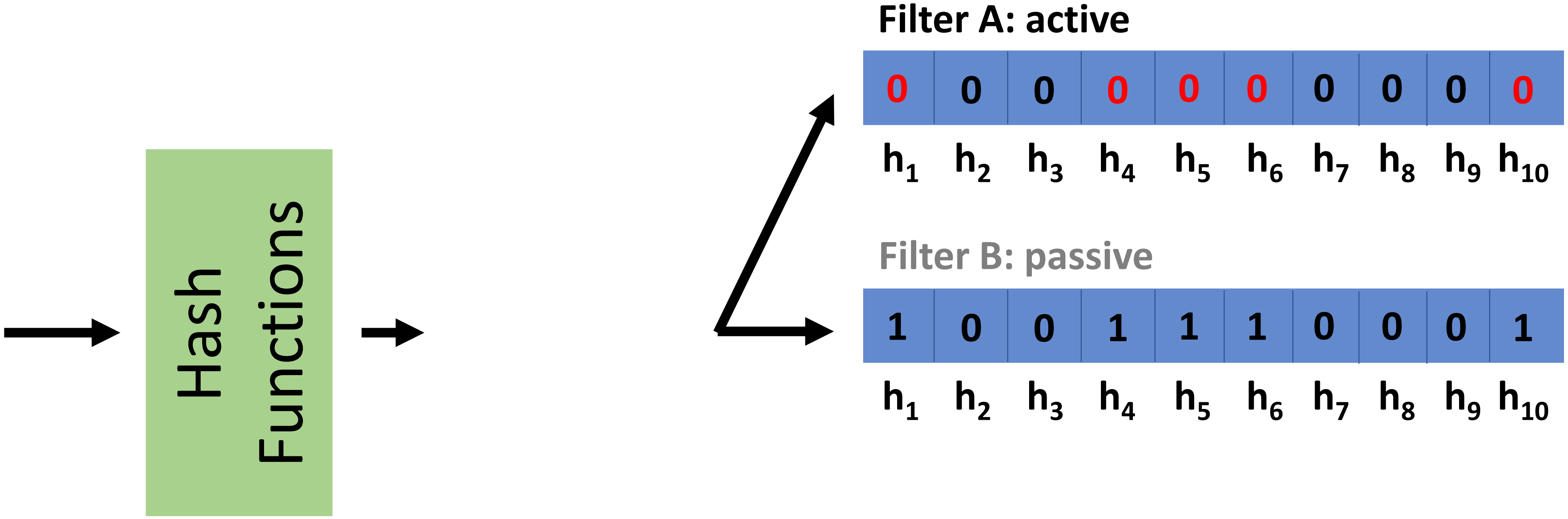
1	0	0	1	1	1	0	0	0	1
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Unified Bloom filter

	Epoch 1	Epoch 2	Epoch 3	
Filter A	✗		…	…
Filter B			…	

! **Clear A**

ℹ **Set = {5, 7}** **Set_A = { }, Set_B = {5, 7}**



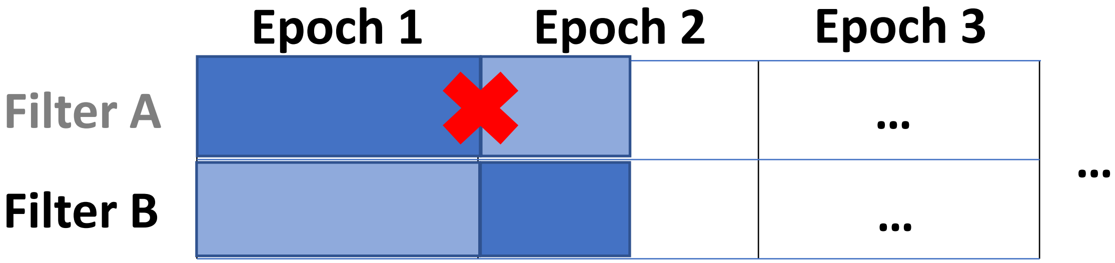
Filter A: active

0	0	0	0	0	0	0	0	0	0
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Filter B: passive

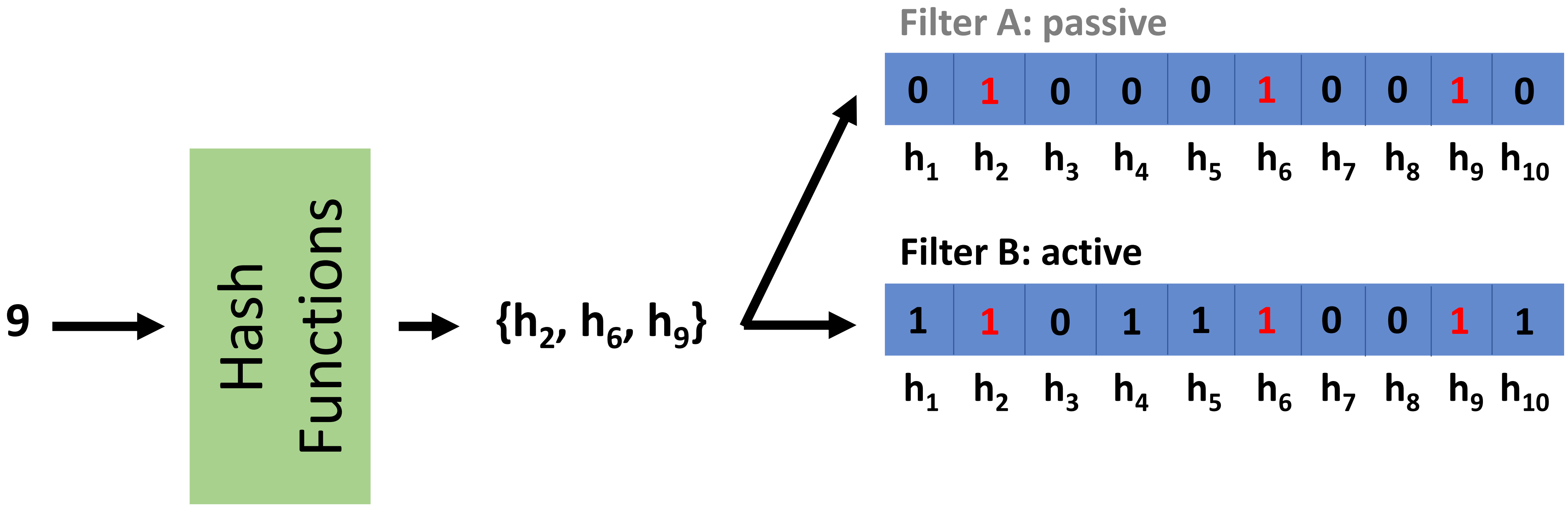
1	0	0	1	1	1	0	0	0	1
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Unified Bloom filter



! **Insert 9**

Set = {5, 7, 9} **Set_A = {9}, Set_B = {5, 7, 9}**



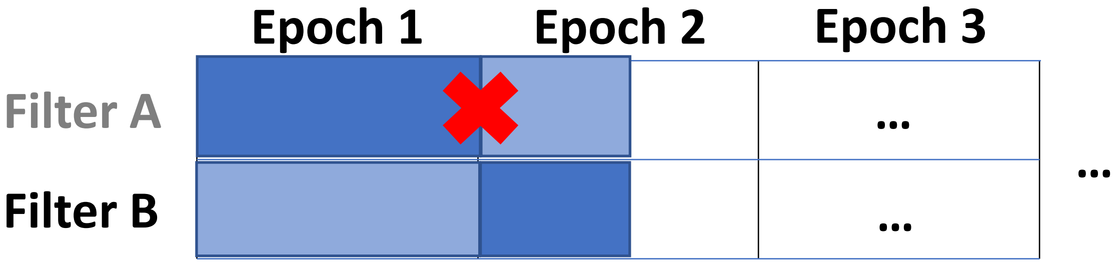
Filter A: passive

0	1	0	0	0	1	0	0	1	0
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Filter B: active

1	1	0	1	1	1	0	0	1	1
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Unified Bloom filter

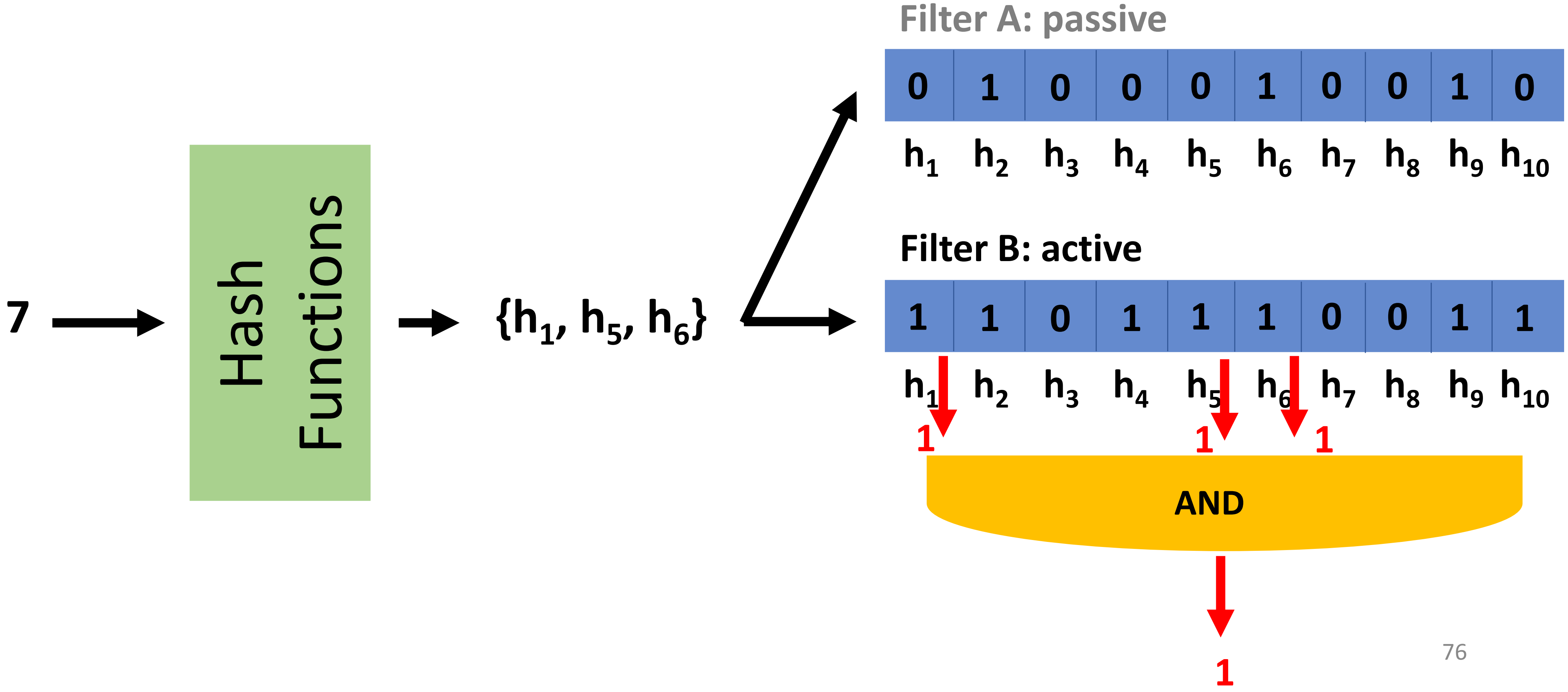


	Epoch 1	Epoch 2	Epoch 3
Filter A			…
Filter B			…

…

! **Test 7**

i **Set = {5, 7, 9}   Set$_A$ = {9}, Set$_B$ = {5, 7, 9}**



7 → Hash Functions → {h_1, h_5, h_6}

Filter A: passive

0	1	0	0	0	1	0	0	1	0
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

Filter B: active

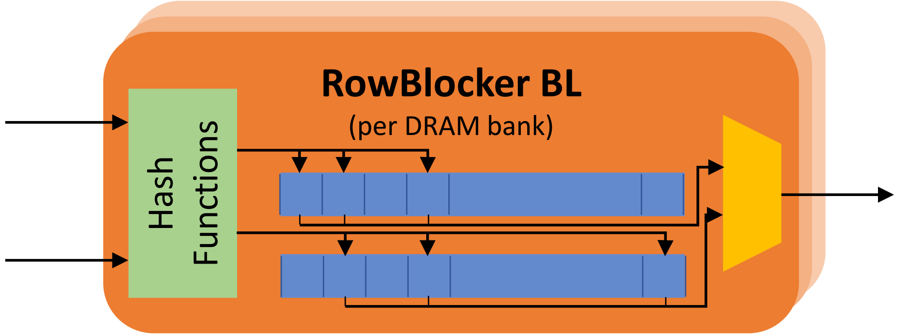
1	1	0	1	1	1	0	0	1	1
h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}

1 1 1

AND

1

Dual counting Bloom filter

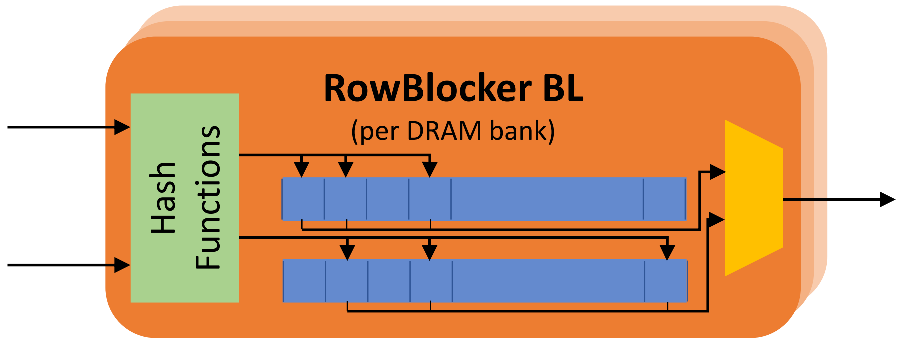


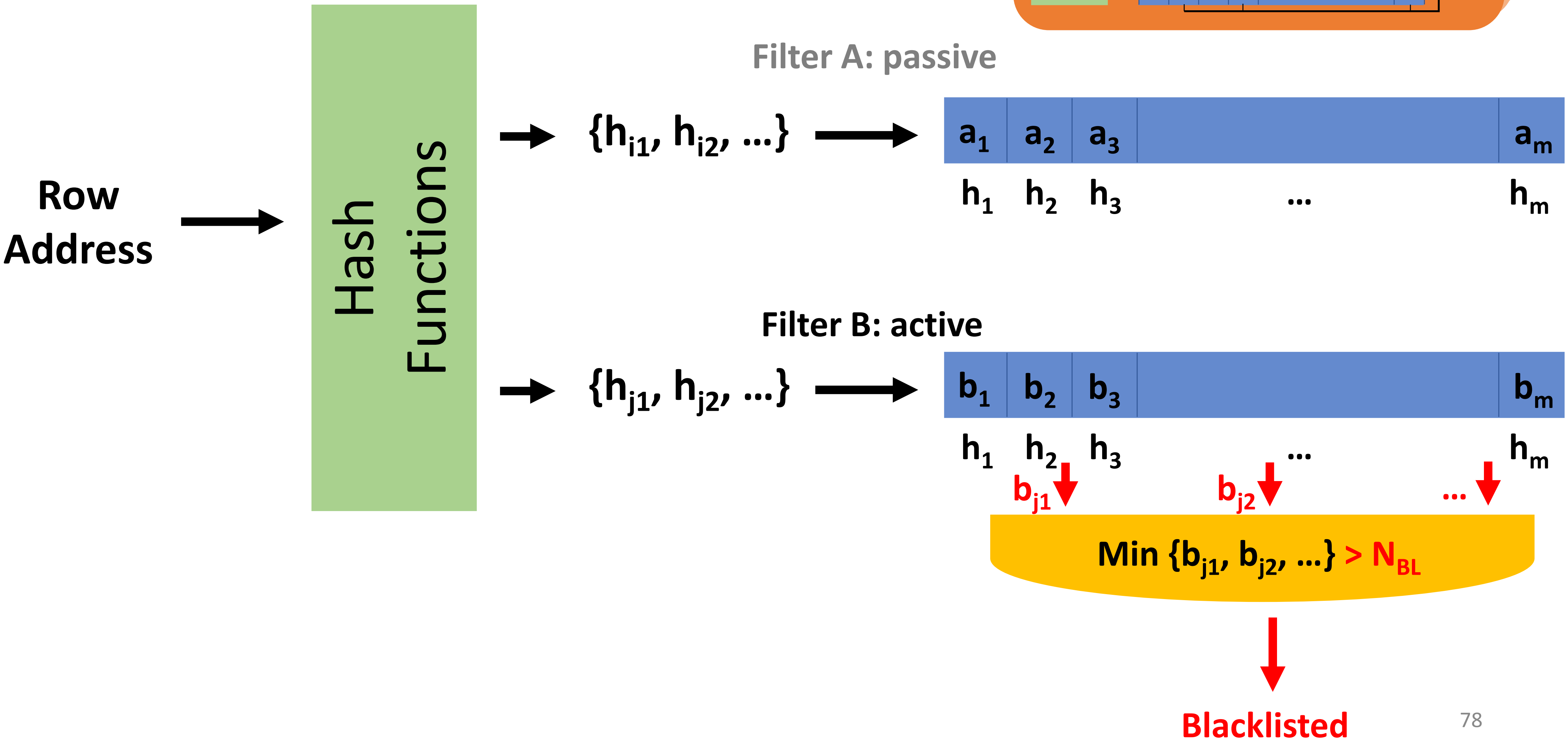
Remember: we want to know how many times a row is activated
(and blacklist it if activation rate > threshold))

Idea: dual counting Bloom filter (D-CBF)
= unified Bloom filter + counting Bloom filter

- *both filters use **different hash functions***
- *hash functions of active filter are **altered at end of epoch***

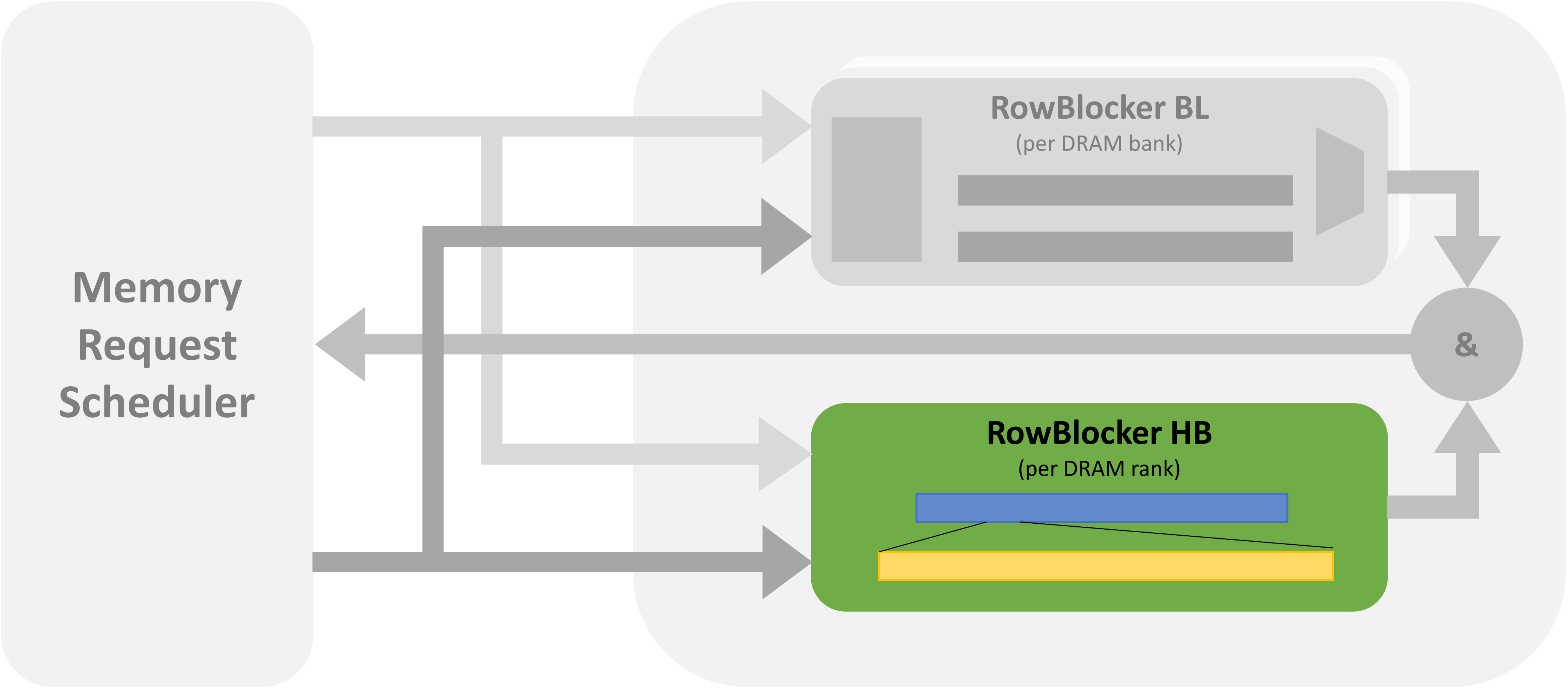
Dual counting Bloom filter



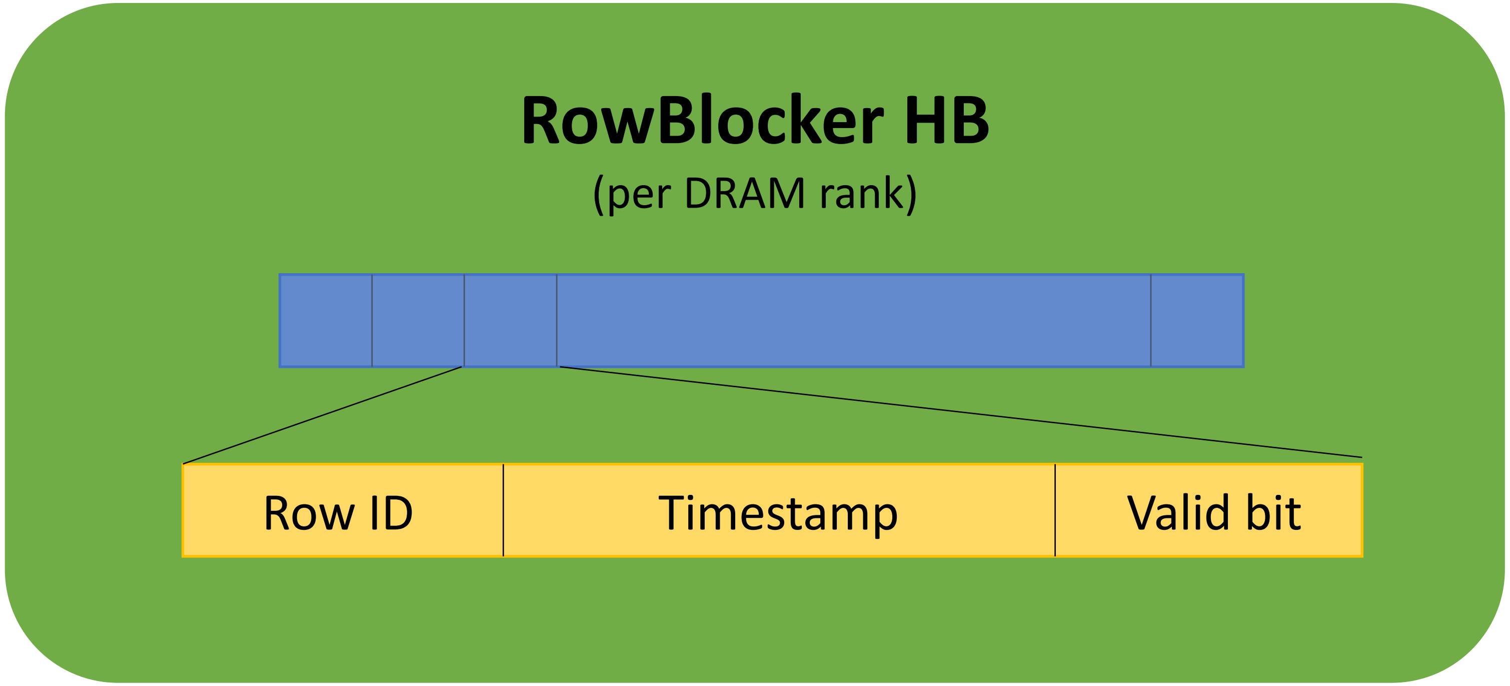


Row Address
Hash Functions
$\{h_{i1}, h_{i2}, \ldots\}$
Filter A: passive
a_1 a_2 a_3 a_m
h_1 h_2 h_3 … h_m
$\{h_{j1}, h_{j2}, \ldots\}$
Filter B: active
b_1 b_2 b_3 b_m
h_1 h_2 h_3 … h_m
b_{j1} b_{j2} …
Min $\{b_{j1}, b_{j2}, \ldots\} > N_{BL}$
Blacklisted

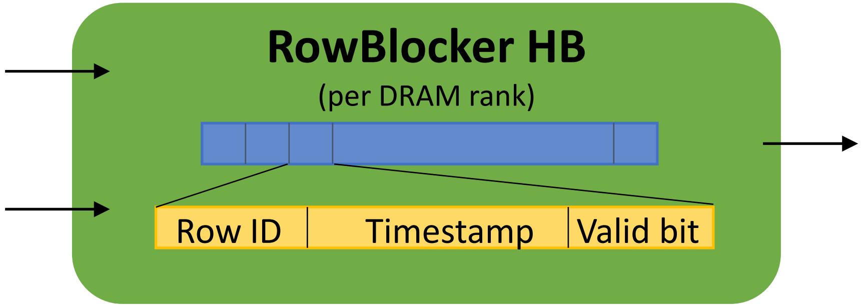
RowBlocker


Memory Request Scheduler
RowBlocker BL
(per DRAM bank)
&
RowBlocker HB
(per DRAM rank)

RowBlocker History Buffer (HB)



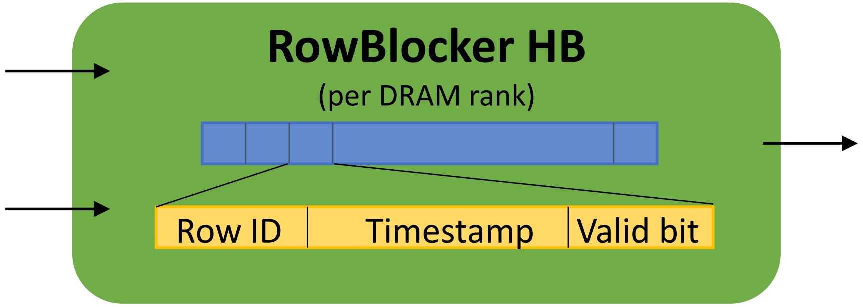
RowBlocker HB



Goal 1: Track which rows were activated recently

Goal 2: Test if current row is one of them

RowBlocker HB



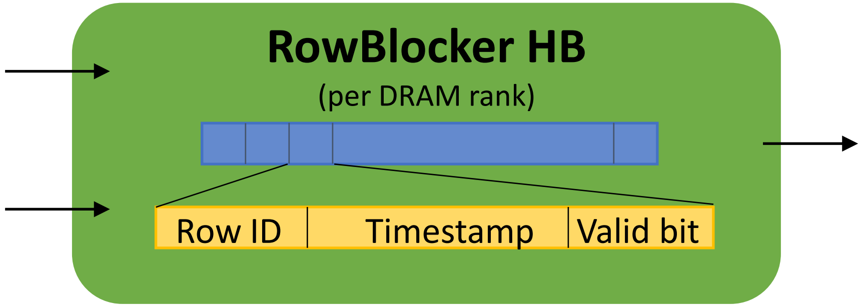
What: circular first-in-first-out (FIFO) queue

