

# **AWARE : Adaptive Wear-leveling and Attack Re-mapping Engine**

We propose a new metric WinWD.

We develop AWARE, an adaptive wear levelling framework.

We introduce attack re-mapping within AWARE to isolate malicious instructions.

We identify potential attack vectors against AWARE and propose countermeasures.

## **Presenter**

Rahul Vigneswaran K

CS23MTECH02002

## **Course Instructor**

Dr. Shirshendu Das



# Table Of Contents

## 01 **Introduction**



- Motivation

## 02 **Proposed Methodology: AWARE**

- Estimation of Write Density
- Adaptive wear-levelling
- Attack Re-mapping

## 03 **Potential Attacks on Aware and Countermeasures**

# Motivation



Western Digital WD Blue SN580 PCIe Gen 4 NVMe SSD Internal Storage, 500GB

★★★★☆ 19,132

300+ bought in past month

₹3,426 M.R.P: ₹12,000 (71% off)



Save extra with No Cost EMI

FREE delivery **Thu, 5 Dec**

Or fastest delivery **Wed, 4 Dec**

Add to cart

# Motivation



Western Digital WD Blue SN580 PCIe Gen 4 NVMe SSD Internal Storage, 500GB

★★★★☆ [19,132](#)

300+ bought in past month

**₹3,426** M.R.P: ₹12,000 (71% off)

Save extra with No Cost EMI

FREE delivery **Thu, 5 Dec**

Or fastest delivery **Wed, 4 Dec**

Add to cart

Amazon's **Choice**



CORSAIR Vengeance 16GB RAM (1x16GB) DDR5 DRAM 5200MHz Memory Kit Black CMK16GX5M1B5200C40

★★★★☆ [14,179](#)

**₹4,199** M.R.P: ₹9,000 (53% off)



Save extra with No Cost EMI

FREE delivery **Wed, 4 Dec**

Or fastest delivery **Tomorrow, 3 Dec**

Add to cart

# Motivation



Western Digital WD Blue SN580 PCIe Gen 4 NVMe SSD Internal Storage, 500GB


★★★★☆ [19,132](#)  
300+ bought in past month

**₹3,426** M.R.P: ₹12,000 (71% off)  
Save extra with No Cost EMI

FREE delivery **Thu, 5 Dec**  
Or fastest delivery **Wed, 4 Dec**

Add to cart

Amazon's **Choice**



CORSAIR Vengeance 16GB RAM (1x16GB) DDR5 DRAM 5200MHz Memory Kit Black CMK16GX5M1B5200C40

★★★★☆ [14,179](#)

**₹4,199** M.R.P: ₹9,000 (53% off)  
Save extra with No Cost EMI

FREE delivery **Wed, 4 Dec**  
Or fastest delivery **Tomorrow, 3 Dec**

Add to cart



Intel Core i9-14900Kf LGA 1700 New Gaming Desktop Processor 24 Cores (8 P-Cores + 16 E-Cores) - Unlocked

★★★★☆ [1,625](#)

**₹53,297** M.R.P: ₹73,700 (28% off)  
Save extra with No Cost EMI


FREE delivery **Wed, 4 Dec**  
Or fastest delivery **Tomorrow, 3 Dec**

Add to cart

More Buying Choices  
₹53,249 ([4 new offers](#))



# Motivation



Western Digital WD Blue SN580 PCIe Gen 4 NVMe SSD Internal Storage, 500GB


★★★★☆ 19,132  
300+ bought in past month

**₹3,426** M.R.P: ₹12,000 (71% off)  
Save extra with No