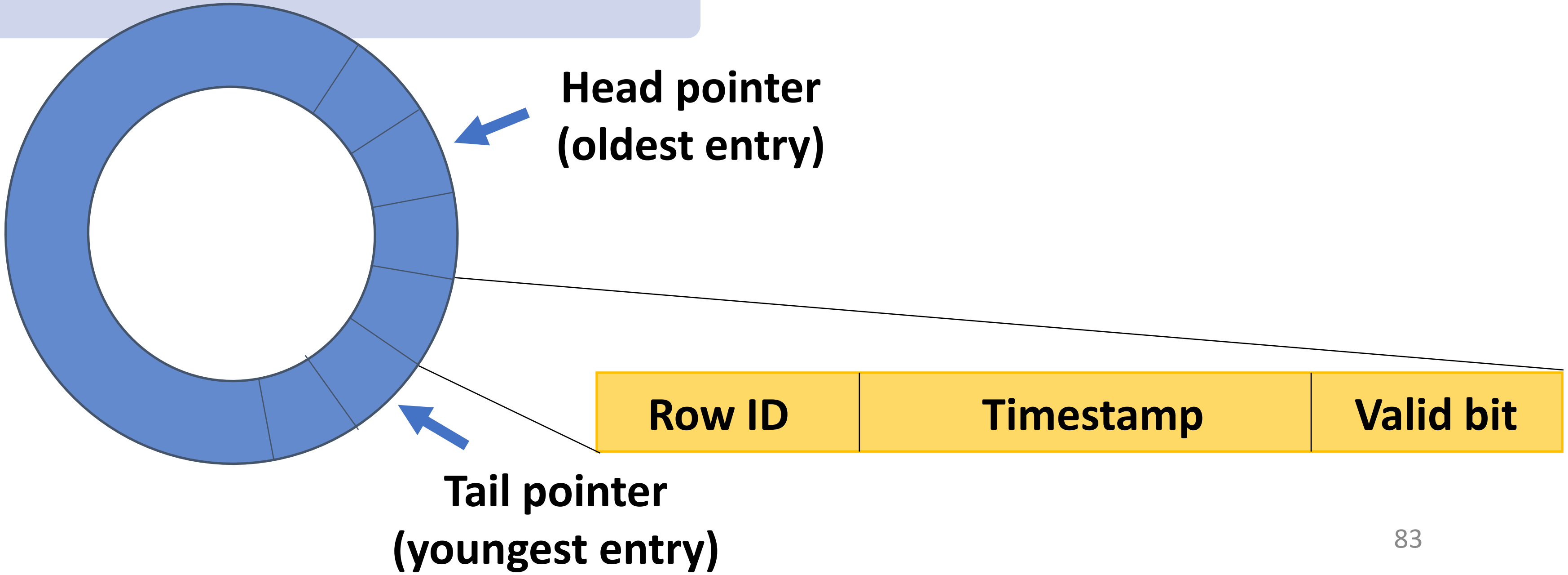
(stores record of rows activated in last t_{delay} time window)

Operations: insert, test, (update)

RowBlocker HB

- Row ID: rank-unique ID for all rows
- Timestamp: current time
- ✓ Valid bit



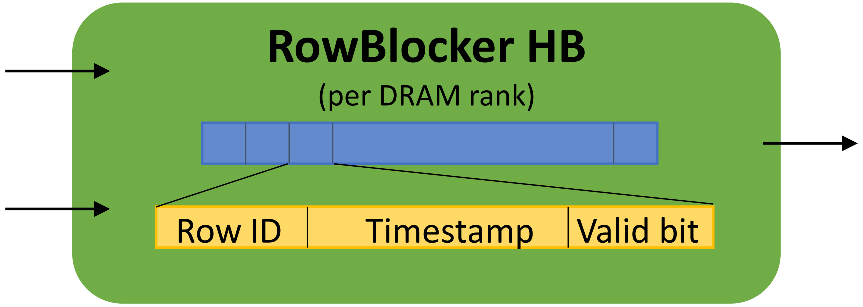


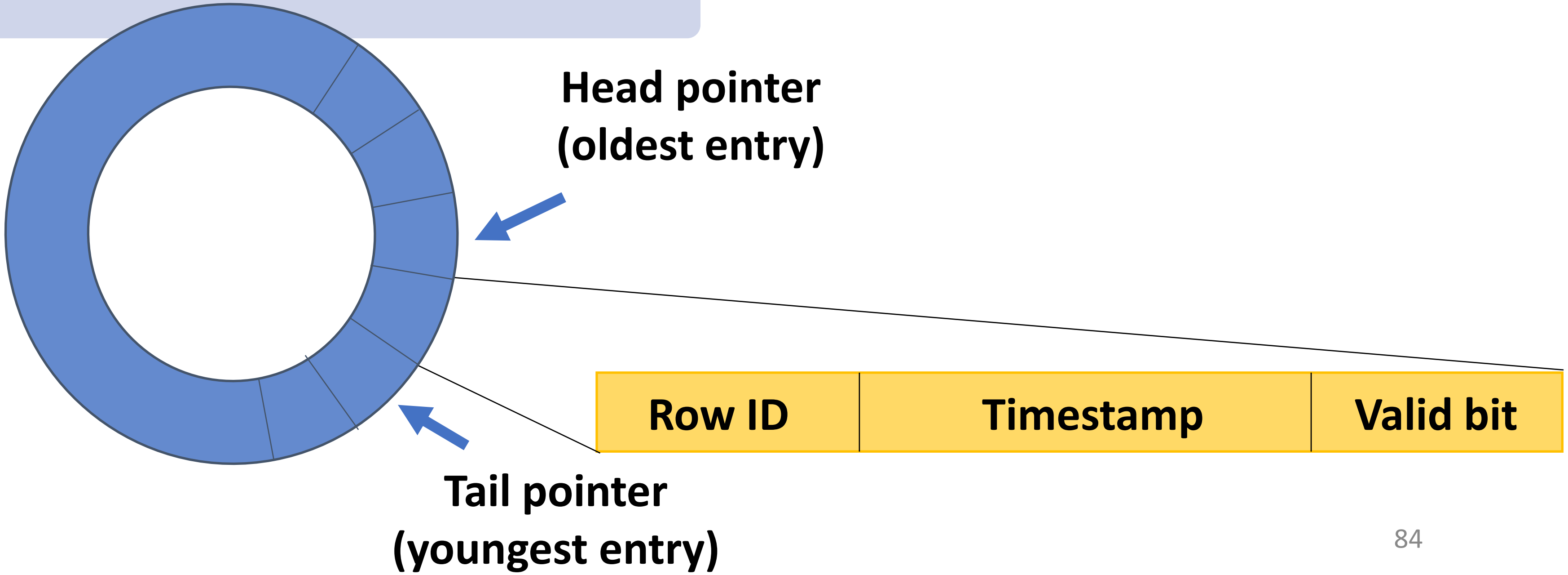
Update

Row ID: rank-unique ID for all rows

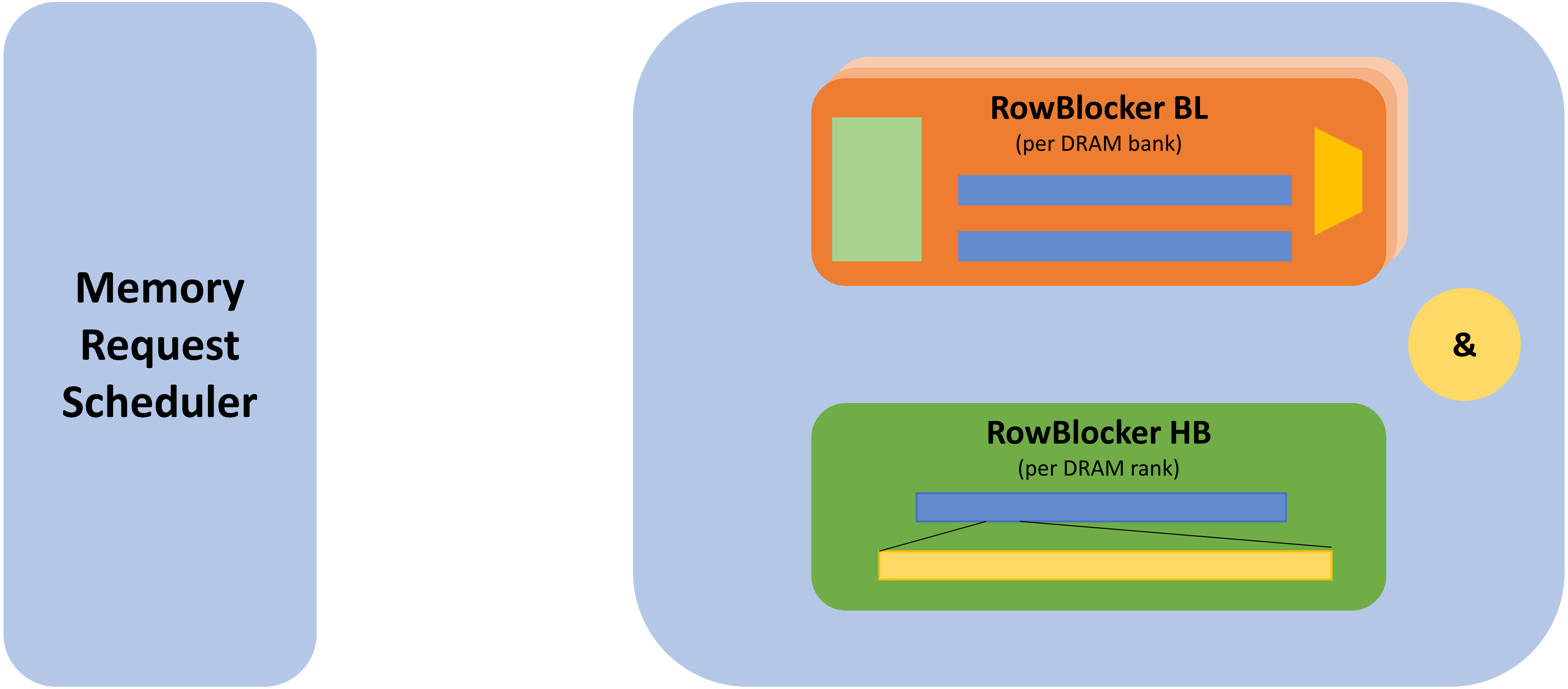
Now - Timestamp >= t_{delay}

Valid bit: **set to 0**

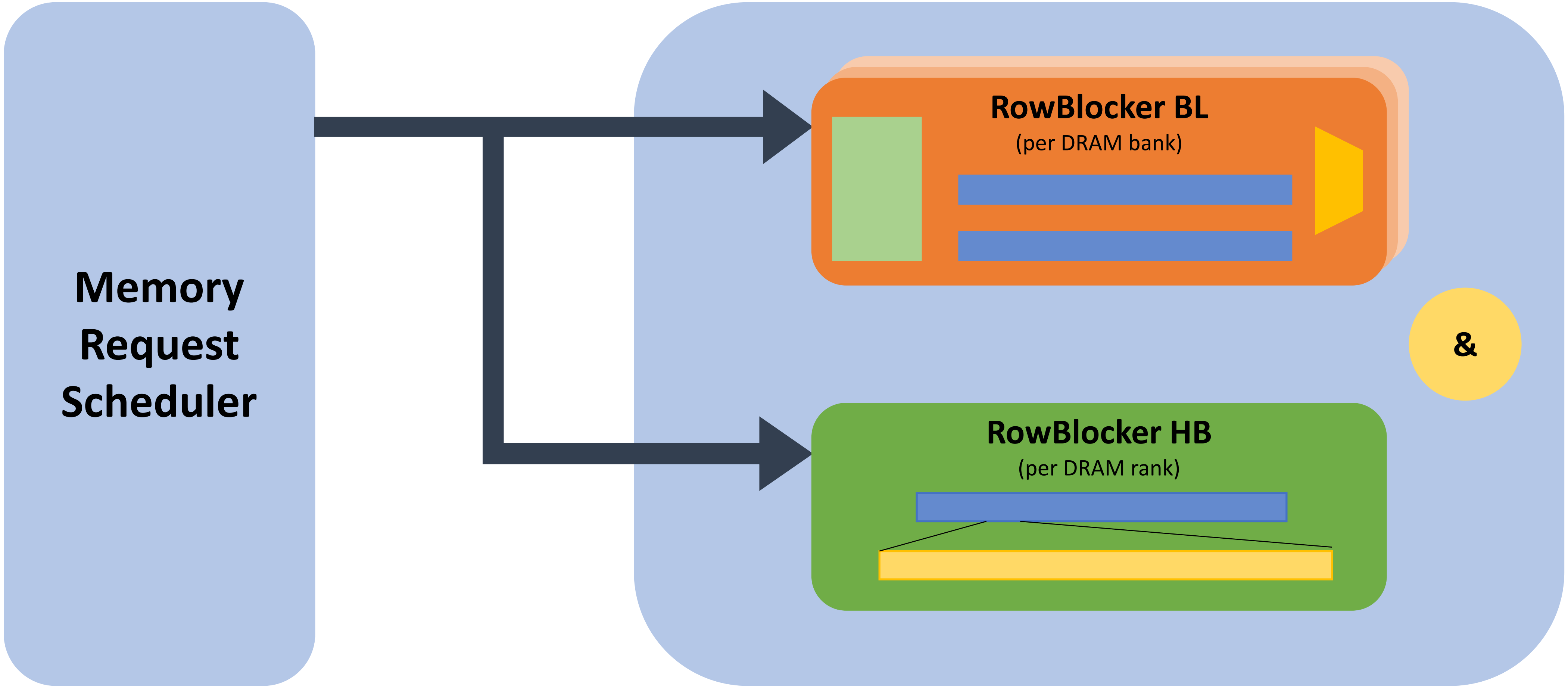




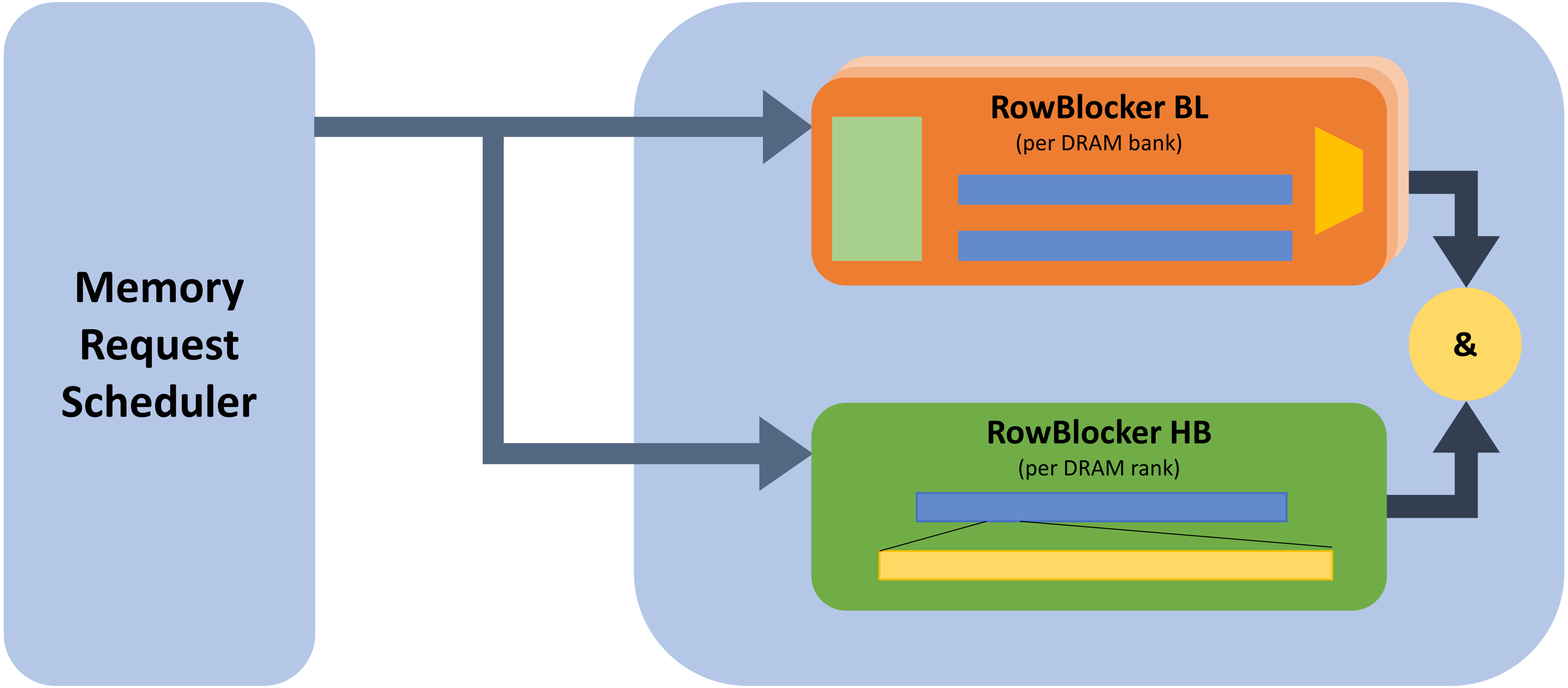
RowBlocker: is this row activation RH-safe?



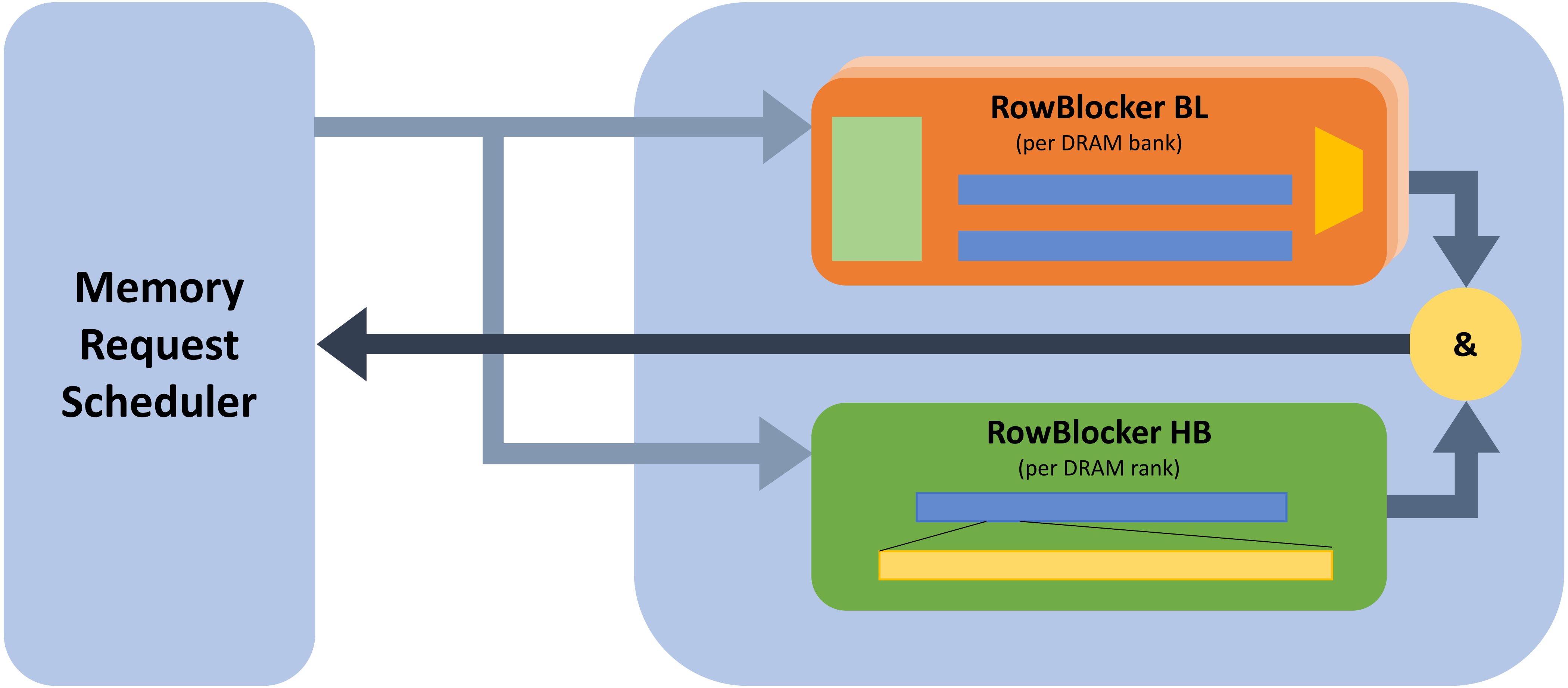
RowBlocker: is this row activation RH-safe?



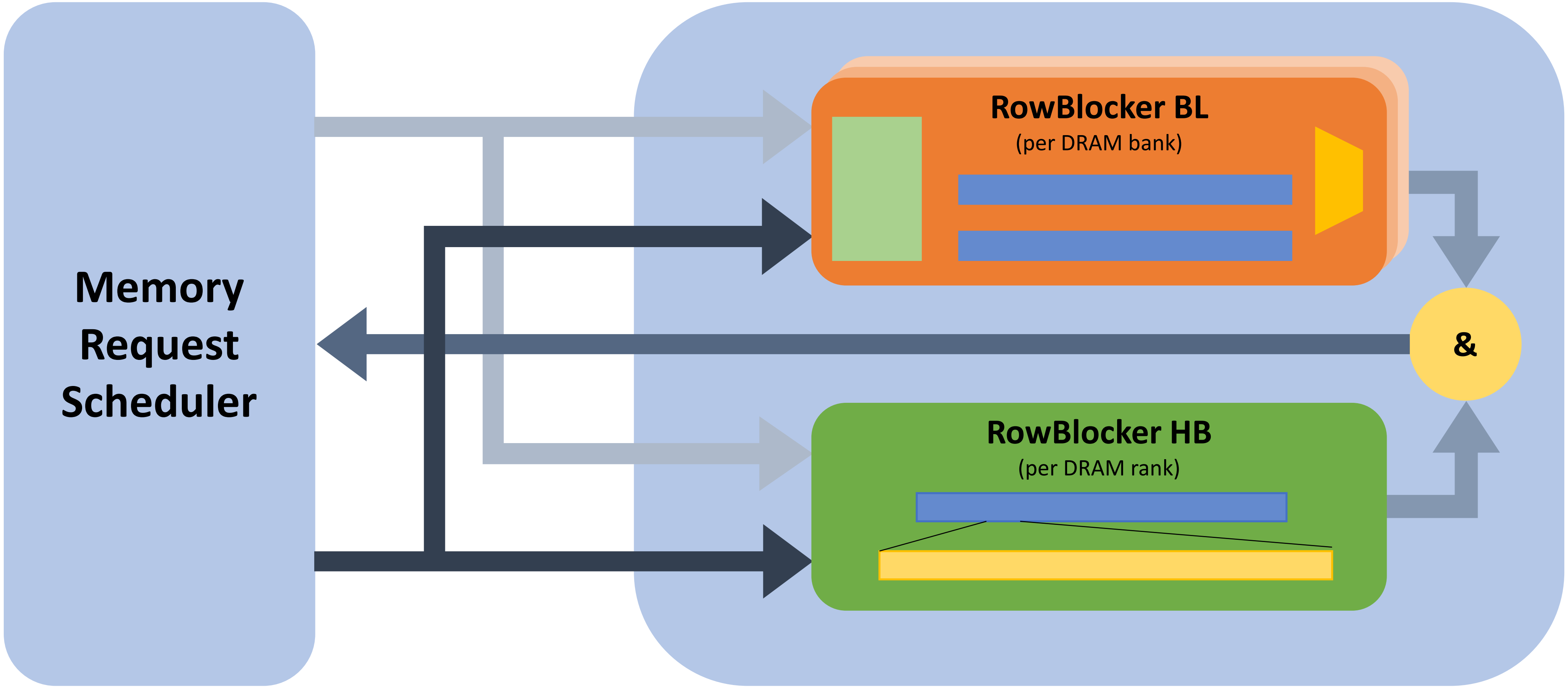
RowBlocker: is this row activation RH-safe?



RowBlocker: is this row activation RH-safe?



RowBlocker: is this row activation RH-safe?



BlockHammer =

RowBlocker

+

AttackThrottler

AttackThrottler

Goal 1: identify potential attacker threads

Goal 2: limit their memory bandwidth usage

AttackThrottler

Goal 1: identify potential attacker threads

Goal 2: limit their memory bandwidth usage

1. Identifying (potential) attacker threads

How: RowHammer Likelihood Index (RHLI)

$$RHLI = \frac{\#\ blacklisted\ row\ activations\ thread\ performs\ to\ DRAM\ bank}{max\ \#\ times\ blacklisted\ row\ can\ be\ activated\ in\ protected\ system}$$

RHLI = 0
(benign threads)

More and more likely to induce bit-flip

Quantifies similarity between a given thread's **memory access pattern** and a **real RowHammer attack**

1. Identifying (potential) attacker threads

Idea: 2 counters per <thread, bank> pair, used same time-interleaving mechanism of D-CBF

2 counters: active + passive counter

- Thread activates blacklisted row in bank → increment both counters
- Only active counter is used to calculate RHLI
- RowBlocker clears active filter in bank → AttackThrottler clears all active counters in bank and switches roles

Calculates RHLI from rows **blacklisted in last two epochs**

AttackThrottler

Goal 1: identify potential attacker threads

Goal 2: limit their memory bandwidth usage

2. Limiting memory bandwidth usage

How: by applying quota to thread's total in-flight memory requests

$$Quota \sim \frac{1}{RHLI}$$

Thread keeps **activating blacklisted row:**
RHLI increases → **quota decreases**

Thread **reaches quota:**
can't make new memory request
(until ongoing request is completed)

Lessens memory bandwidth usage of attacker threads → frees up memory bandwidth for benign threads

AttackThrottler: 3rd goal?

Goal 1: identify potential attacker threads → RHLI

Goal 2: limit their memory bandwidth usage → quota

3. Share info with the Operating System

What: Share <thread, DRAM bank> RHLI values with OS

Goal: mitigate RH attack at software level

e.g., by killing or descheduling attacker thread

Results

We compare BlockHammer with:

Baseline system: no RH mitigation

Three probabilistic mitigation mechanisms: PARA, ProHIT, MRLoc

Three deterministic mitigation mechanisms: CBT, TWiCe, Graphene

Results

Hardware complexity analysis
→ scalable & low cost

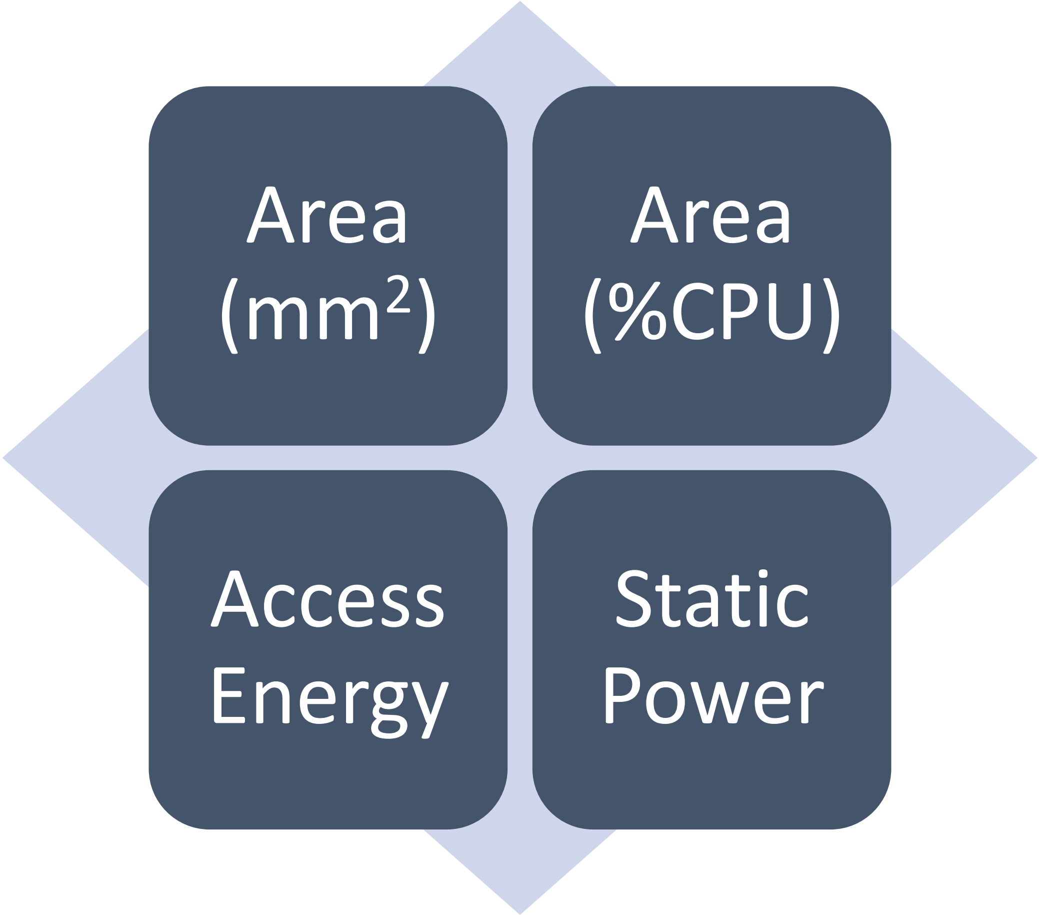
Performance & energy consumption
→ scalable & efficient

Results

Hardware complexity analysis
→ scalable & low cost

Performance & energy consumption
→ scalable & efficient

1. Hardware complexity analysis



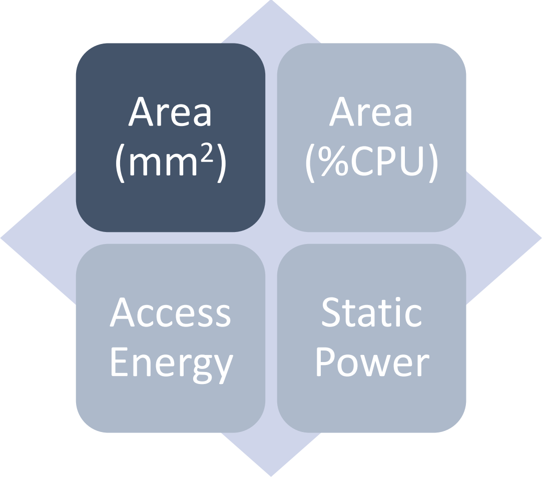
Area (mm^2)

Area (%CPU)

Access Energy

Static Power

1. Hardware complexity analysis



Area (mm^2)

Area (%CPU)

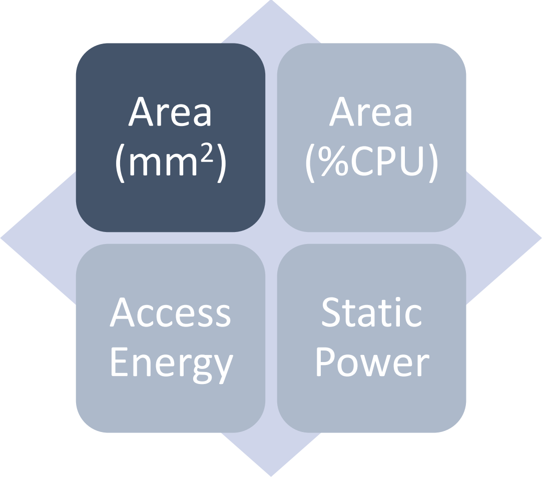
Access Energy

Static Power

RowHammer threshold 32K

- PARA, PRoHIT, MRLoc → extremely area-efficient (because probabilistic)
- Graphene << TWiCe, BlockHammer < CBT

1. Hardware complexity analysis



Area (mm^2)

Area (%CPU)

Access Energy

Static Power

RowHammer threshold 32K

- PARA, PRoHIT, MRLoc → extremely area-efficient (because probabilistic)
- Graphene << TWiCe, BlockHammer < CBT

RowHammer threshold 1K

- Graphene x28.5, TWiCE x34.5, CBT x19.7 ↔ BlockHammer x11.2
- **New order: Graphene < BlockHammer << TWiCE << CBT**
  - **BlockHammer is catching up!**

1. Hardware complexity analysis

Conclusion 1: BlockHammer is **more scalable** than other RowHammer mitigation mechanisms

Conclusion 2: Graphene mostly better than BlockHammer…

for now at least…

- RowHammer will get worse → maybe < 1K? (currently at 9.6K)
- Graphene does not scale as well!

Results

Hardware complexity analysis
→ scalable & low cost

Performance & energy consumption
→ scalable & efficient

2. Performance & energy consumption

Single-core system performance

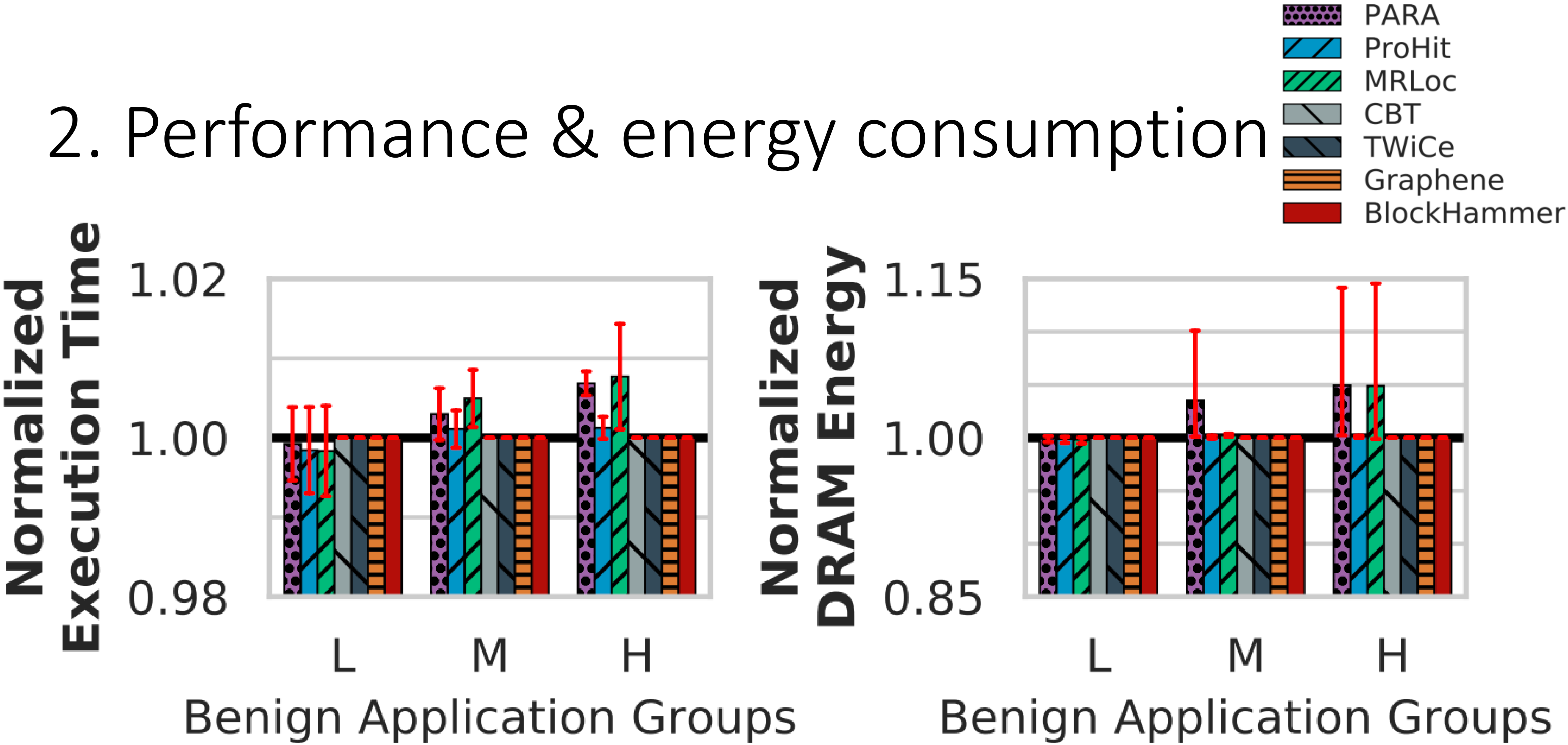
Eight-core system performance

- Without RH attack
- With RH attack

Scalability

- Without RH attack
- With RH attack

2. Performance & energy consumption



BlockHammer has no performance or energy overhead for single-core benign applications

2. Performance & energy consumption

Single-core system performance

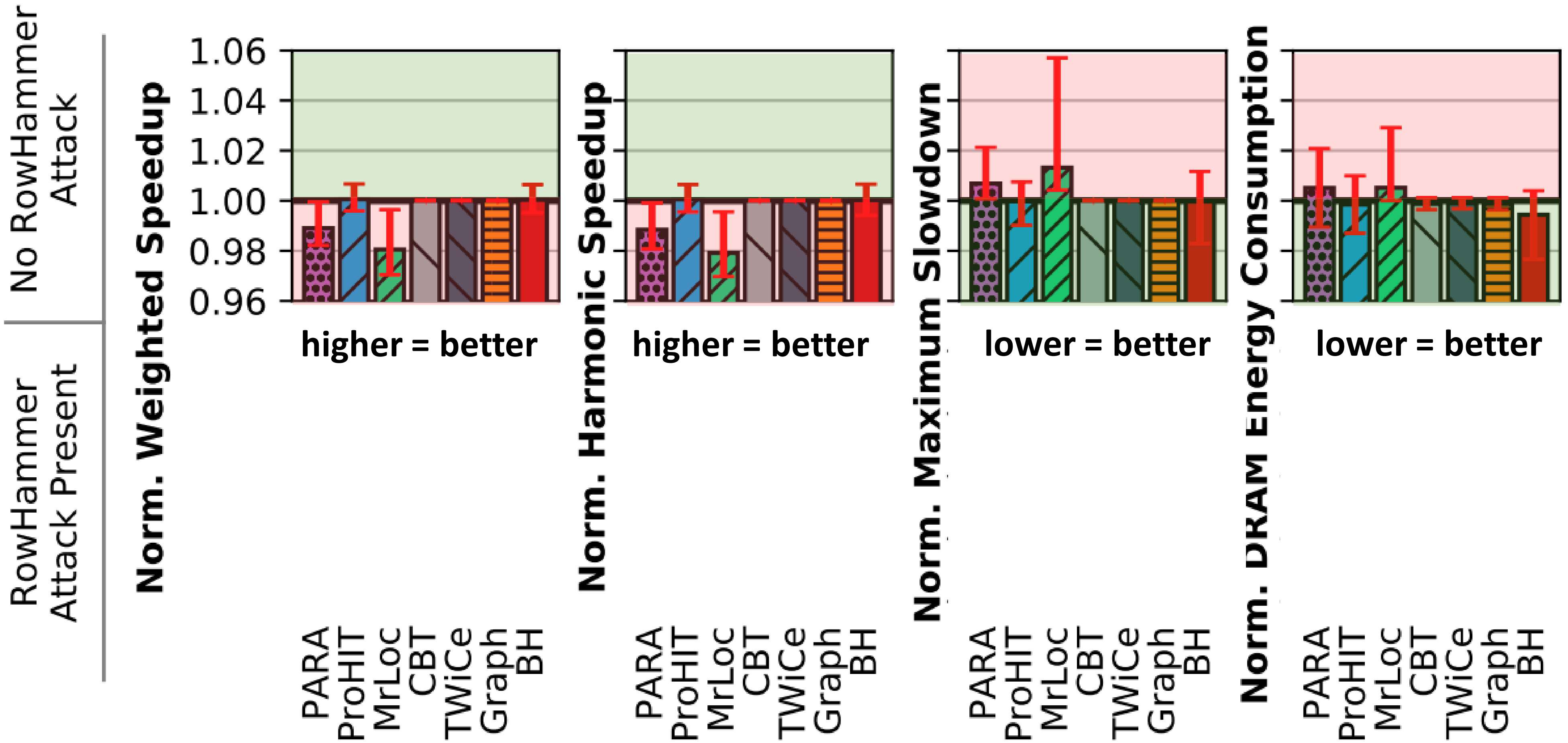
Eight-core system performance

- Without RH attack **(8B)**
- With RH attack **(7B, 1RH)**

Scalability

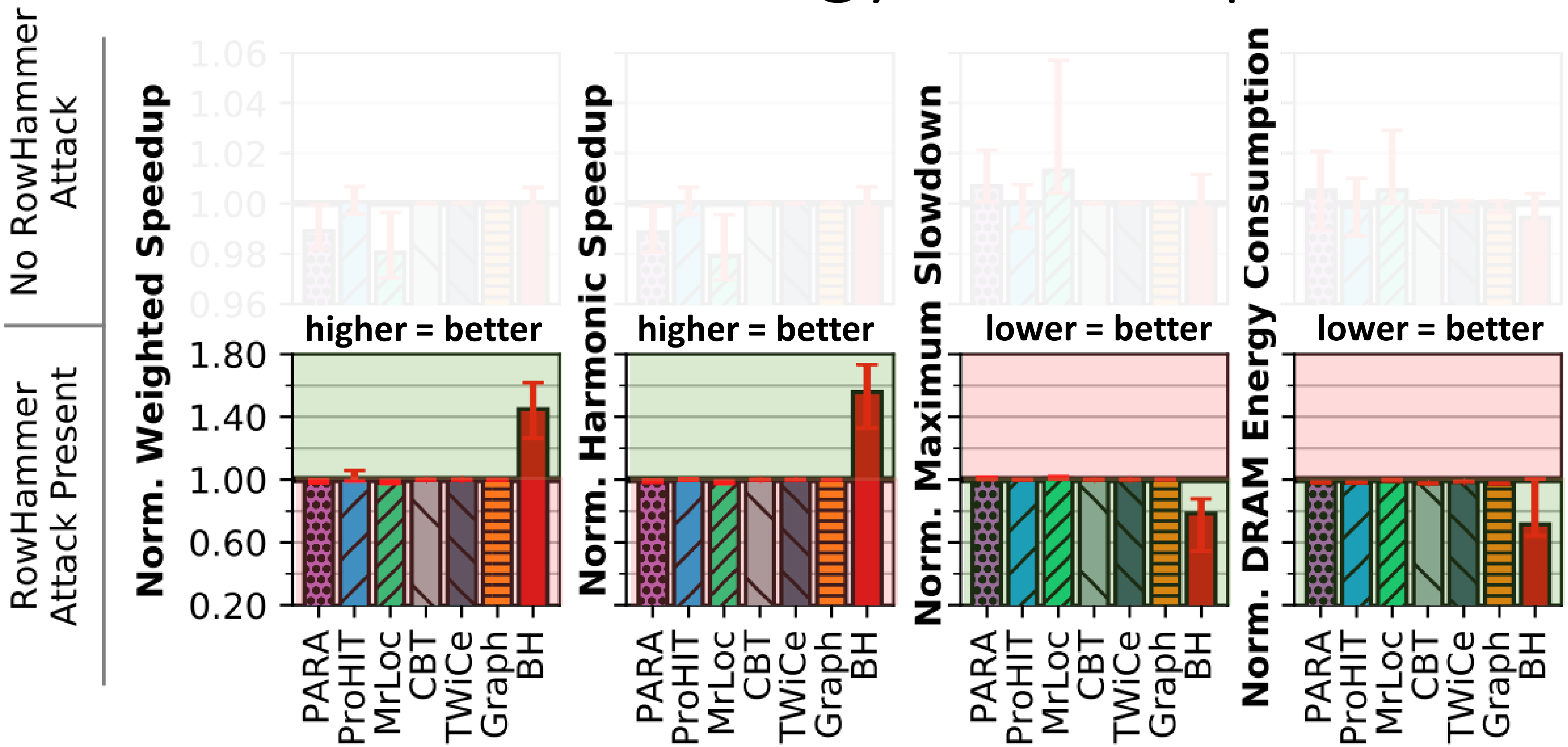
- Without RH attack
- With RH attack

2. Performance & energy consumption



BlockHammer has competitive performance and energy consumption when no attack is present

2. Performance & energy consumption



BlockHammer has much higher performance of benign applications and lower DRAM energy consumption when attack is present

2. Performance & energy consumption

Single-core system performance

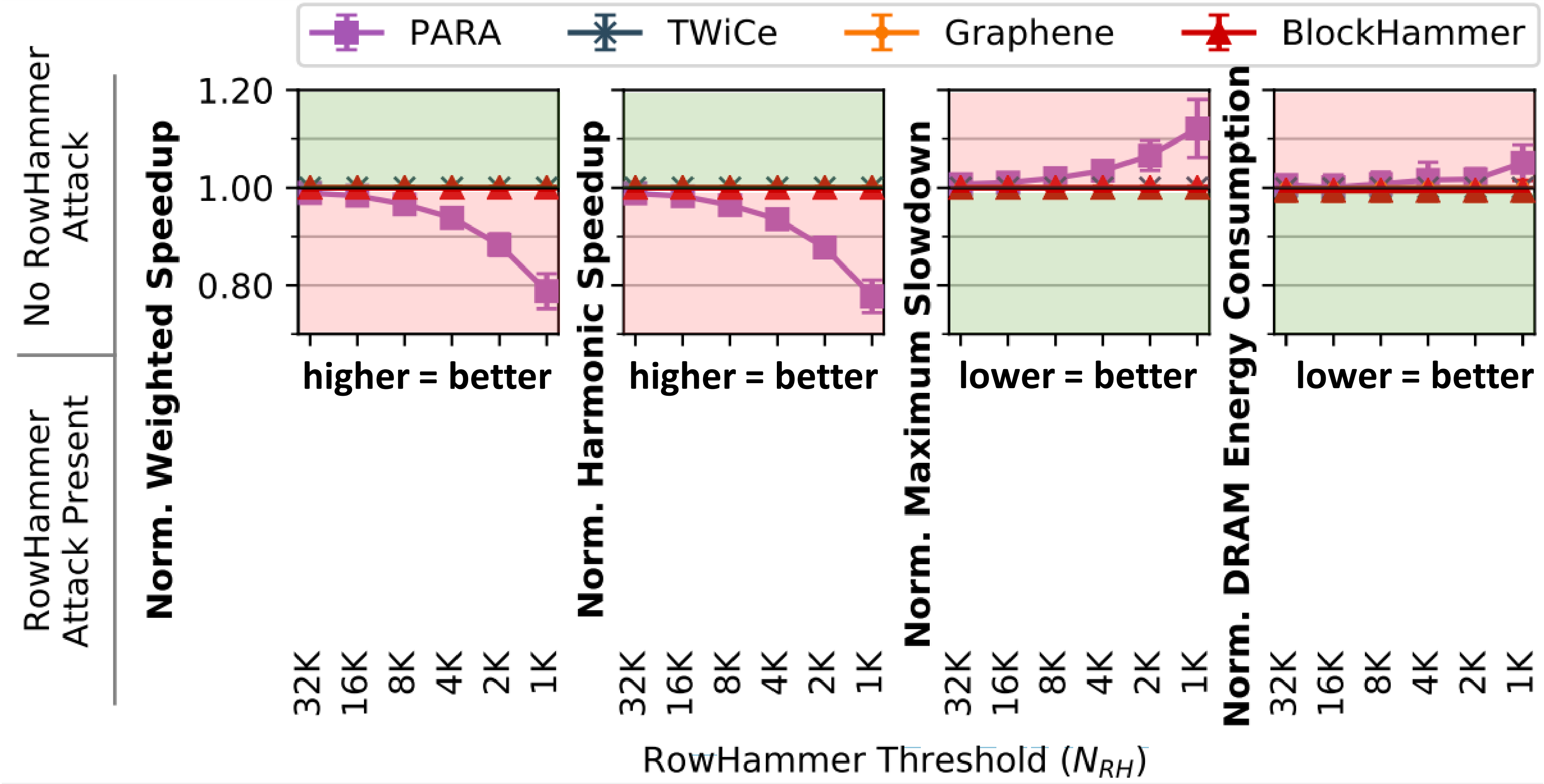
Eight-core system performance

- Without RH attack
- With RH attack

Scalability

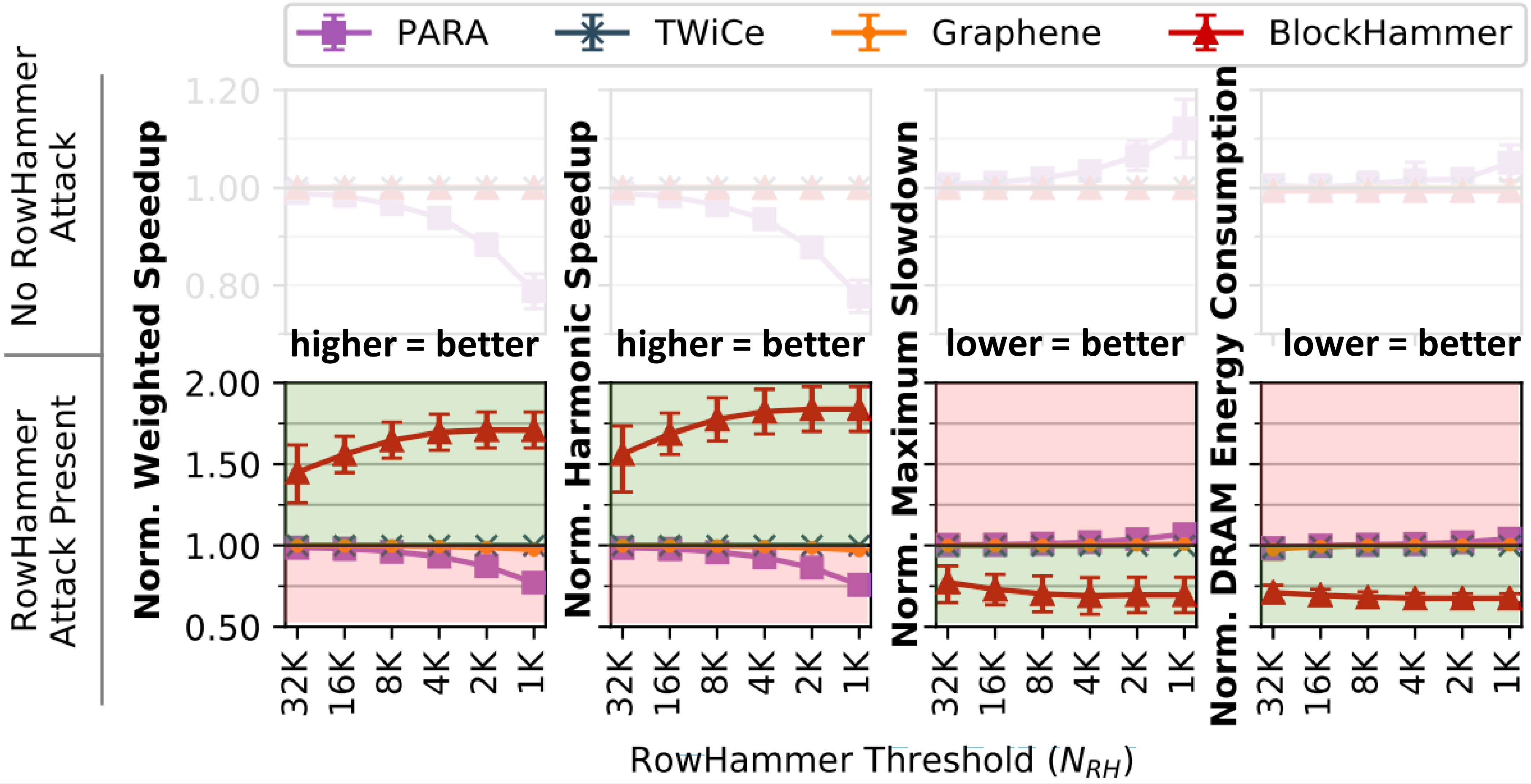
- Without RH attack
- With RH attack

2. Performance & energy consumption



BlockHammer has negligible performance and energy consumption overheads and still does if RH worsens (when no attack is present)

2. Performance & energy consumption



BlockHammer has significantly better performance and lower energy consumption as RH worsens (when attack is present)

2. Performance & energy consumption

Conclusion 1: When the system is **not under attack,** BlockHammer is **competitive** with the other state-of-the-art mechanisms, also **at the lowest RH thresholds**

Conclusion 2: In the presence of a RH attack, BlockHammer has **significantly better performance and lower energy consumption** than all other state-of-the-art mechanisms, **even at lower RH thresholds**

Summary

Summary & Conclusion

Problem:

- Memory density scaling of DRAM chips causes increasing vulnerability to RowHammer, but most solutions can't scale accordingly
- Current solutions often require knowledge of or modification to DRAM internals

Goal:

- Find a scalable and efficient way to prevent RowHammer, without knowledge of or modification to DRAM internals

Mechanisms:

- RowBlocker: tracking all row activations efficiently (by using Bloom filters) and throttling RowHammer unsafe row accesses
- AttackThrottler: identifying (RHLI) and throttling (quota) potential attacker threads

Results:

- Hardware complexity: most scalable solution (Graphene currently more efficient but not as scalable)
- Performance & energy:
  - No RowHammer attack: competitive, even at lower RH thresholds
  - RowHammer attack: significantly better than all other solutions

Strengths & Weaknesses

Strengths

- BlockHammer still **scales well** when DRAM chips are getting more vulnerable to RowHammer
- Implementation requires no knowledge of or modifications to DRAM internals (**completely implemented in memory controller**)
- Makes **distinction** between **benign** applications and potential **attacks**
- Introduces many **new concepts** and even more possible **improvements**
- Innovative idea → groundwork for **new type of RowHammer mitigation**: proactive throttling

Weaknesses

- Completely implemented in memory controller → **cannot be implemented in already manufactured processor chips**

- Some **empirically-determined parameters** (e.g., Bloom filter size)
  - Partially determines false positive rate → room for improvement!

- Evaluation is simulated on **DDR4-based** memory subsystem → what about LPDDR4?
  - Results probably similar
  - And hardware designers will redo it anyway…

Discussion

Discussion

- Should we always aim for deterministic solutions or are probabilistic methods not that bad?
- Can we lower BlockHammer's hardware complexity by adopting a probabilistic approach? What would you change in BlockHammer to achieve that?
  - Remember:

    BlockHammer = RowBlocker (D-CBF + HB) + AttackThrottler (RHLI + quota)
- Is it a good idea to modify BlockHammer into a probabilistic mitigation mechanism? Why (not)?
- Are there other ways to reduce BlockHammer's hardware complexity?

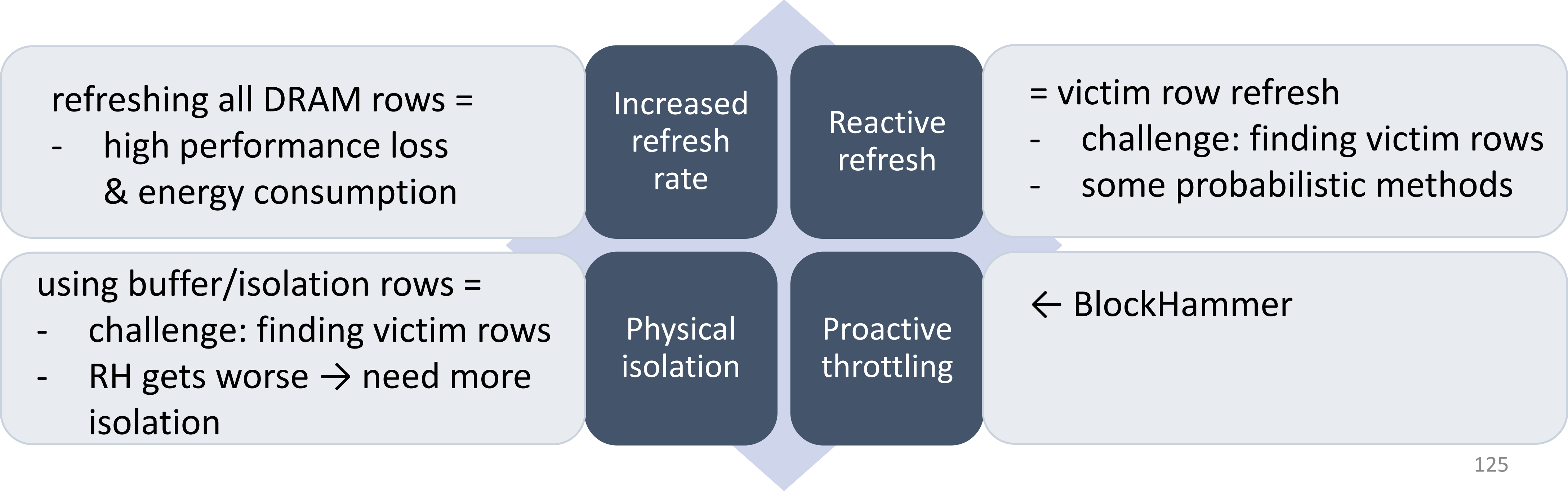
Discussion

- Once we can quickly reverse-engineer DRAM address mappings, will BlockHammer still be the best approach?
- What would be the ideal RowHammer mitigation mechanism and why?

Discussion

- Do you think we can combine (parts of) BlockHammer with other mitigation mechanisms? What would be the (dis)advantages?
  - Remember:

    BlockHammer = RowBlocker (D-CBF + HB) + AttackThrottler (RHLI + quota)
- Do you have any other ideas to improve BlockHammer?



refreshing all DRAM rows =
- high performance loss & energy consumption

Increased refresh rate

Reactive refresh

= victim row refresh
- challenge: finding victim rows
- some probabilistic methods

using buffer/isolation rows =
- challenge: finding victim rows
- RH gets worse → need more isolation

Physical isolation

Proactive throttling

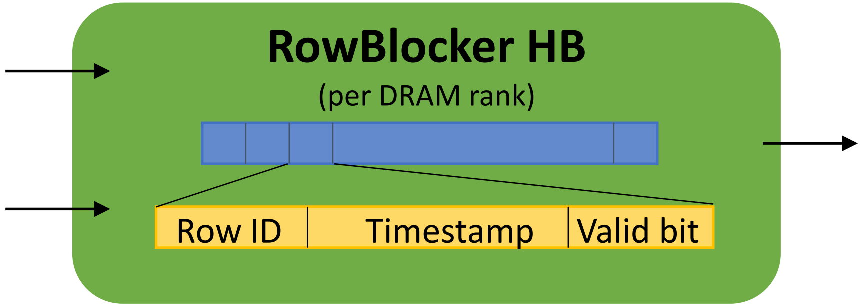
← BlockHammer

Discussion

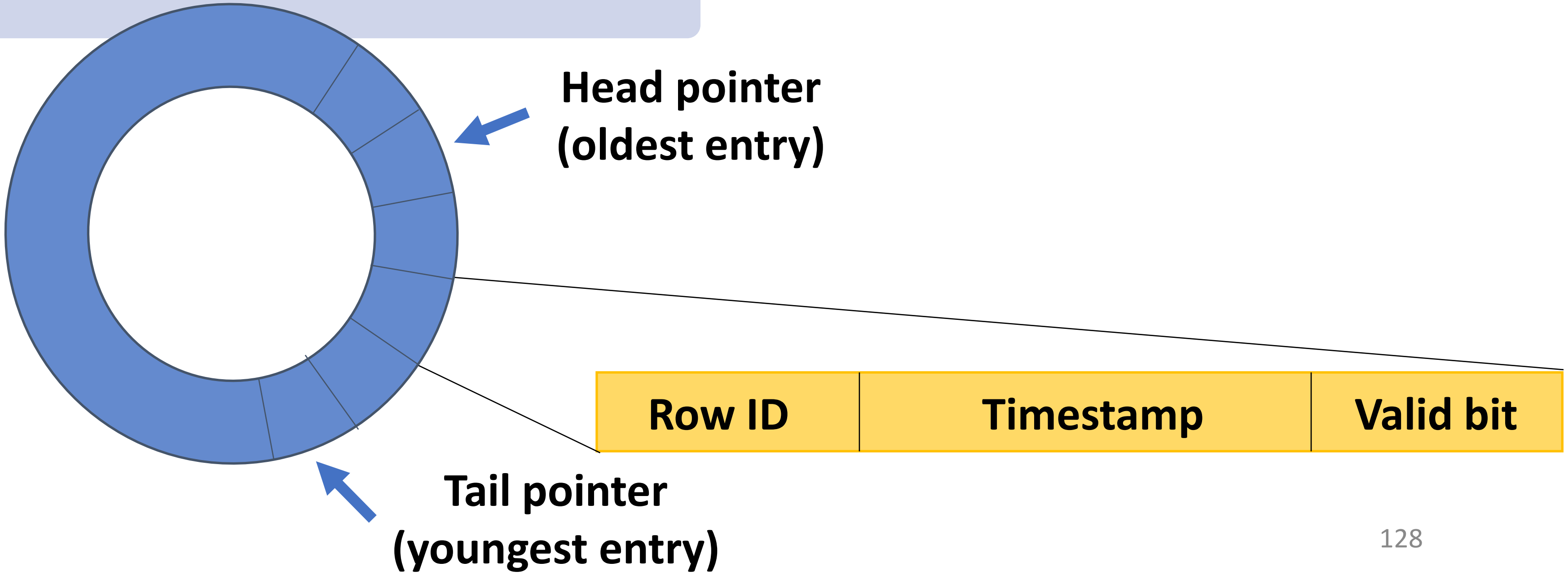
- What can we do with the RHLI at the software level?
  - E.g. killing or descheduling a thread
  - What problems would you encounter?

Backup Slides

Insert

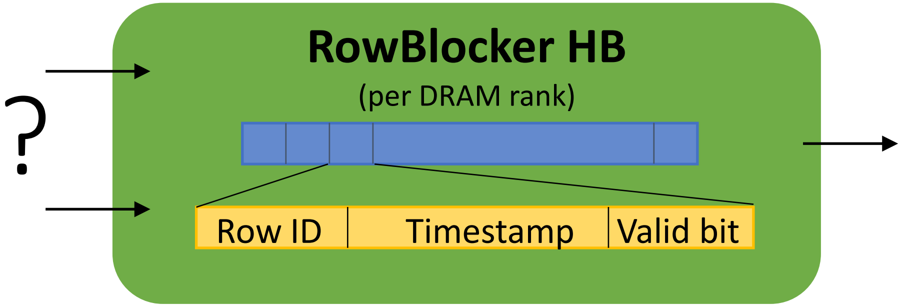


- Row ID: rank-unique ID for all rows
- Timestamp: current time
- ✓ Valid bit: **set to 1**

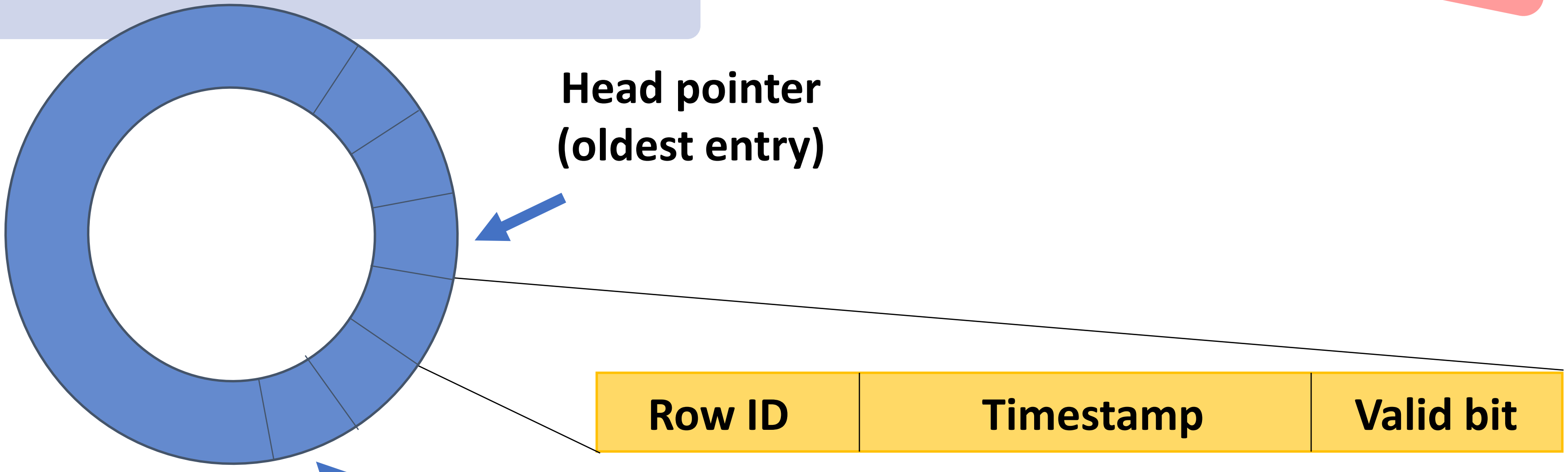


Test: row recently activated?

- Row ID **== to be accessed row**
- Timestamp
- Valid bit **== 1**

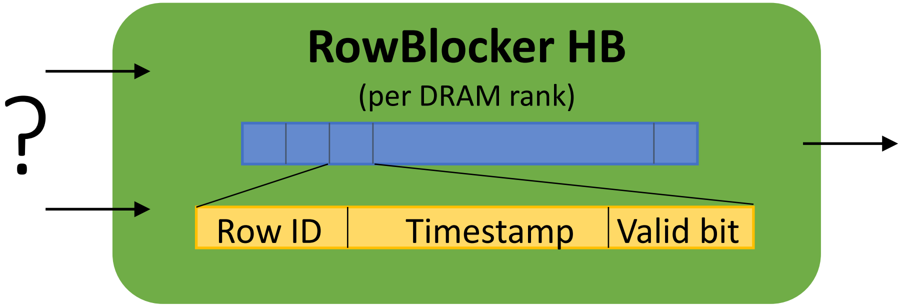


We want low latency!

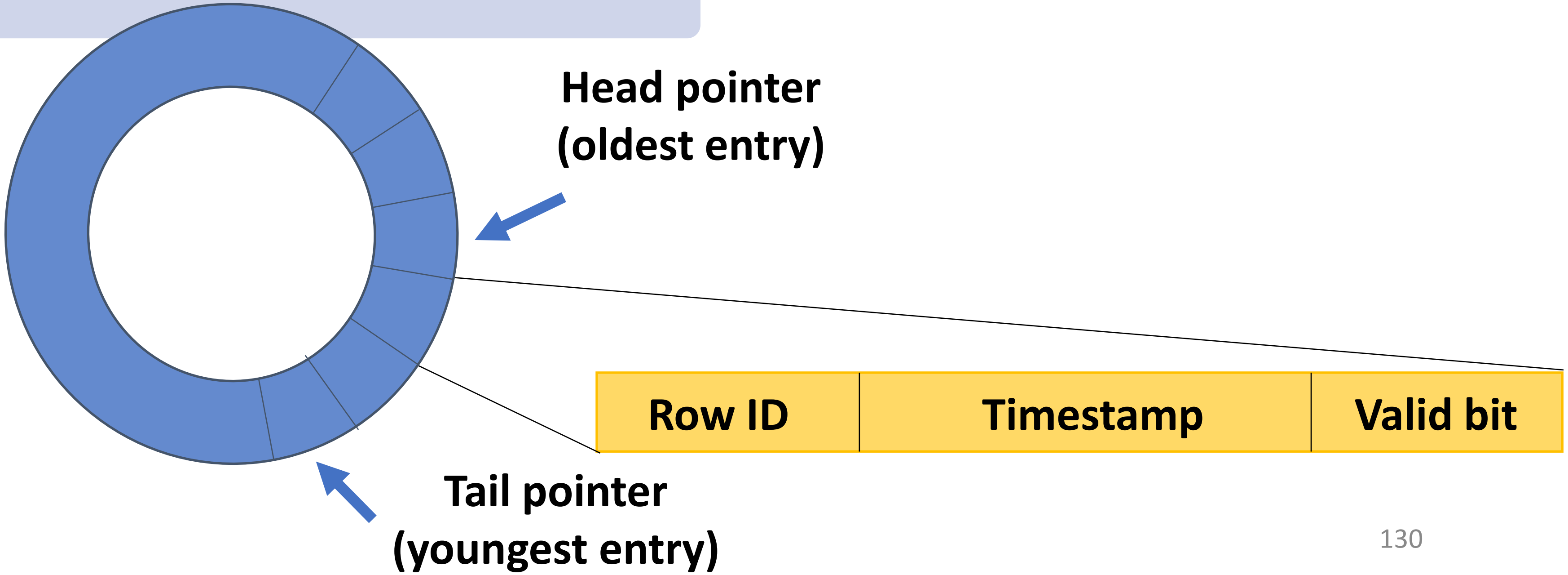

Head pointer (oldest entry)
Row ID
Timestamp
Valid bit
Tail pointer (youngest entry)

Test: row recently activated?

- Row ID **== to be accessed row**
- Timestamp
- ✓ Valid bit **== 1**



Store row addresses in CAM


Head pointer (oldest entry)
Tail pointer (youngest entry)
Row ID
Timestamp
Valid bit

Comparison

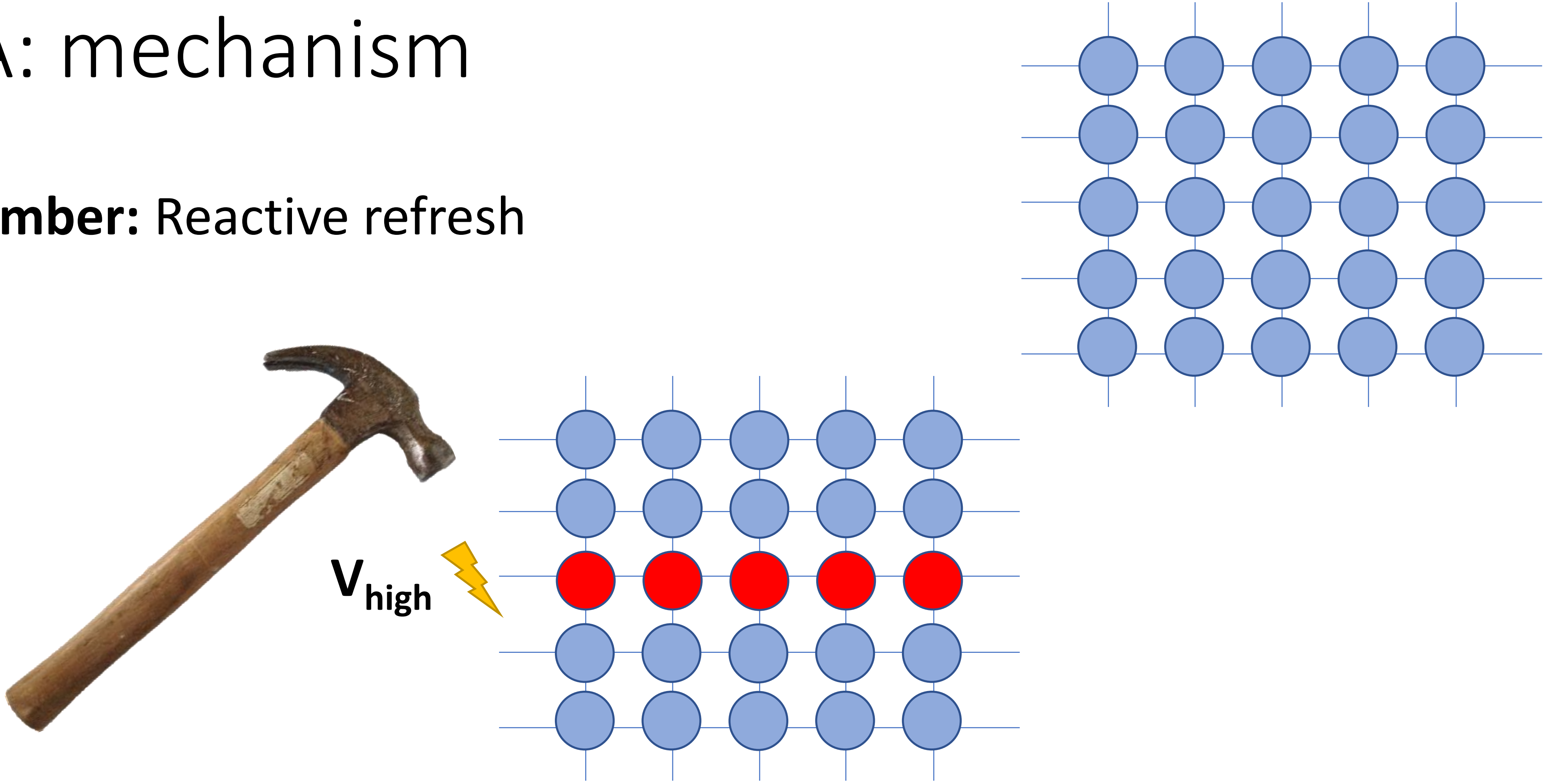
- Compare BlockHammer with
  - (Baseline system: no RH mitigation)
  - **3 probabilistic mitigation mechanisms** (errors still possible)
    - **PARA**
    - **ProHIT**
    - **MRLoc**
  - 3 deterministic mitigation mechanisms (usually area overhead)
    - CBT
    - TWiCe
    - Graphene

PARA: definition

- = Probabilistic Adjacent Row Activation
- Row gets activated → adjacent rows get activated (= refreshed) with probability p

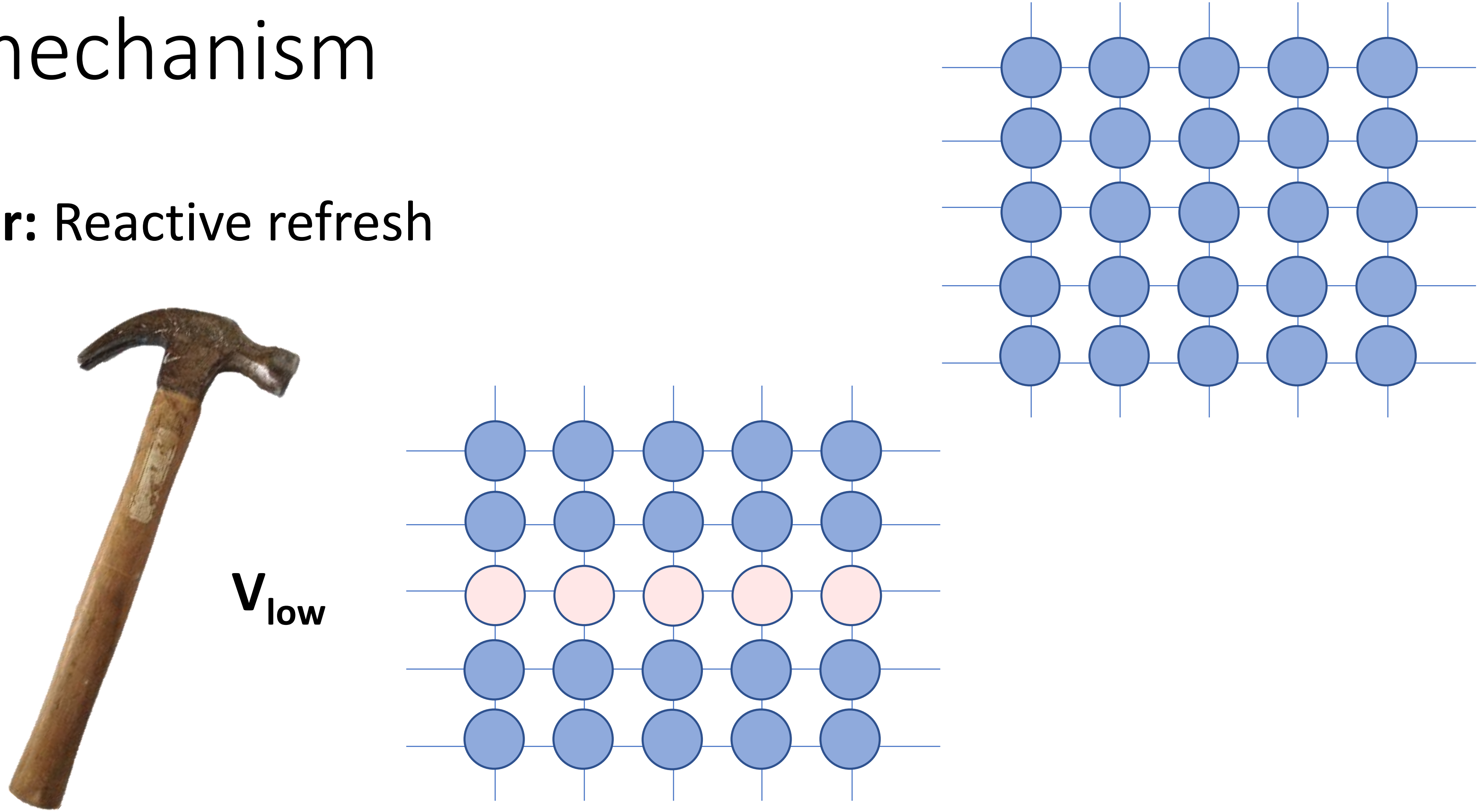
PARA: mechanism

- **Remember:** Reactive refresh



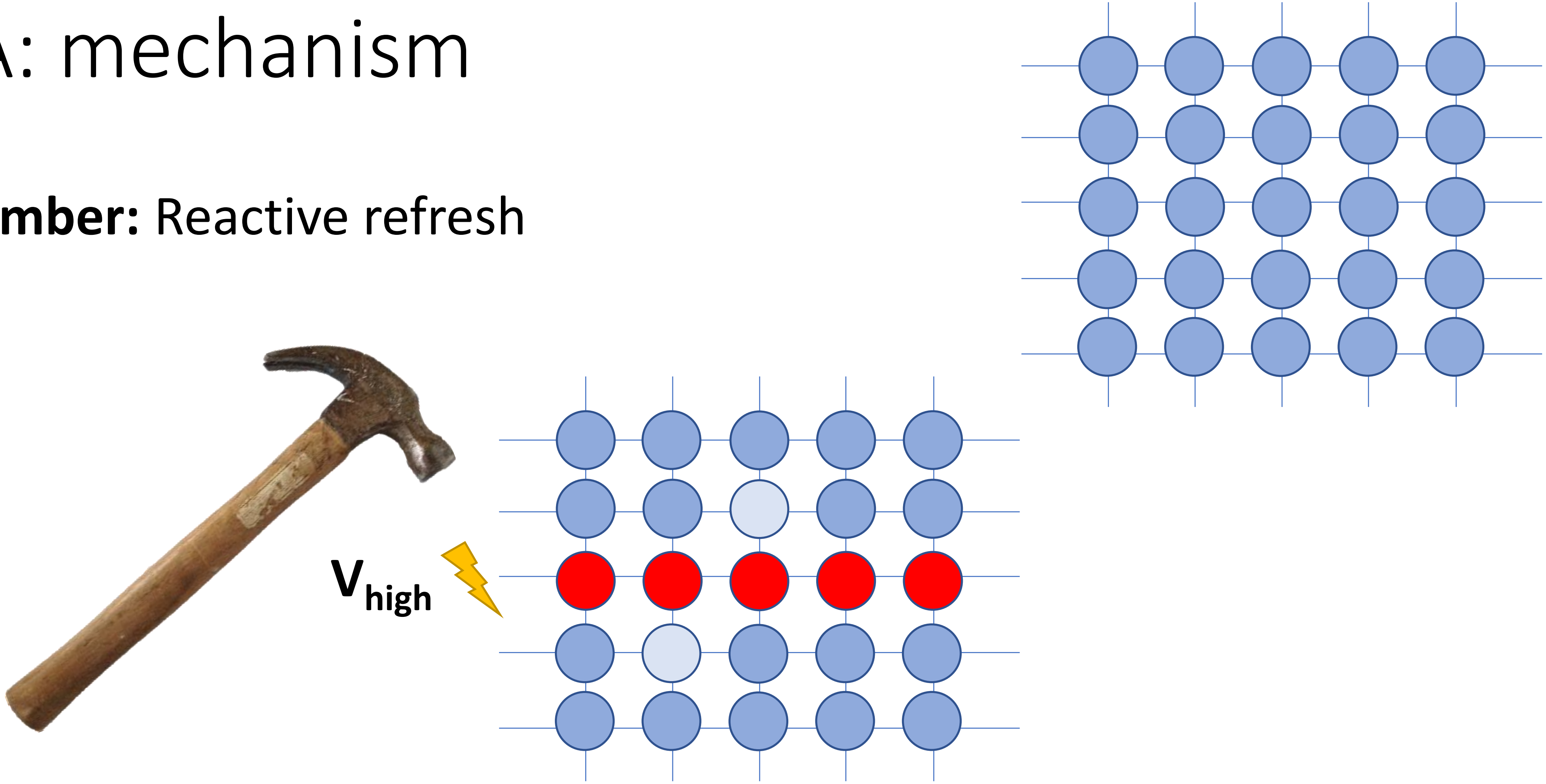
PARA: mechanism

- **Remember:** Reactive refresh



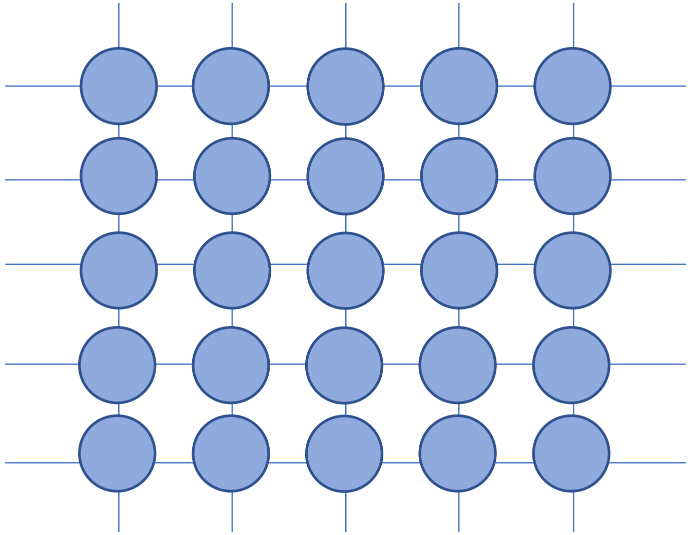
PARA: mechanism

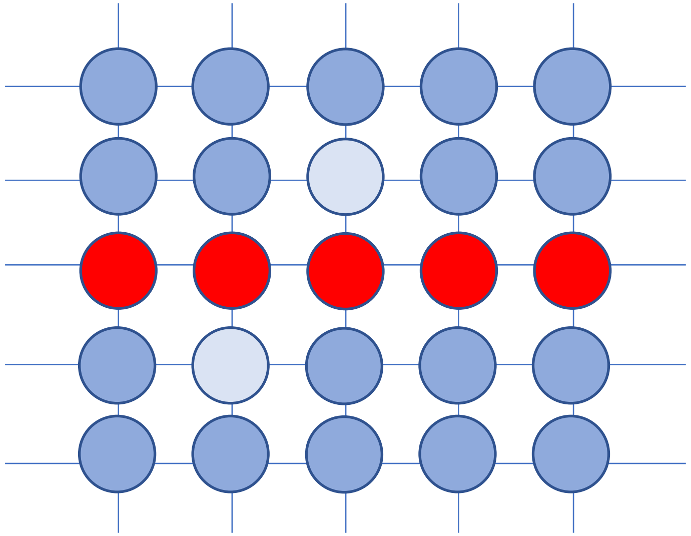
- **Remember:** Reactive refresh



PARA: mechanism

- **Remember:** Reactive refresh



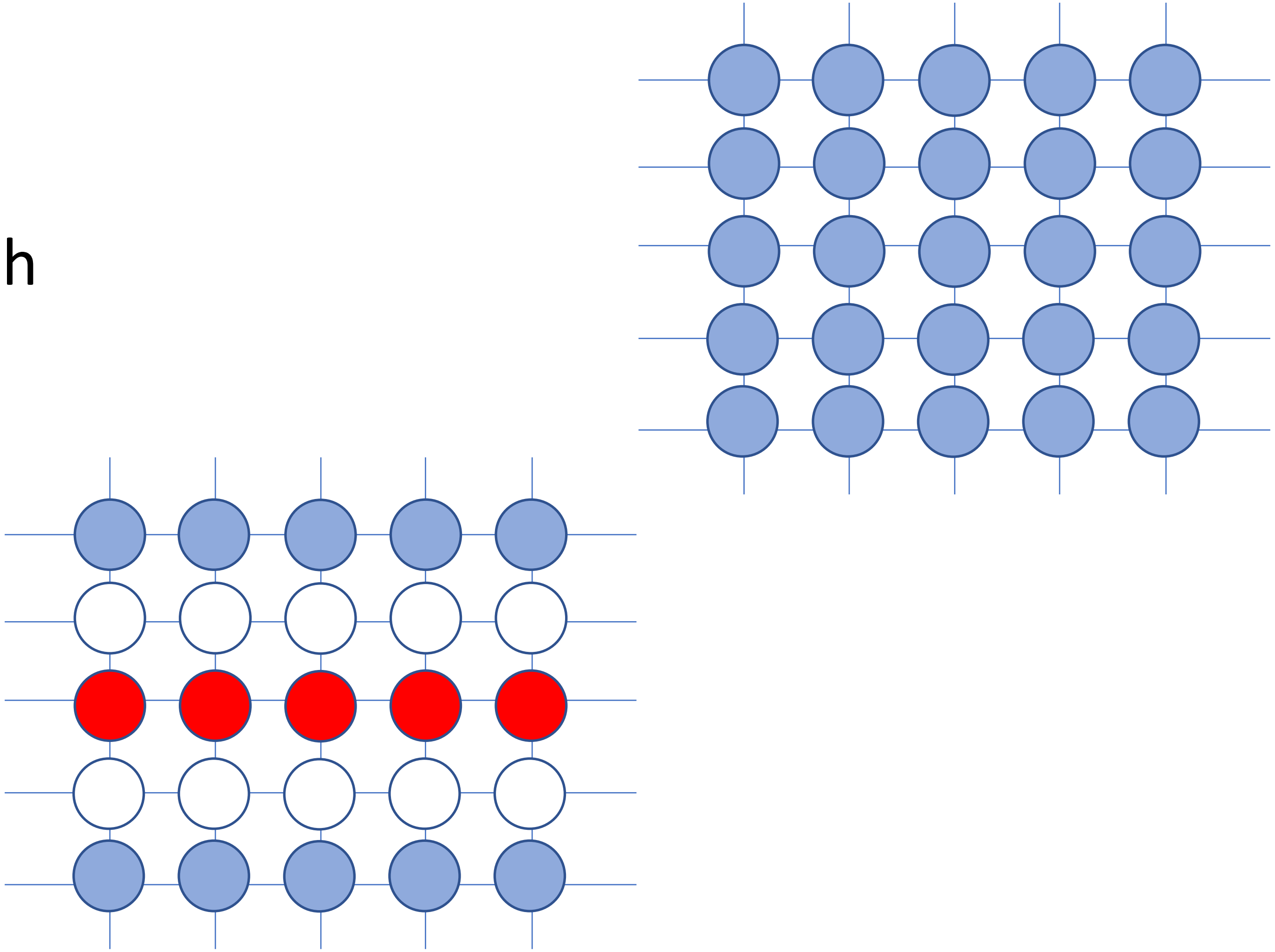


REFRESH
with probability p

REFRESH
with probability p

PARA: mechanism

- **Remember:** Reactive refresh

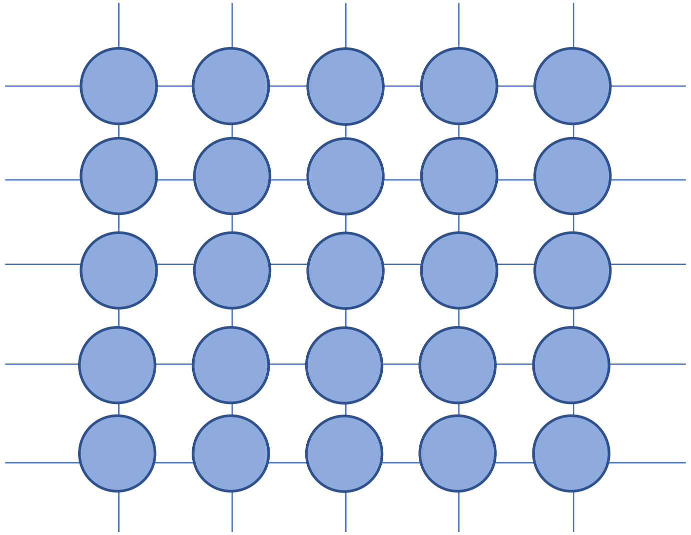


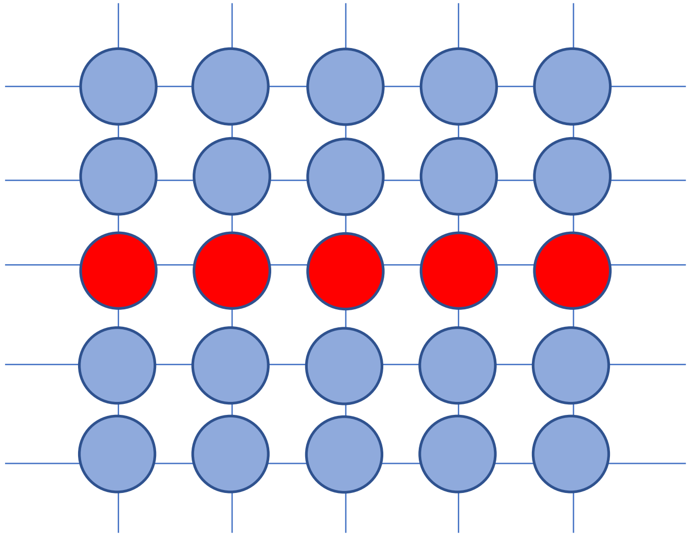
REFRESH
with probability p

REFRESH
with probability p

PARA: mechanism

- **Remember:** Reactive refresh





REFRESH

with probability p

REFRESH

with probability p

PARA: weaknesses

- Cannot prevent bit-flips with 100% certainty (probabilistic!)
- Performance → vulnerable to applications with mix of few frequently activated rows and many randomly activated ones (often the case in memory-intensive programs) → solution: ProHIT
- Knowledge on in-DRAM mapping needed

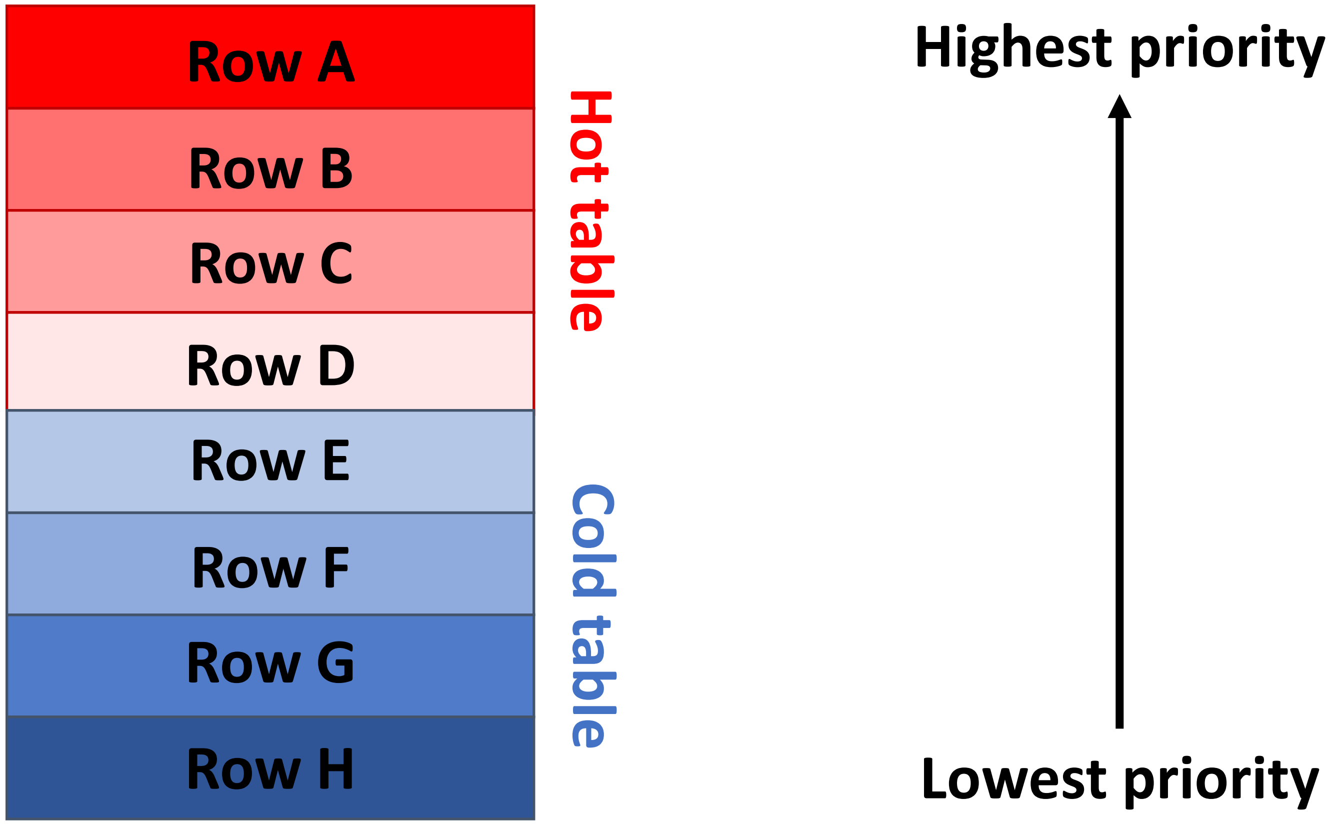
ProHIT: definition

- Based on PARA
- Selects victim rows by **considering the access patterns** of applications (on top of probabilistic selection) → done by Probabilistic History Table

- Key operations: row activation →
  - Probabilistic table promotion (from cold to hot)
  - Probabilistic promotion (from hot to hotter, i.e. higher priority)
  - Probabilistic insertion (into highest priority cold table slot)
  - Probabilistic eviction (one of the cold entries is evicted)

ProHIT: mechanism

Activate row K



Row A

Row B

Row C

Row D

Row E

Row F

Row G

Row H

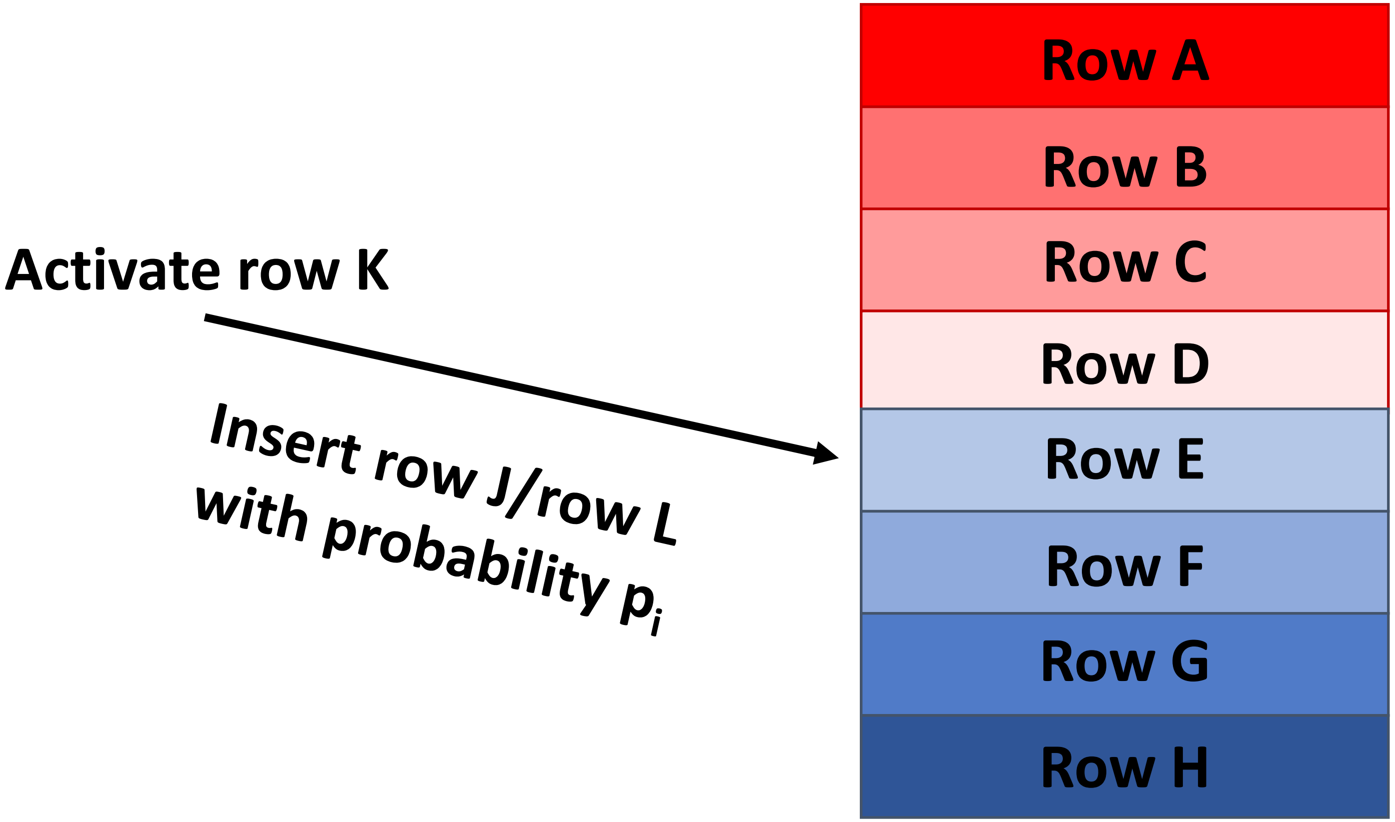
Hot table

Cold table

Highest priority

Lowest priority

ProHIT: mechanism



Activate row K

Insert row J/row L with probability p_i

Row A

Row B

Row C

Row D

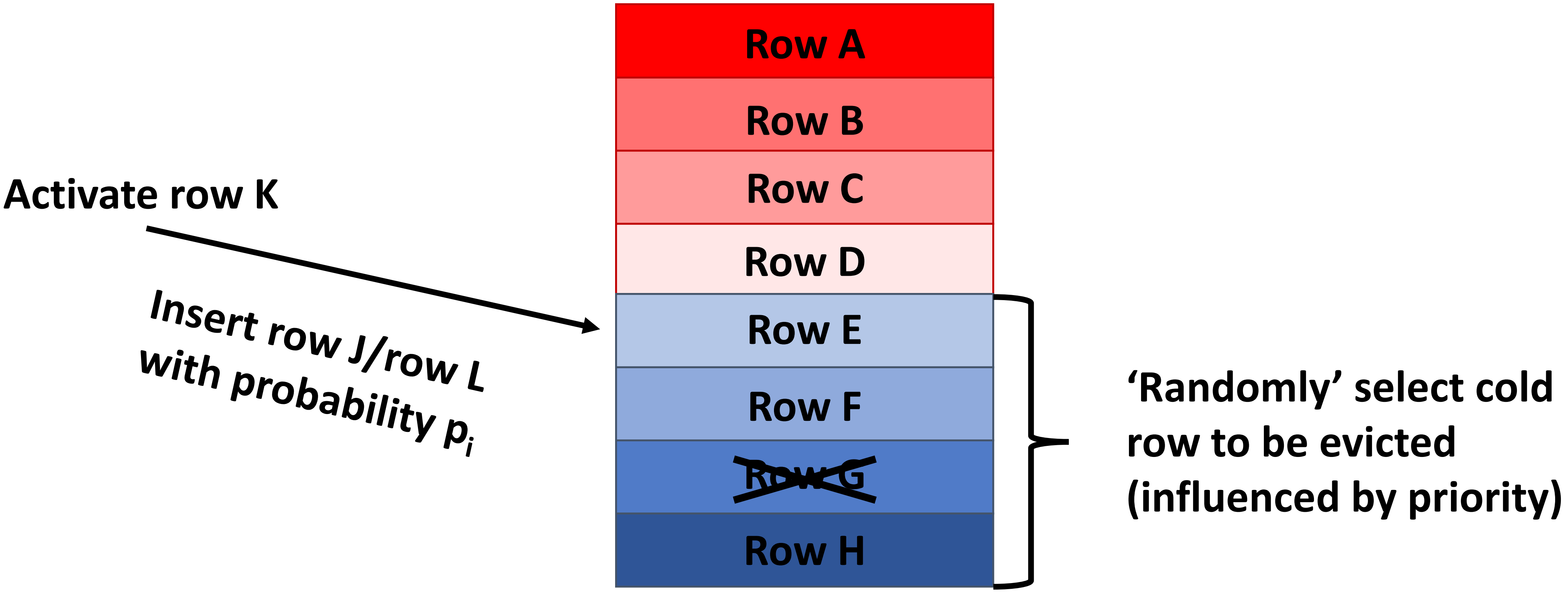
Row E

Row F

Row G

Row H

ProHIT: mechanism



Activate row K

Insert row J/row L with probability p_i

Row A

Row B

Row C

Row D

Row E

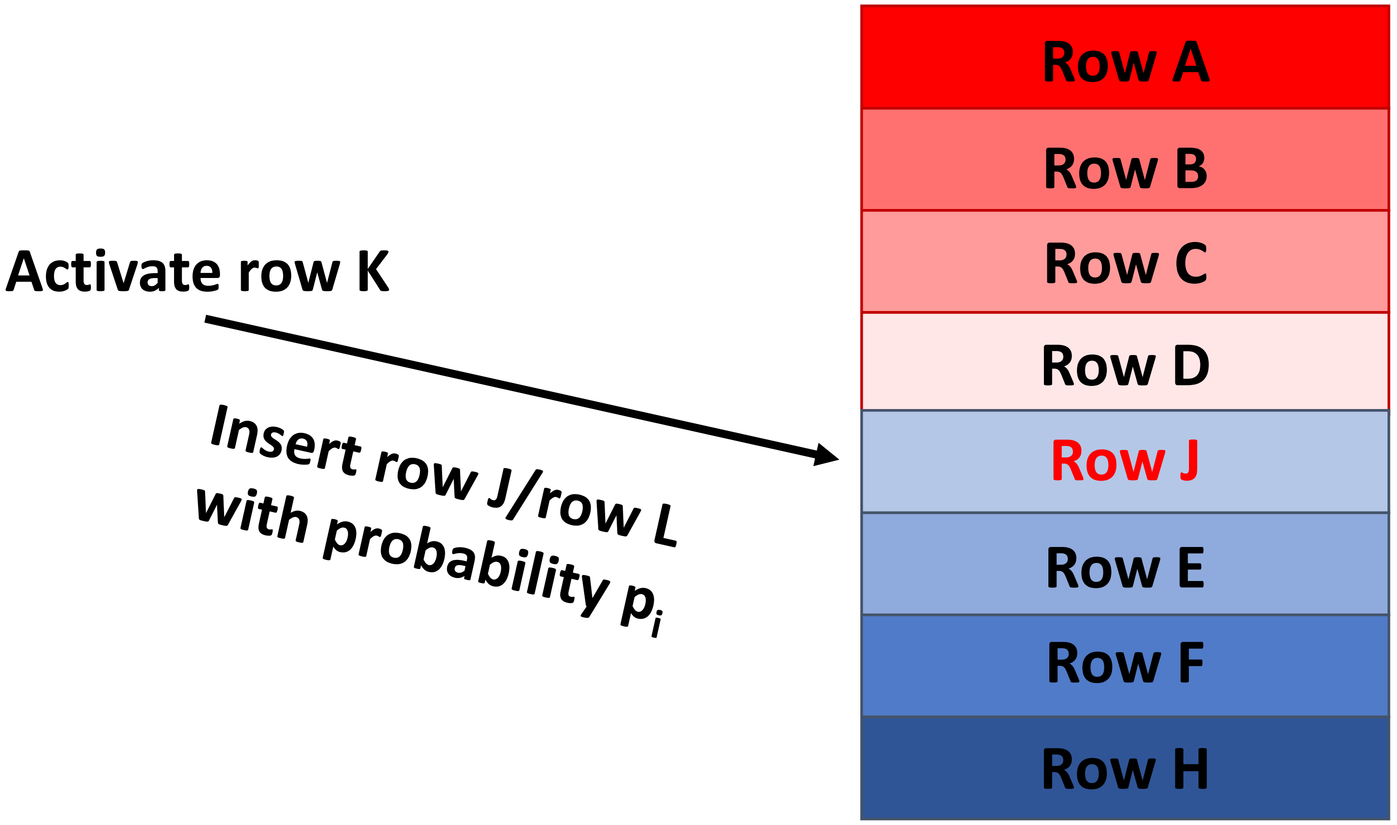
Row F

~~Row G~~

Row H

‘Randomly’ select cold row to be evicted (influenced by priority)

ProHIT: mechanism



Activate row K

Insert row J/row L with probability p_i

Row A

Row B

Row C

Row D

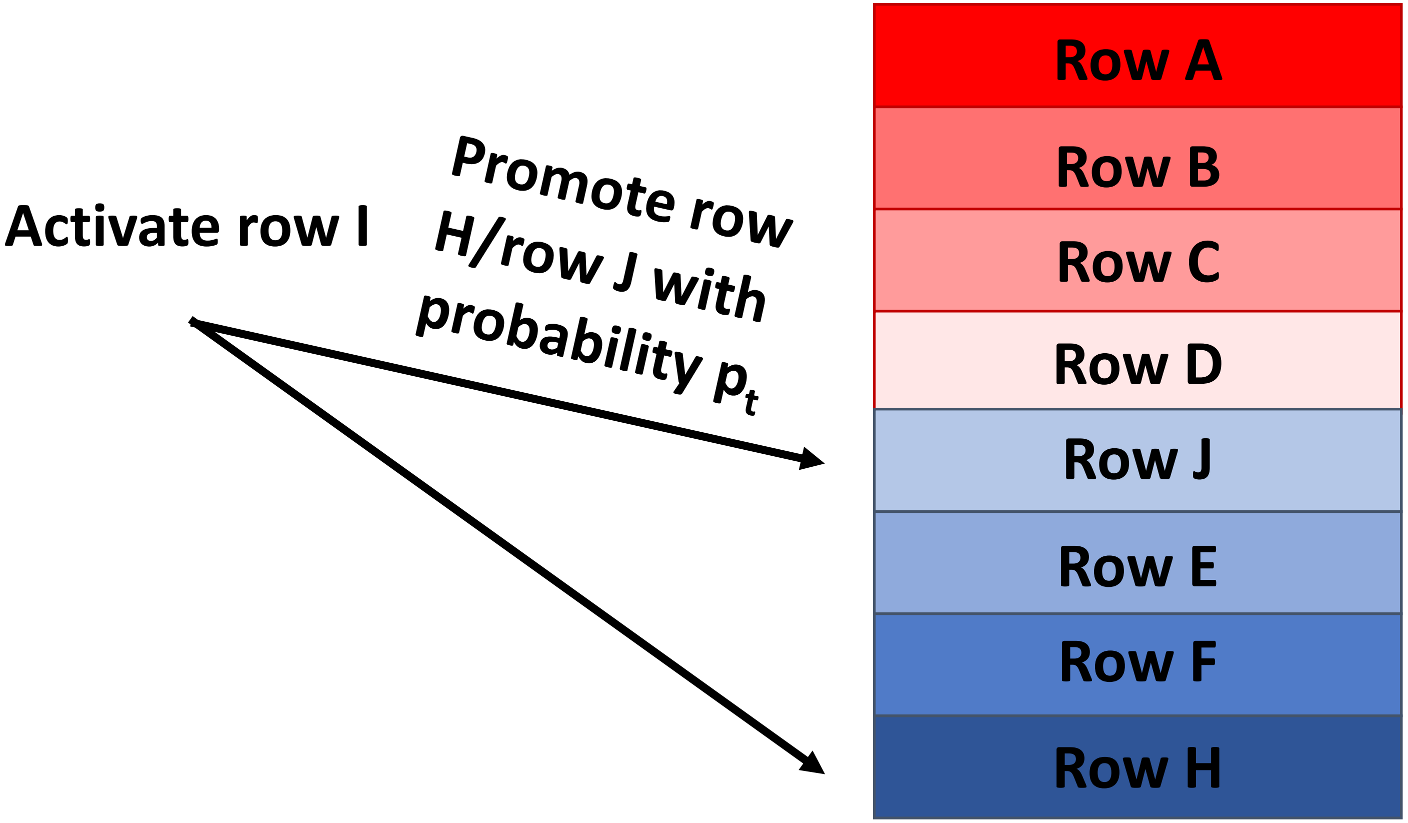
Row J

Row E

Row F

Row H

ProHIT: mechanism



Activate row I

Promote row H/row J with probability p_t

Row A

Row B

Row C

Row D

Row J

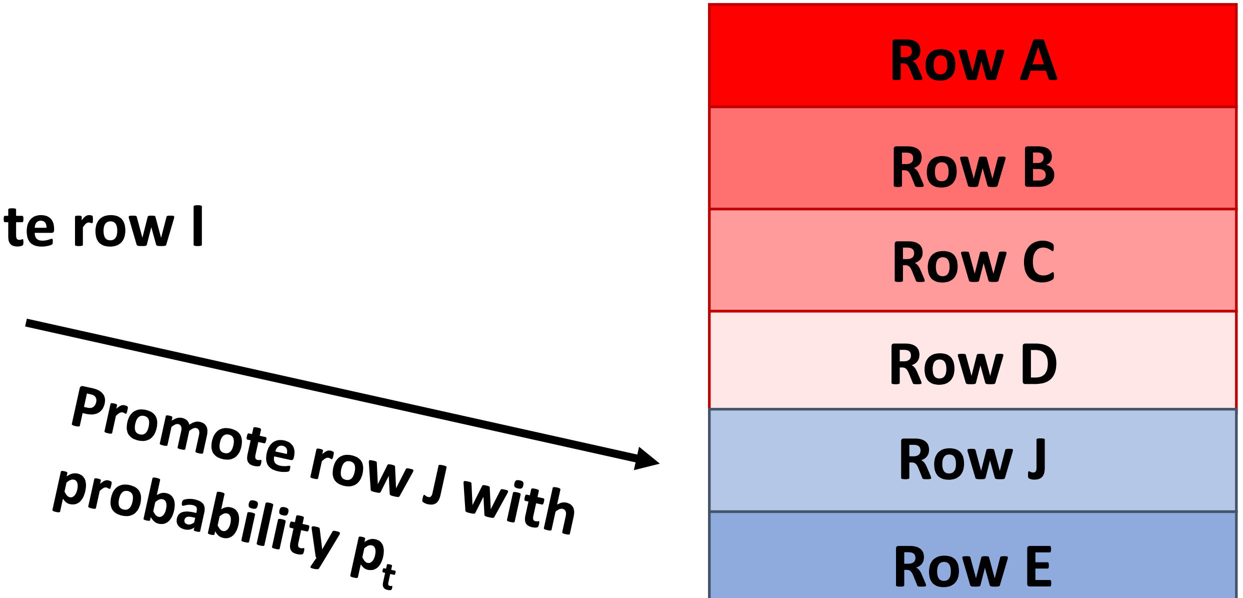
Row E

Row F

Row H

ProHIT: mechanism

Activate row I



Promote row J with probability p_t

Row A

Row B

Row C

Row D

Row J

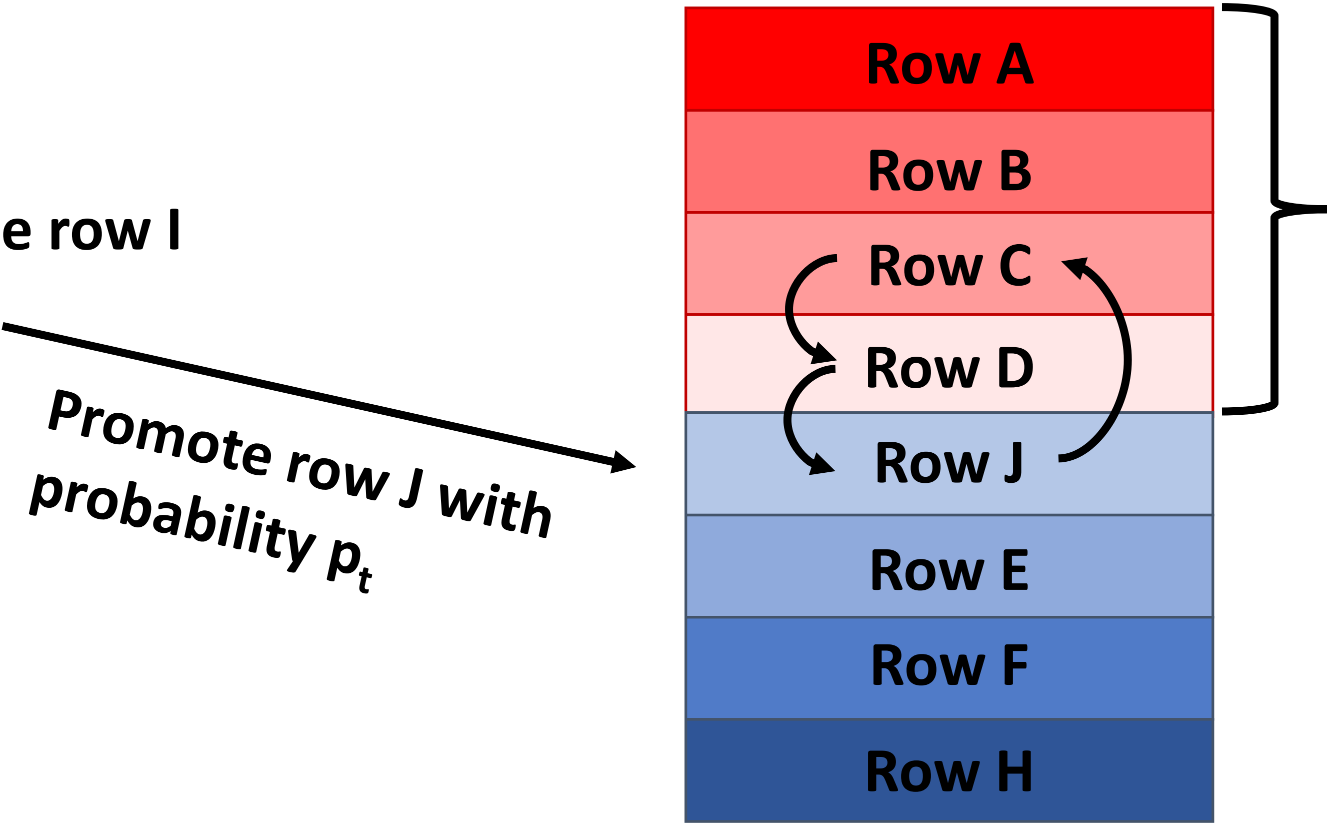
Row E

Row F

Row H

ProHIT: mechanism

Activate row I



Promote row J with probability p_t

Row A

Row B

Row C

Row D

Row J

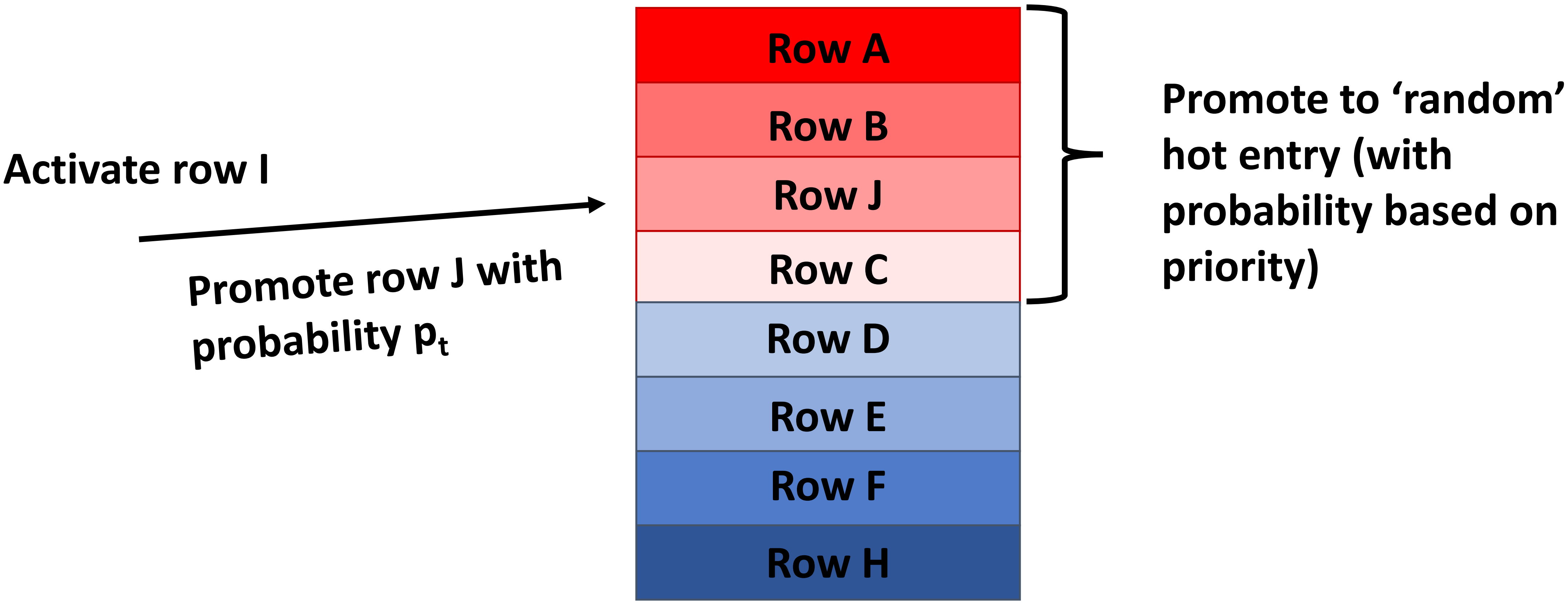
Row E

Row F

Row H

Promote to ‘random’ hot entry (with probability based on priority)

ProHIT: mechanism



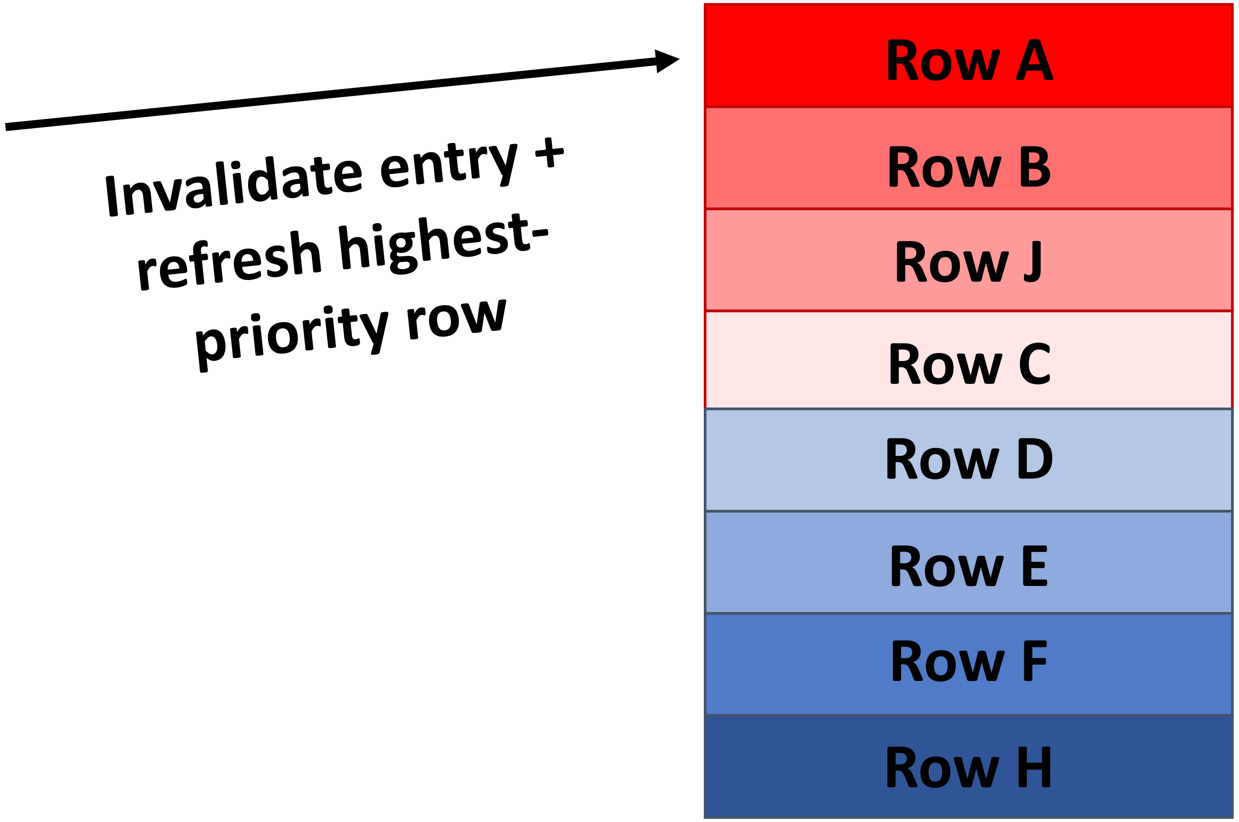
Activate row I

Promote row J with probability p_t

Promote to ‘random’ hot entry (with probability based on priority)

ProHIT: mechanism

After specific time interval



Invalidate entry + refresh highest-priority row

Row A

Row B

Row J

Row C

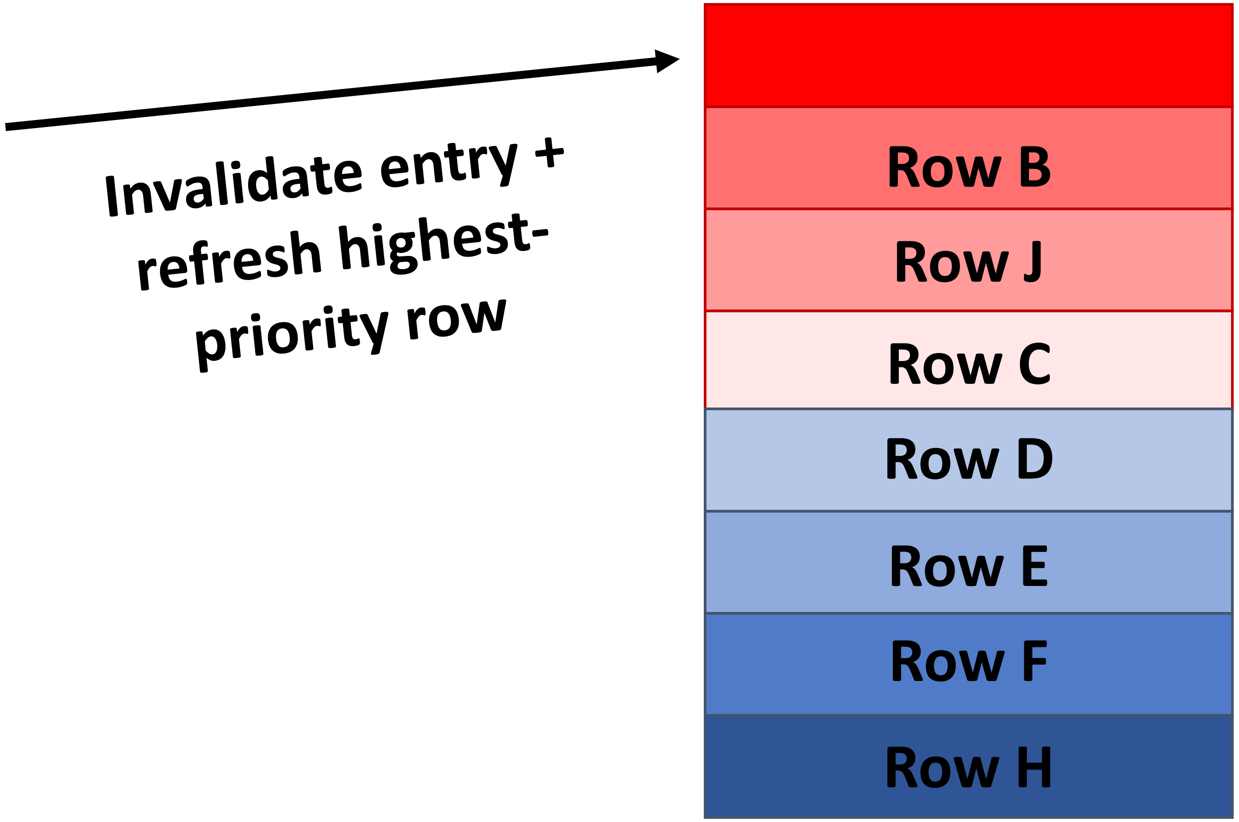
Row D

Row E

Row F

Row H

ProHIT: mechanism



Invalidate entry + refresh highest-priority row

ProHIT: weaknesses

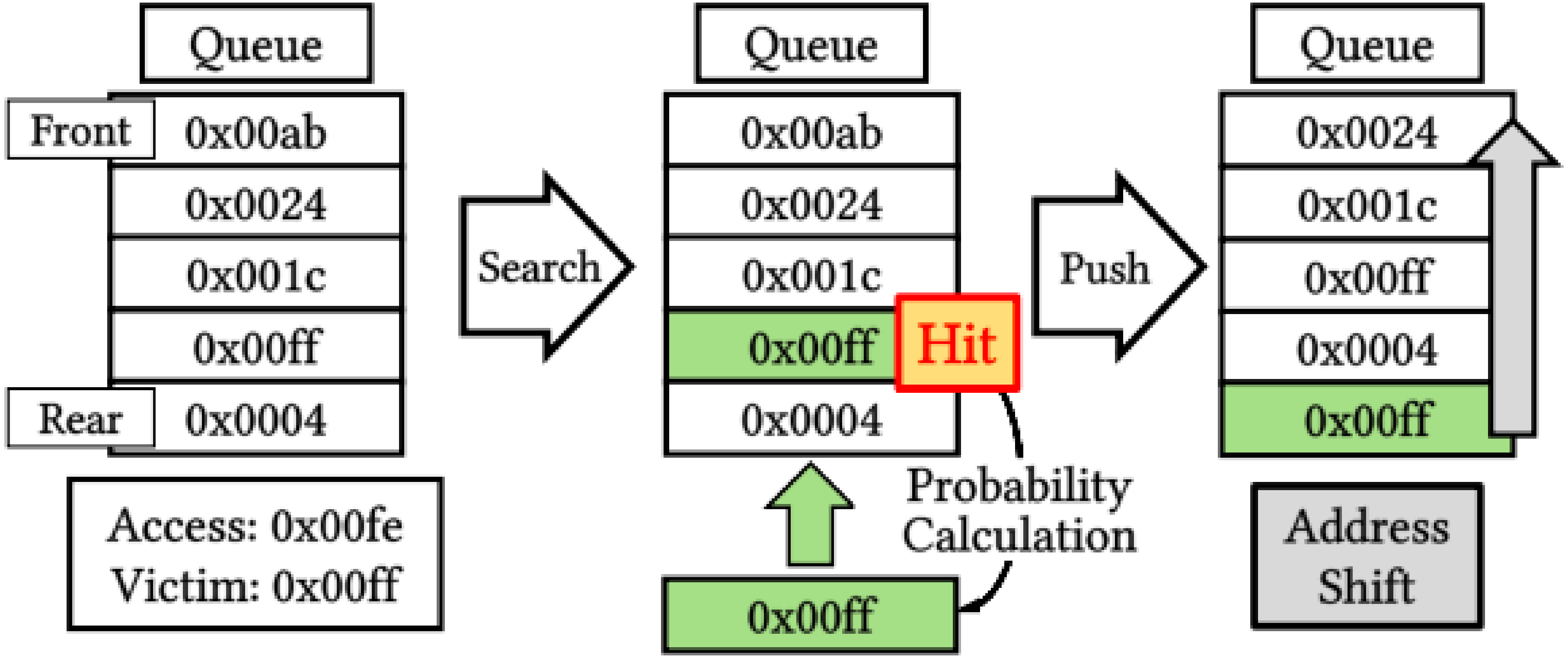
- Still cannot prevent bit-flips with 100% certainty (probabilistic!)
- But at least we have better performance!
- Knowledge on in-DRAM mapping still needed

MRLoc: definition

- Based on PARA
- **M**itigating **R**ow-hammering based on memory **L**ocality

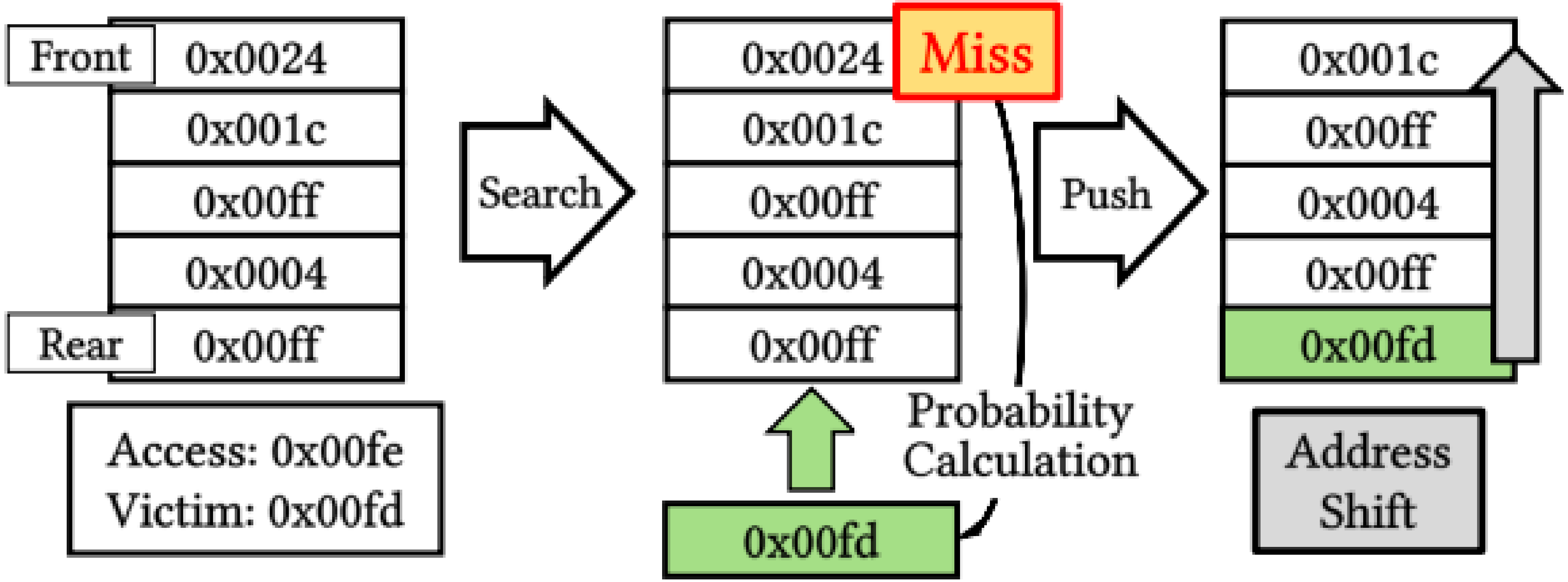
- Optimizes refresh probability based on memory locality
  - If a certain row has been accessed recently, a higher probability is assigned to its corresponding victim rows
- Victim rows are stored in queue

MRLoc: mechanism



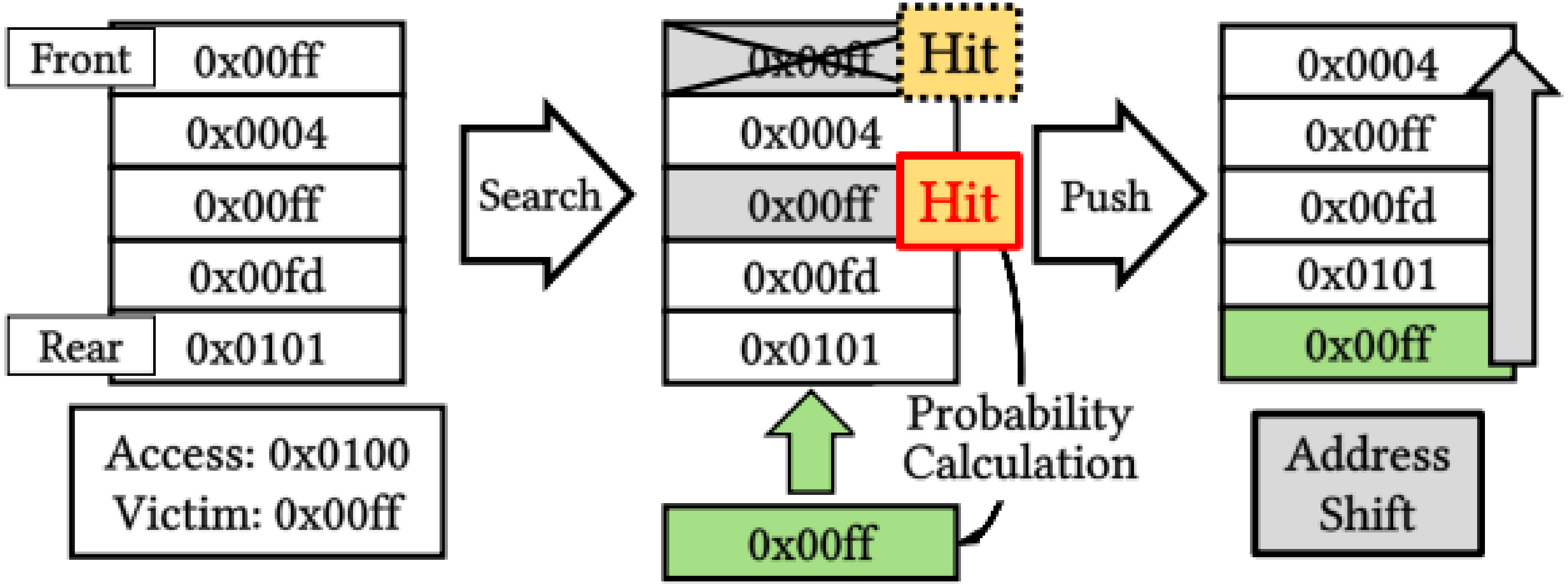
From J. M. You and J.-S. Yang, “MRLoc: Mitigating Row-Hammering Based on Memory Locality,” in DAC, 2019.

MRLoc: mechanism



From J. M. You and J.-S. Yang, “MRLoc: Mitigating Row-Hammering Based on Memory Locality,” in DAC, 2019.

MRLoc: mechanism



From J. M. You and J.-S. Yang, “MRLoc: Mitigating Row-Hammering Based on Memory Locality,” in DAC, 2019.

MRLoc: weaknesses

- Cannot prevent bit-flips with 100% certainty (probabilistic!)
- Even worse performance now …
- Knowledge on in-DRAM mapping needed

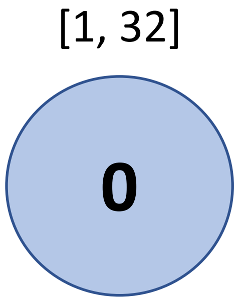
Comparison

- Compare BlockHammer with
  - (Baseline system: no RH mitigation)
  - 3 probabilistic mitigation mechanisms
    - PARA
    - ProHIT
    - MRLoc
  - **3 deterministic mitigation mechanisms**
    - **CBT**
    - **TWiCe**
    - **Graphene**

CBT: definition

- = Counter-Based Tree
- Tree of counters that count row activations in disjoint memory regions
  - Whenever parent node reaches certain threshold, memory region is halved (one half for each child)
  - Predefined threshold for each level
  - Leaf node reaches threshold: counter reset + refresh of respective memory region

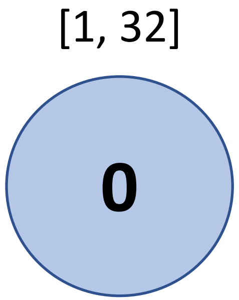
CBT: mechanism



Threshold = 2

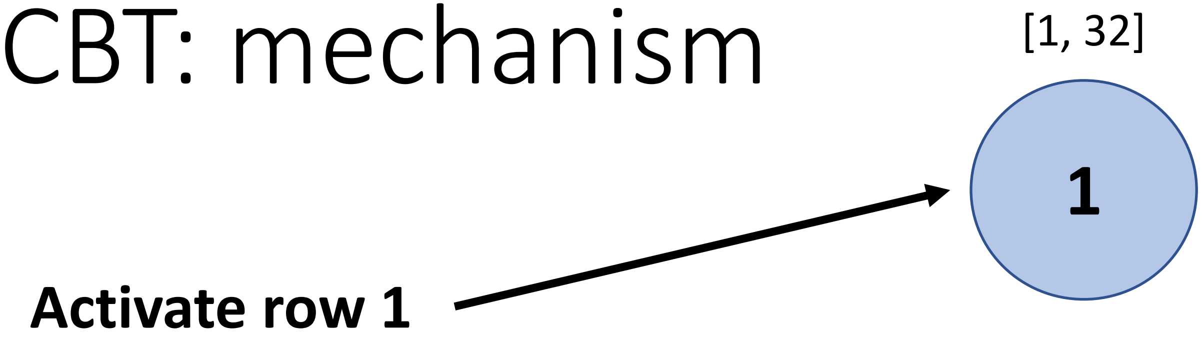
CBT: mechanism

Activate row 1



Threshold = 2

CBT: mechanism



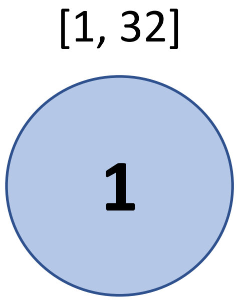
[1, 32]

1

Activate row 1

Threshold = 2

CBT: mechanism

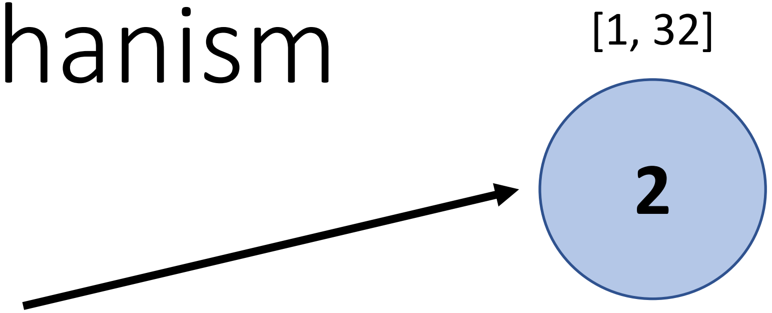


Threshold = 2

Activate row 4

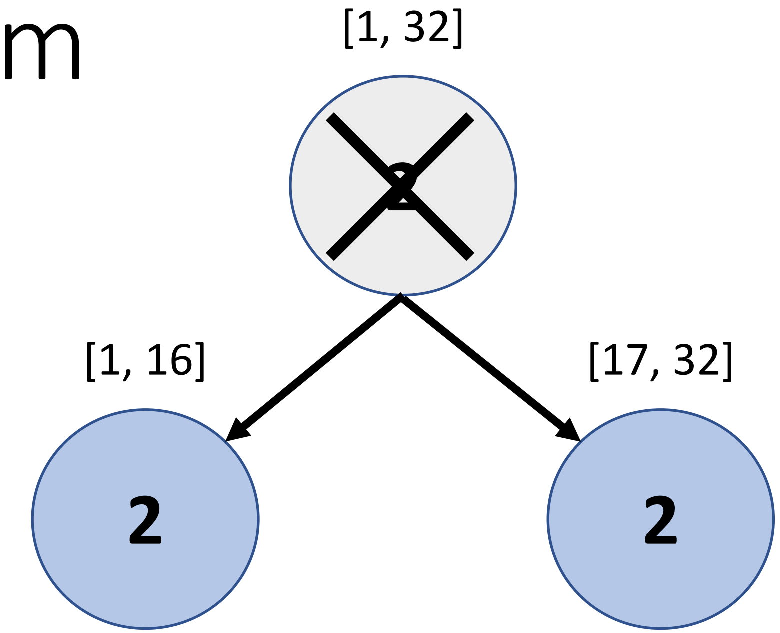
CBT: mechanism

Activate row 4



Threshold = 2

CBT: mechanism

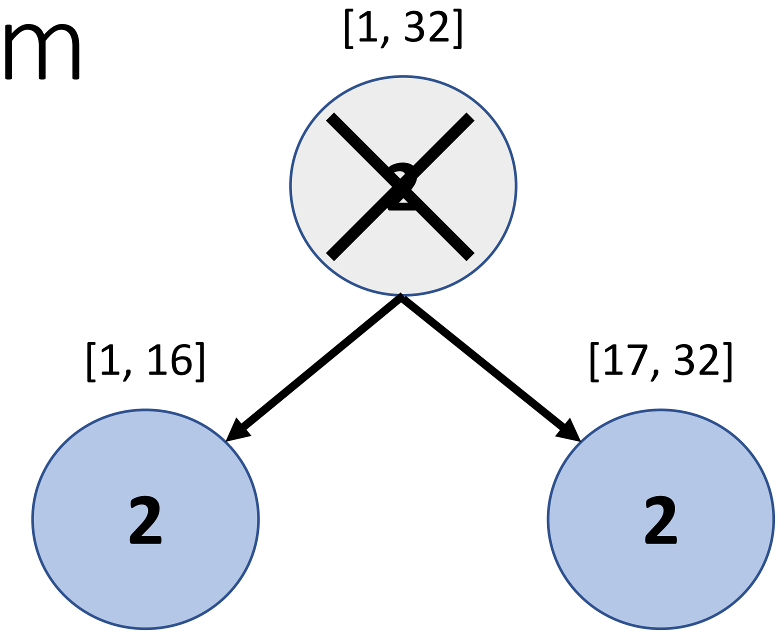


Threshold = 2

Threshold = 5

CBT: mechanism

Activate row 4

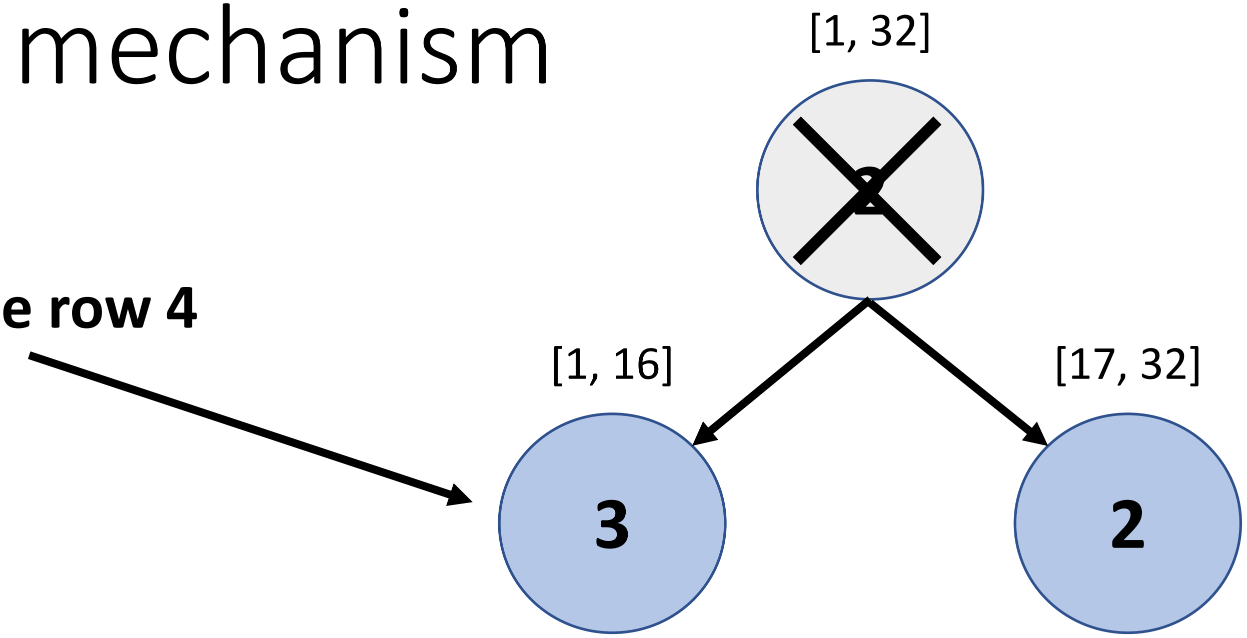


Threshold = 2

Threshold = 5

CBT: mechanism

Activate row 4

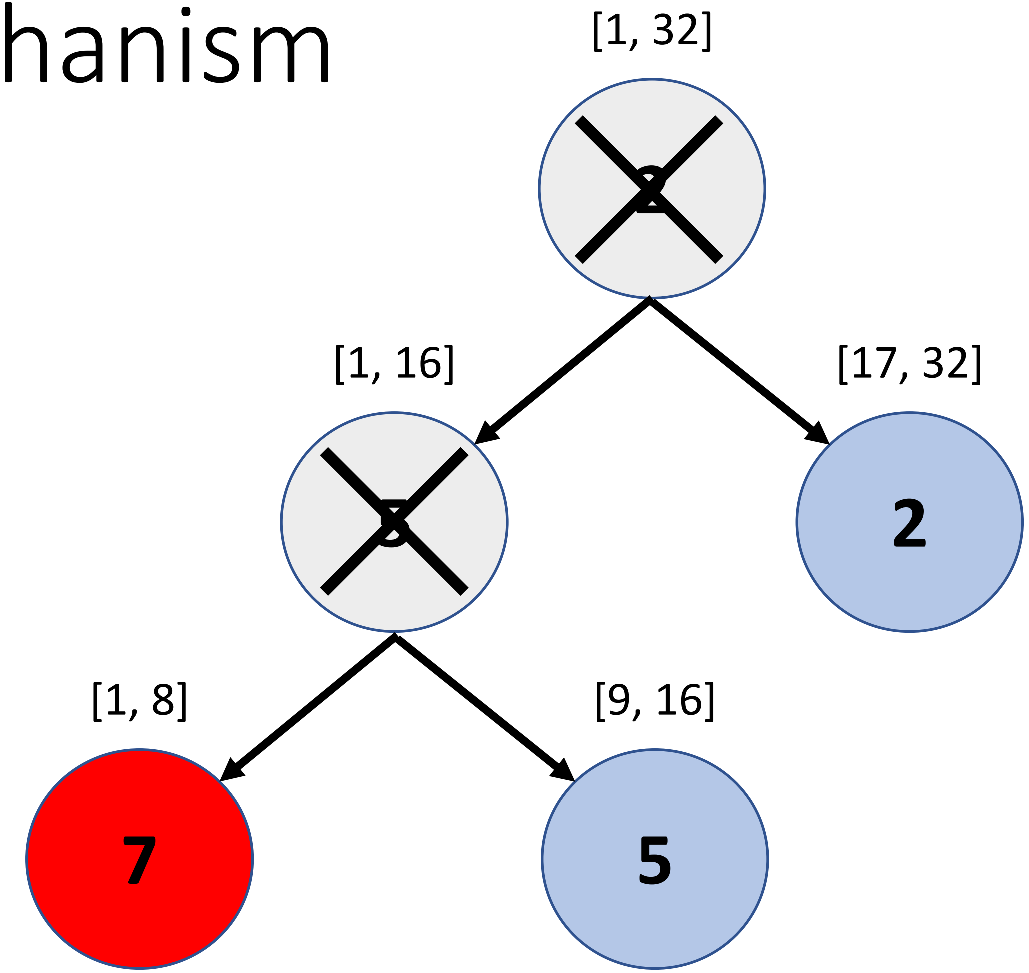


Threshold = 2

Threshold = 5

CBT: mechanism

And so on …

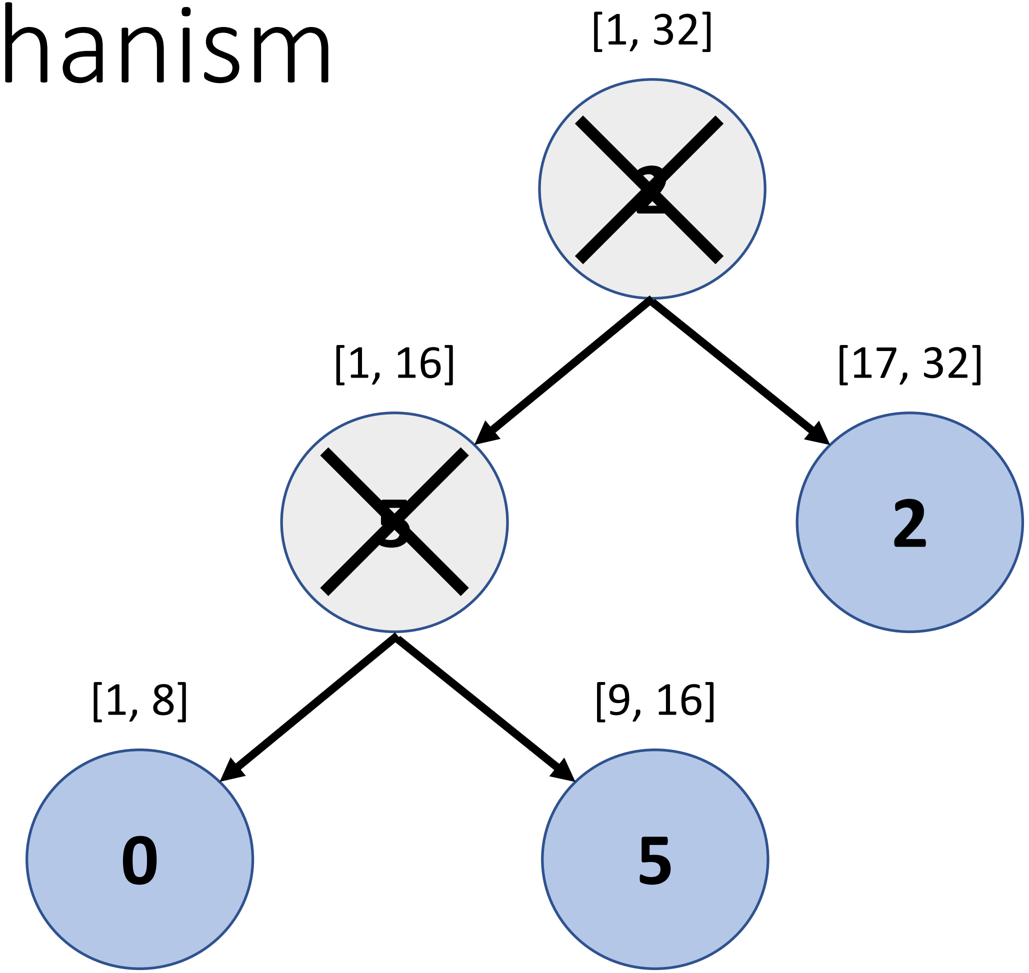


Threshold = 2

Threshold = 5

Threshold = 7

CBT: mechanism



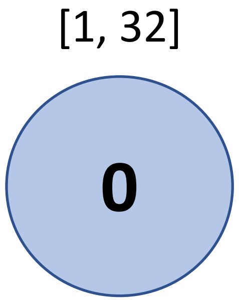
Threshold = 2

Threshold = 5

Reset & Refresh!!

Threshold = 7

CBT: mechanism



Threshold = 2

At end of refresh period (e.g. 64 ms)

CBT: weaknesses

- Area vs. performance trade-off
  - More levels means smaller memory region size and thus more correct refreshes (better performance), but at higher area cost
- Assumes rows are contiguous but might not be the case → DRAM remaps addresses internally

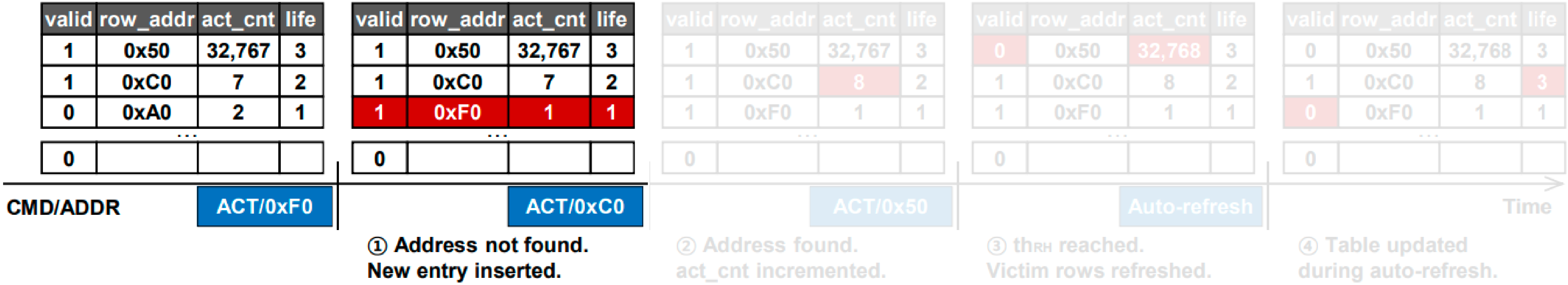
TWiCe: definition

- = Time Window Counter based row refresh

- Maximum number of DRAM ACTs over t_{REFW} is bounded
- Counter table: Valid bit | Row address | Activation count | Life
  - Counter table + counter logic
  - Activation count: records number of activations to the target row address
  - Valid bit: is entry valid?
  - Life: # consecutive pruning intervals for which entry stays valid in the table

TWiCe: mechanism

- Row activation
  - Not in table → allocate entry



valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
0	0xA0	2	1
…			
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
1	0xF0	1	1
…			
0			

① Address not found. New entry inserted.

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	8	2
1	0xF0	1	1
…			
0			

② Address found. act_cnt incremented.

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	2
1	0xF0	1	1
…			
0			

③ th_{RH} reached. Victim rows refreshed.

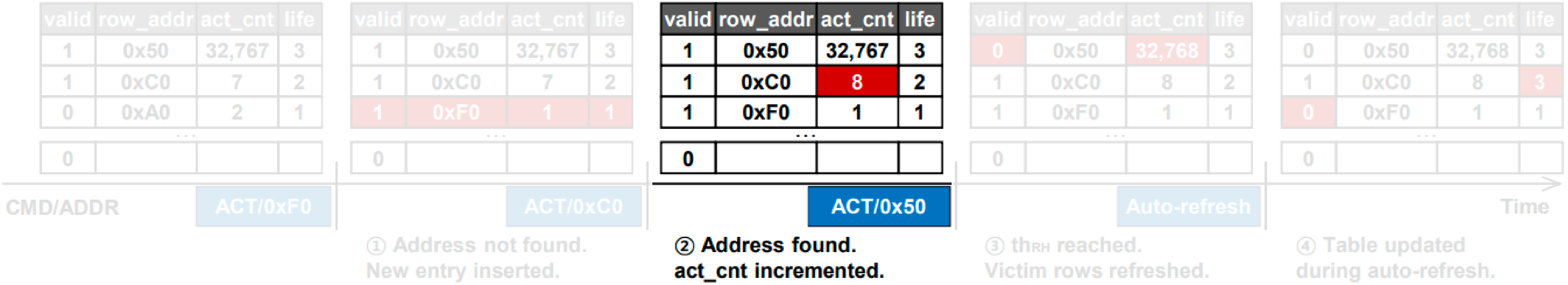
valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	3
0	0xF0	1	1
…			
0			

④ Table updated during auto-refresh.

From E. Lee et al., “TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters,” in ISCA, 2019.

TWiCe: mechanism

- Row activation
  - Not in table → allocate entry
  - In table → increment activation count



valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
0	0xA0	2	1
	...		
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
1	0xF0	1	1
	...		
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	8	2
1	0xF0	1	1
	...		
0			

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	2
1	0xF0	1	1
	...		
0			

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	3
0	0xF0	1	1
	...		
0			

① Address not found.
New entry inserted.

② Address found.
act_cnt incremented.

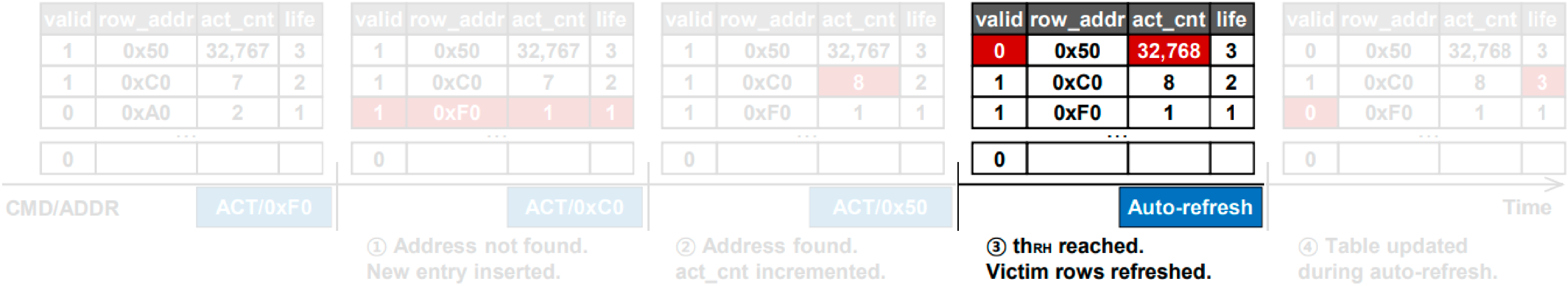
③ th_{RH} reached.
Victim rows refreshed.

④ Table updated
during auto-refresh.

From E. Lee et al., “TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters,” in ISCA, 2019.

TWiCe: mechanism

- Activation count reaches threshold → refresh victim rows & set valid bit to 0



valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
0	0xA0	2	1
…			
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
1	0xF0	1	1
…			
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	8	2
1	0xF0	1	1
…			
0			

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	2
1	0xF0	1	1
…			
0			

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	3
0	0xF0	1	1
…			
0			

① Address not found.
New entry inserted.

② Address found.
act_cnt incremented.

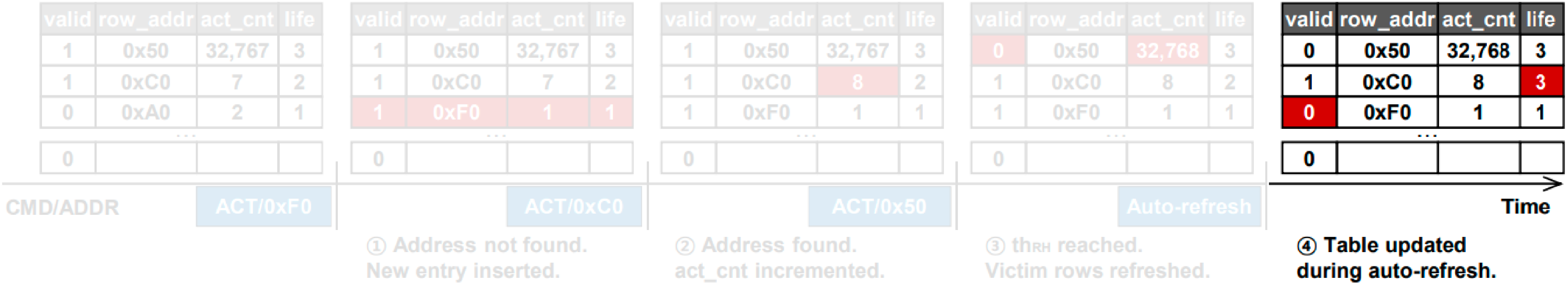
③ th$_{RH}$ reached.
Victim rows refreshed.

④ Table updated
during auto-refresh.

From E. Lee et al., “TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters,” in ISCA, 2019.

TWiCe: mechanism

- After each pruning interval
  - All entries with activation count < th_{PI} x life → removed (NOT refreshed)
  - Activation count ≥ th_{PI} x life → increment life



valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
0	0xA0	2	1
...			
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	7	2
1	0xF0	1	1
...			
0			

valid	row_addr	act_cnt	life
1	0x50	32,767	3
1	0xC0	8	2
1	0xF0	1	1
...			
0			

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	2
1	0xF0	1	1
...			
0			

valid	row_addr	act_cnt	life
0	0x50	32,768	3
1	0xC0	8	3
0	0xF0	1	1
...			
0			

① Address not found. New entry inserted.

② Address found. act_cnt incremented.

③ th_{RH} reached. Victim rows refreshed.

④ Table updated during auto-refresh.

From E. Lee et al., "TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters," in ISCA, 2019.

TWiCe: weaknesses

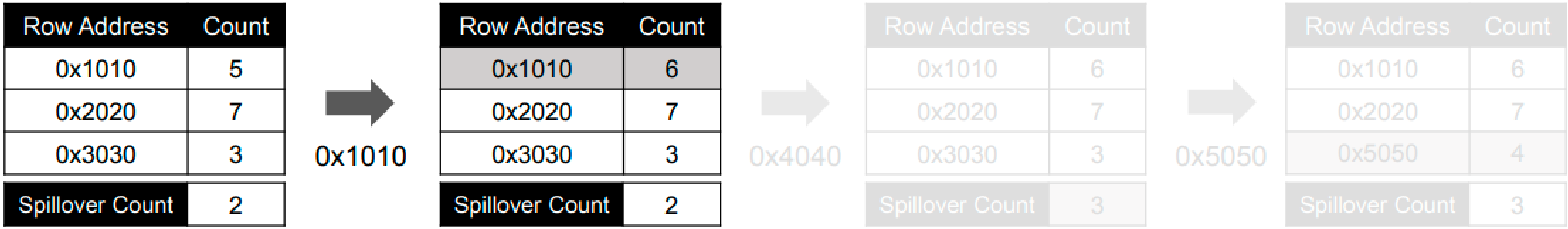
- Relatively large area overhead as RH gets worse! (in comparison to BH and Graphene)
- Needs to identify victim rows → requires knowledge of DRAM internals!

Graphene: definition

- Misra-Gries algorithm
  - Solves frequent elements problem
  - Find all elements in a (finite!) stream that occur more than a given fraction of the time
  - Here: elements = memory requests

Graphene: mechanism

- Activate row
    - Row in table → increase count



Row Address	Count
0x1010	5
0x2020	7
0x3030	3
Spillover Count	2

→ 0x1010

Row Address	Count
0x1010	6
0x2020	7
0x3030	3
Spillover Count	2

→ 0x4040

Row Address	Count
0x1010	6
0x2020	7
0x3030	3
Spillover Count	3

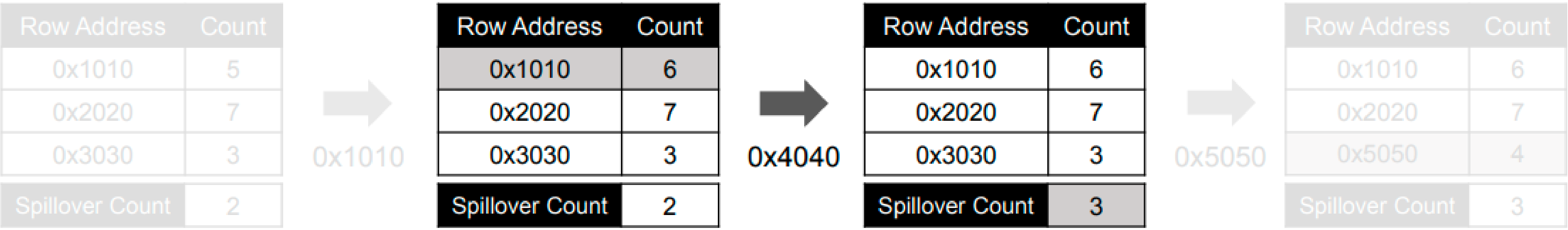
→ 0x5050

Row Address	Count
0x1010	6
0x2020	7
0x5050	4
Spillover Count	3

From Y. Park et al., “Graphene: Strong yet Lightweight Row Hammer Protection,” in MICRO, 2020

Graphene: mechanism

- Activate row
  - Row not in table AND spillover count < count of all entries → increment spillover count



Row Address	Count
0x1010	5
0x2020	7
0x3030	3
Spillover Count	2

Row Address	Count
0x1010	6
0x2020	7
0x3030	3
Spillover Count	2

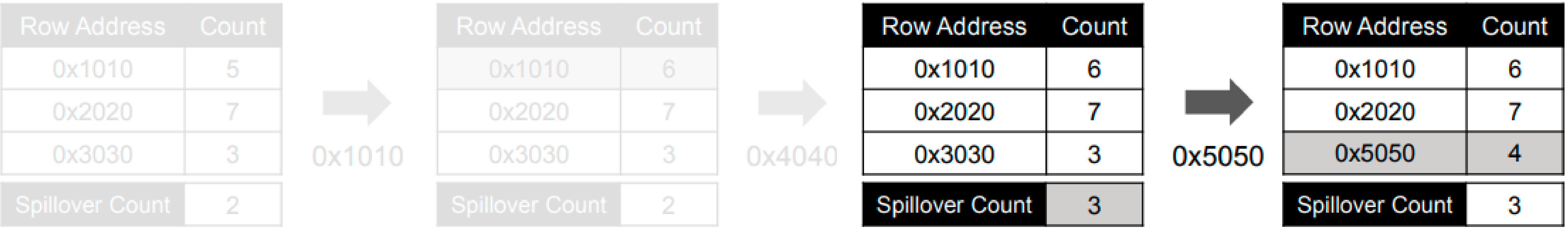
Row Address	Count
0x1010	6
0x2020	7
0x3030	3
Spillover Count	3

Row Address	Count
0x1010	6
0x2020	7
0x5050	4
Spillover Count	3

From Y. Park et al., "Graphene: Strong yet Lightweight Row Hammer Protection," in MICRO, 2020

Graphene: mechanism

- Activate row
  - Row not in table AND spillover count >= count of some entry X → replace entry X with new row + increment count of that row



Row Address	Count
0x1010	5
0x2020	7
0x3030	3
Spillover Count	2

Row Address	Count
0x1010	6
0x2020	7
0x3030	3
Spillover Count	2

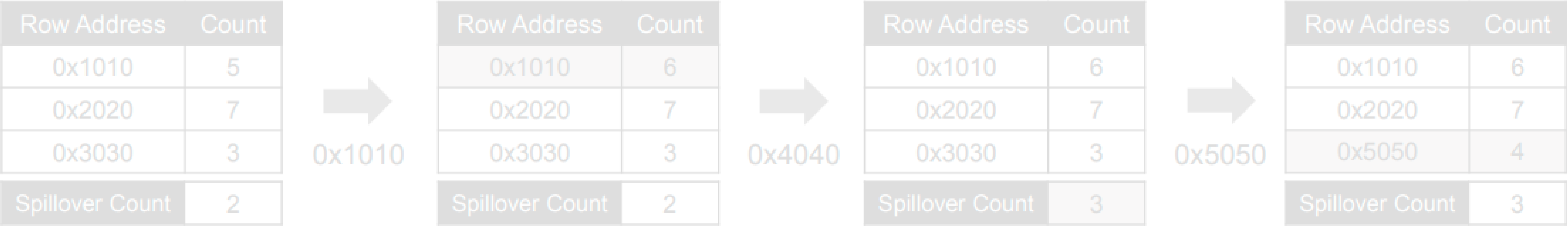
Row Address	Count
0x1010	6
0x2020	7
0x3030	3
Spillover Count	3

Row Address	Count
0x1010	6
0x2020	7
0x5050	4
Spillover Count	3

From Y. Park et al., "Graphene: Strong yet Lightweight Row Hammer Protection," in MICRO, 2020

Graphene: mechanism

- Count == (multiple of) threshold → refresh victim rows



Row Address	Count
0x1010	5
0x2020	7
0x3030	3
Spillover Count	2

0x1010

Row Address	Count
0x1010	6
0x2020	7
0x3030	3
Spillover Count	2

0x4040

Row Address	Count
0x1010	6
0x2020	7
0x3030	3
Spillover Count	3

0x5050

Row Address	Count
0x1010	6
0x2020	7
0x5050	4
Spillover Count	3

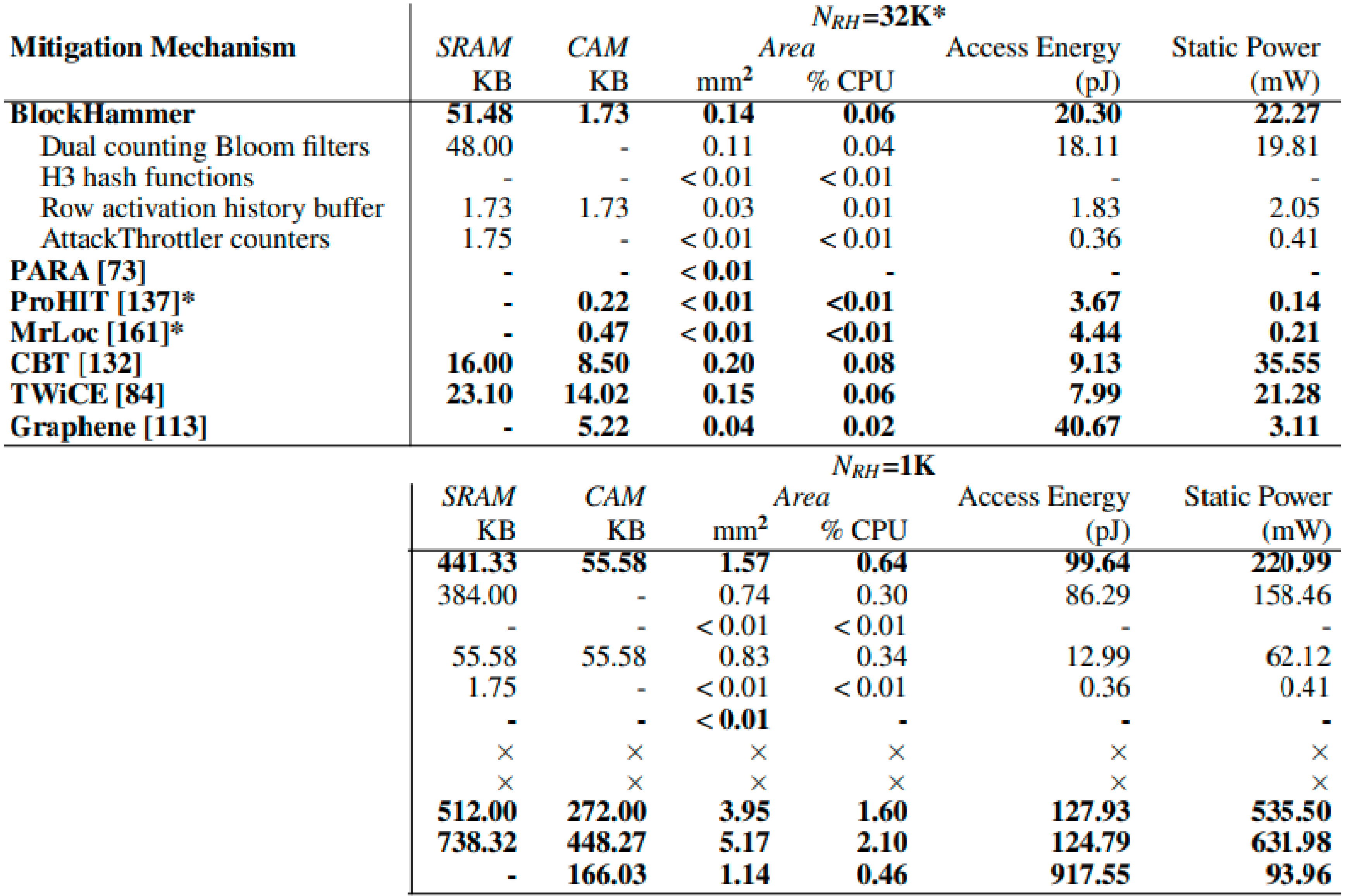
From Y. Park et al., “Graphene: Strong yet Lightweight Row Hammer Protection,” in MICRO, 2020

Graphene: weaknesses

- Needs to identify victim rows → requires knowledge of DRAM internals

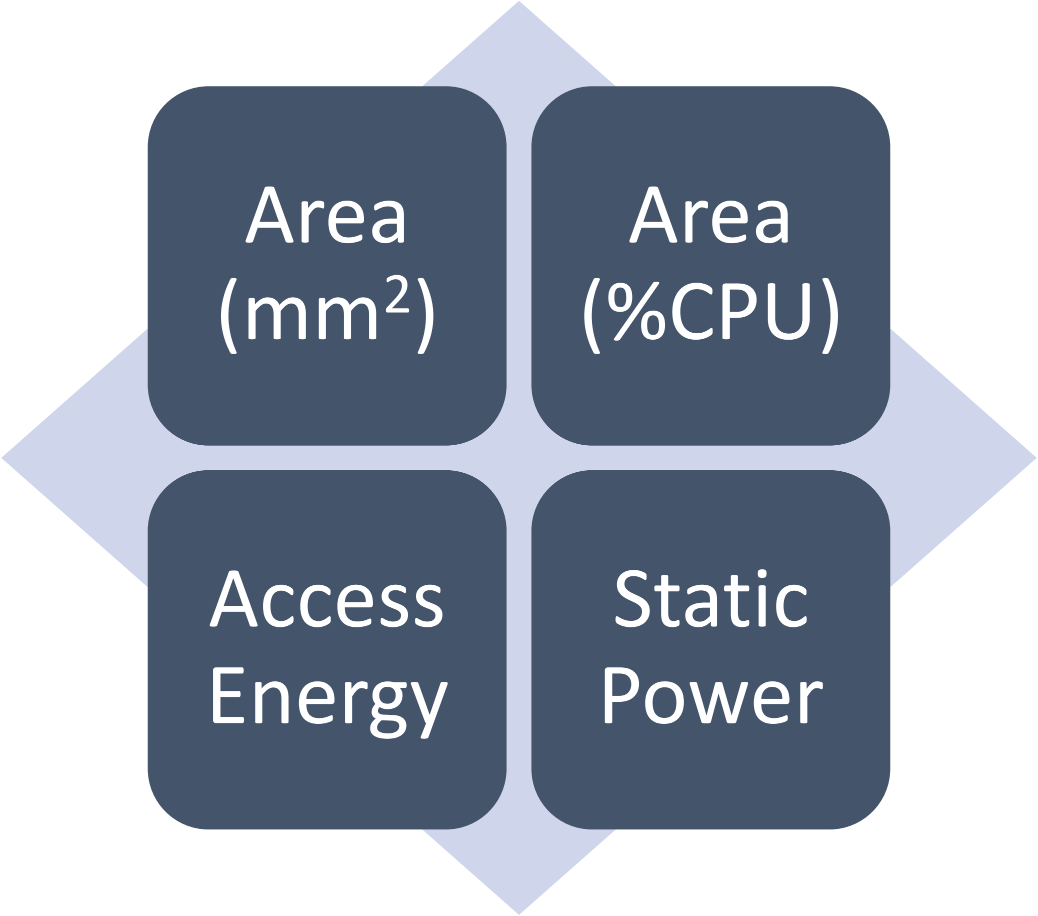
Currently one of the best solutions (has good performance and low area overhead)

1. Hardware complexity analysis



	N_{RH}=32K*						N_{RH}=1K					
Mitigation Mechanism	*SRAM*	*CAM*	*Area*		Access Energy	Static Power	*SRAM*	*CAM*	*Area*		Access Energy	Static Power
	KB	KB	mm^2	% CPU	(pJ)	(mW)	KB	KB	mm^2	% CPU	(pJ)	(mW)
BlockHammer	**51.48**	**1.73**	**0.14**	**0.06**	**20.30**	**22.27**	**441.33**	**55.58**	**1.57**	**0.64**	**99.64**	**220.99**
Dual counting Bloom filters	48.00	-	0.11	0.04	18.11	19.81	384.00	-	0.74	0.30	86.29	158.46
H3 hash functions	-	-	< 0.01	< 0.01	-	-	-	-	< 0.01	< 0.01	-	-
Row activation history buffer	1.73	1.73	0.03	0.01	1.83	2.05	55.58	55.58	0.83	0.34	12.99	62.12
AttackThrottler counters	1.75	-	< 0.01	< 0.01	0.36	0.41	1.75	-	< 0.01	< 0.01	0.36	0.41
PARA [73]	**-**	**-**	**< 0.01**	**-**	**-**	**-**	**-**	**-**	**< 0.01**	**-**	**-**	**-**
ProHIT [137]*	**-**	**0.22**	**< 0.01**	**<0.01**	**3.67**	**0.14**	×	×	×	×	×	×
MrLoc [161]*	**-**	**0.47**	**< 0.01**	**<0.01**	**4.44**	**0.21**	×	×	×	×	×	×
CBT [132]	**16.00**	**8.50**	**0.20**	**0.08**	**9.13**	**35.55**	**512.00**	**272.00**	**3.95**	**1.60**	**127.93**	**535.50**
TWiCE [84]	**23.10**	**14.02**	**0.15**	**0.06**	**7.99**	**21.28**	**738.32**	**448.27**	**5.17**	**2.10**	**124.79**	**631.98**
Graphene [113]	**-**	**5.22**	**0.04**	**0.02**	**40.67**	**3.11**	**-**	**166.03**	**1.14**	**0.46**	**917.55**	**93.96**

1. Hardware complexity analysis



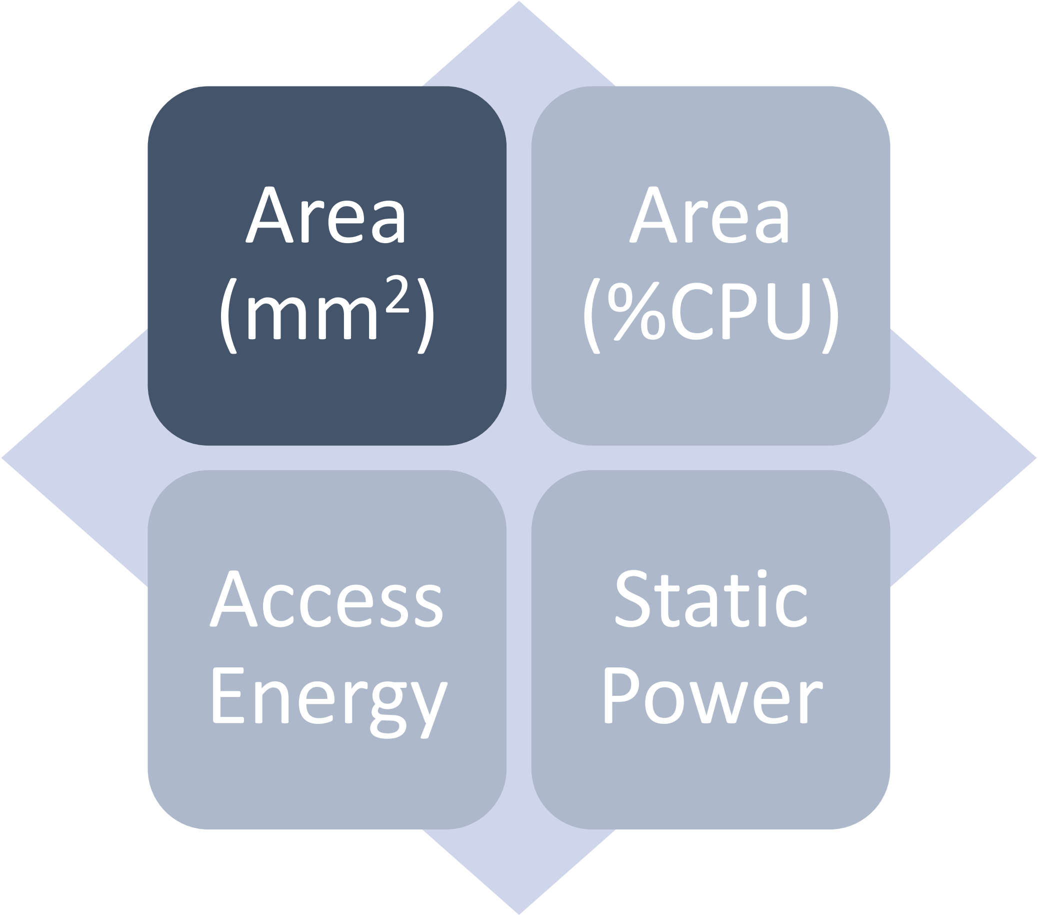
Area (mm^2)

Area (%CPU)

Access Energy

Static Power

1. Hardware complexity analysis



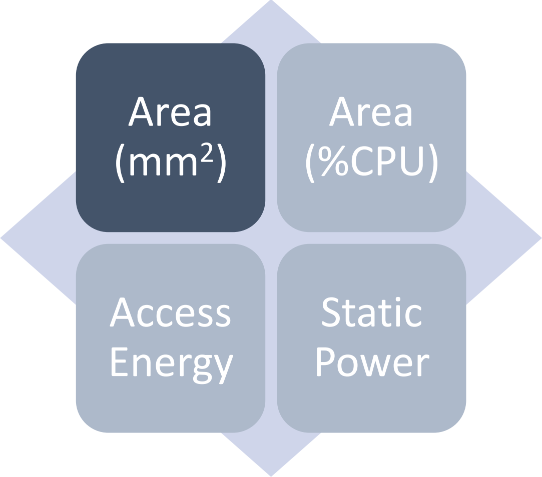
Area (mm^2)

Area (%CPU)

Access Energy

Static Power

1. Hardware complexity analysis



Area (mm^2)

Area (%CPU)

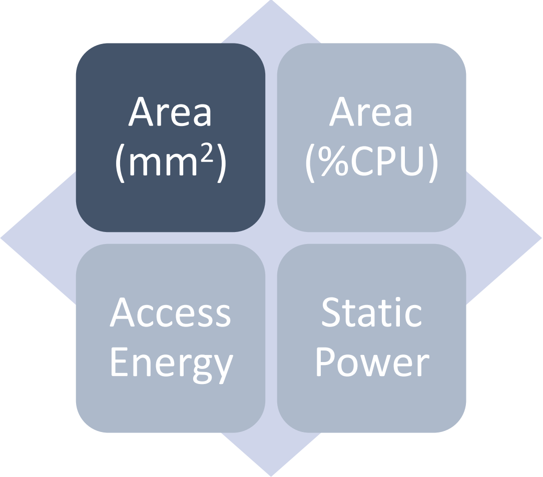
Access Energy

Static Power

RowHammer threshold 32K

- PARA, PRoHIT, MRLoc → extremely area-efficient (because probabilistic)
- Graphene << TWiCe, BlockHammer < CBT → still relatively area-efficient

1. Hardware complexity analysis



Area (mm^2)

Area (%CPU)

Access Energy

Static Power

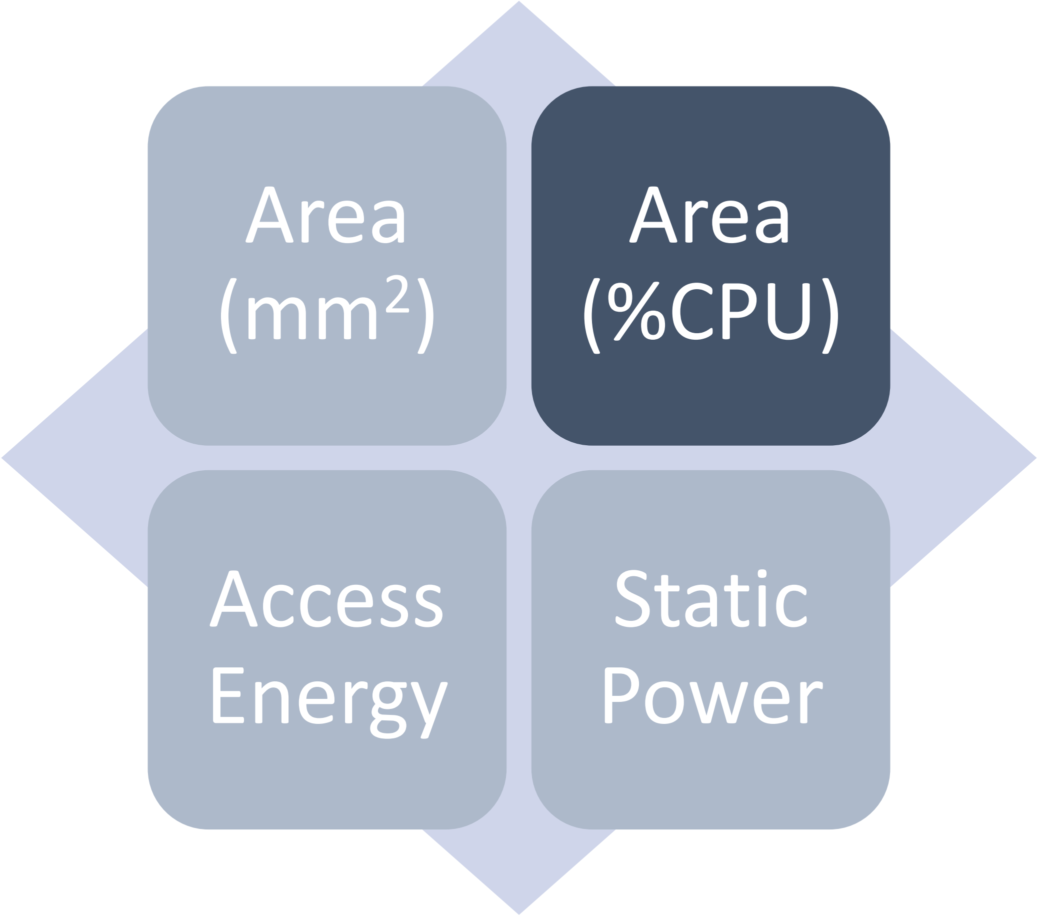
RowHammer threshold 32K

- PARA, PRoHIT, MRLoc → extremely area-efficient (because probabilistic)
- Graphene << TWiCe, BlockHammer < CBT → still relatively area-efficient

RowHammer threshold 1K

- Graphene x28.5, TWiCE x34.5, CBT x19.7 ↔ BlockHammer x11.2
- **New order: Graphene < BlockHammer << TWiCE << CBT**
  - **BlockHammer is catching up!**

1. Hardware complexity analysis



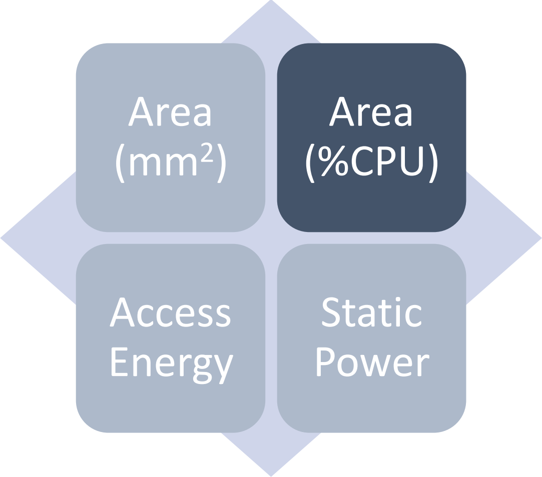
Area (mm^2)

Area (%CPU)

Access Energy

Static Power

1. Hardware complexity analysis



Area (mm^2)

Area (%CPU)

Access Energy

Static Power

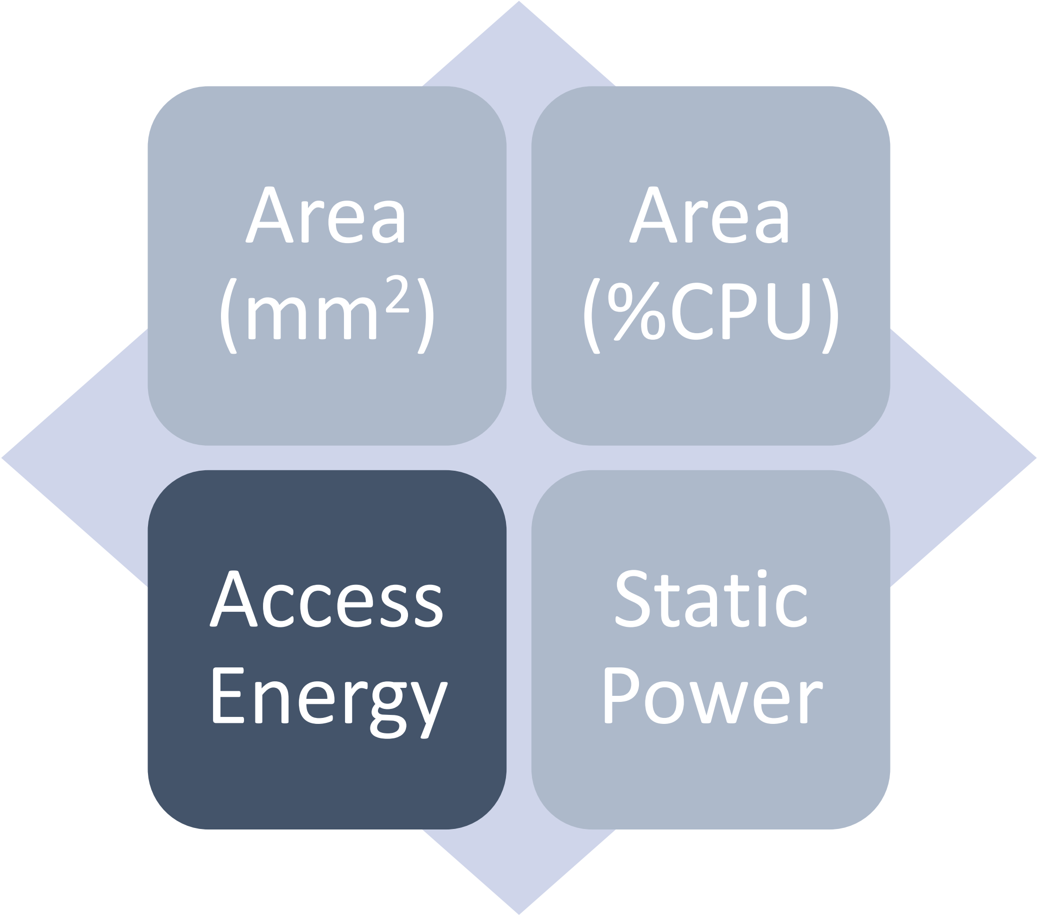
RowHammer threshold 32K

- PARA, PRoHIT, MRLoc → extremely area-efficient (because probabilistic)
- Graphene << TWiCe, BlockHammer < CBT → still relatively area-efficient

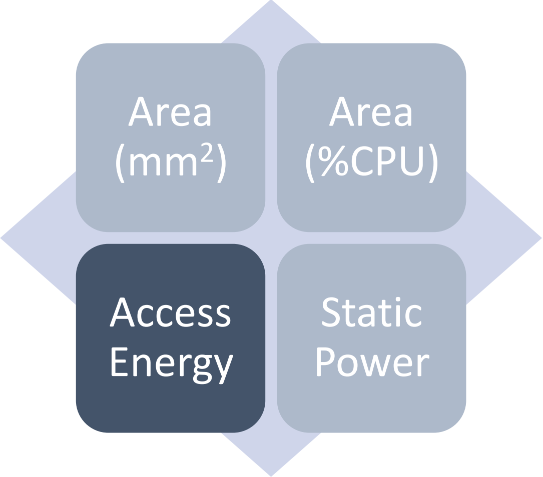
RowHammer threshold 1K

- Graphene x23, TWiCE x35, CBT x20 ↔ BlockHammer x10.7
- **New order: Graphene < BlockHammer << TWiCE << CBT**
  - **BlockHammer is catching up!**

1. Hardware complexity analysis



1. Hardware complexity analysis



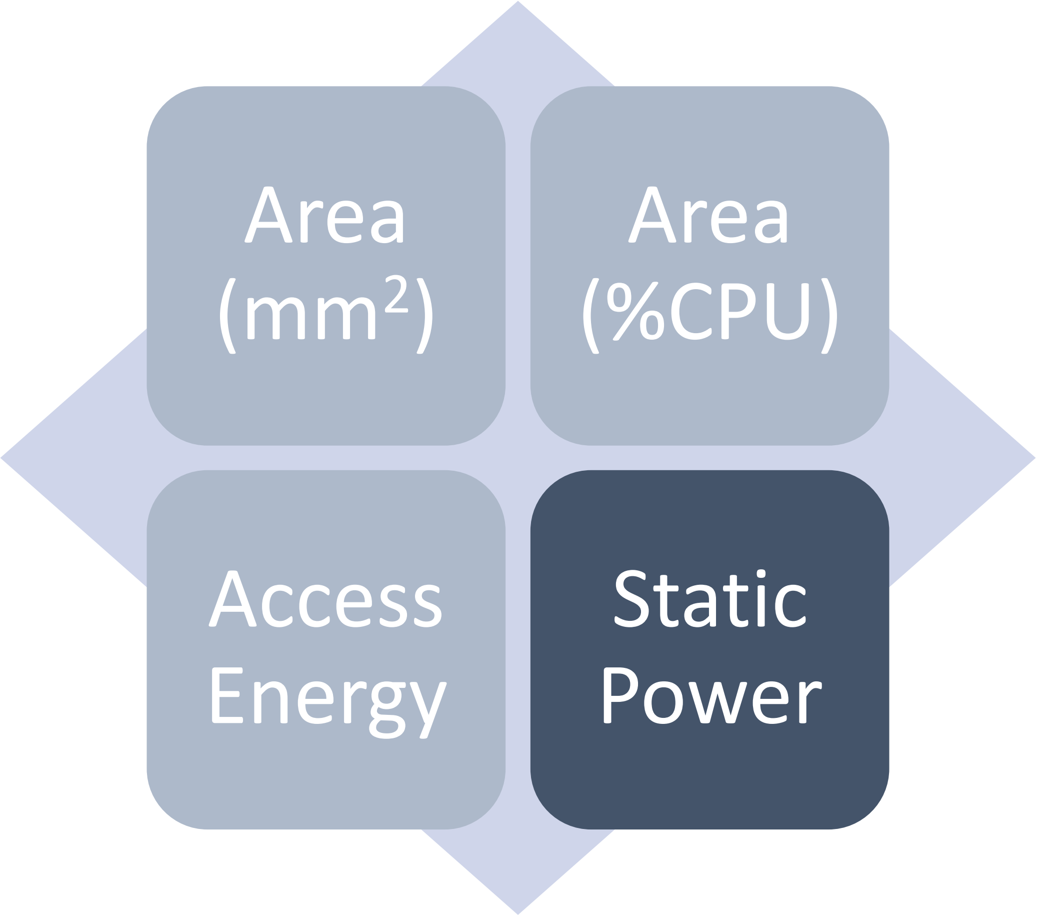
RowHammer threshold 32K

- PRoHIT, MRLoc → extremely efficient (because probabilistic)
- TWiCe < CBT << BlockHammer << Graphene → still relatively efficient

RowHammer threshold 1K

- Graphene x22.6, TWiCE x15.6, CBT x14 ↔ BlockHammer x4.9
- **New order: BlockHammer <<< TWiCE, CBT << Graphene**
  - **BlockHammer is <u>most efficient</u>!**

1. Hardware complexity analysis



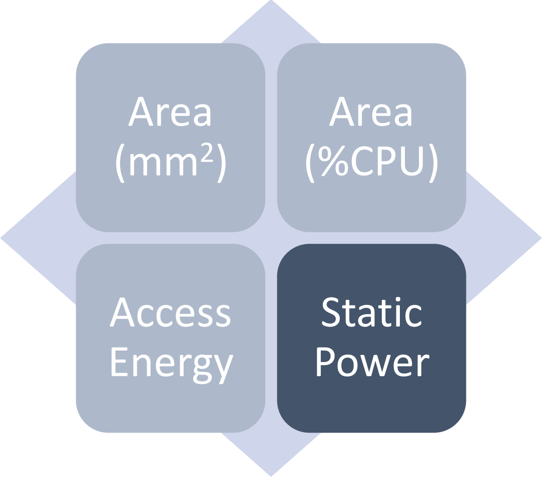
Area (mm^2)

Area (%CPU)

Access Energy

Static Power

1. Hardware complexity analysis


Area (mm^2)
Area (%CPU)
Access Energy
Static Power

RowHammer threshold 32K

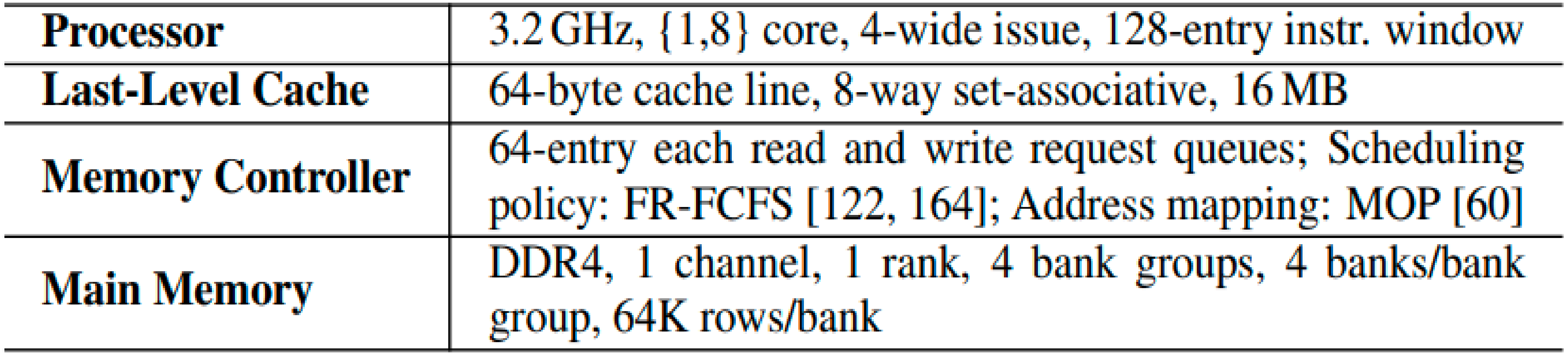
- PRoHIT, MRLoc → extremely efficient (because probabilistic)
- Graphene << TWiCe, BlockHammer << CBT → still relatively efficient

RowHammer threshold 1K

- Graphene x30.2, TWiCE x29.7, CBT x15.1 ↔ BlockHammer x9.9
- **New order: Graphene << BlockHammer <<< TWiCE << CBT**
  - **BlockHammer is catching up!**

2. Performance & energy consumption

- Setup: DDR4 memory



Processor	3.2 GHz, {1,8} core, 4-wide issue, 128-entry instr. window
Last-Level Cache	64-byte cache line, 8-way set-associative, 16 MB
Memory Controller	64-entry each read and write request queues; Scheduling policy: FR-FCFS [122, 164]; Address mapping: MOP [60]
Main Memory	DDR4, 1 channel, 1 rank, 4 bank groups, 4 banks/bank group, 64K rows/bank

Table 5: Simulated system configuration.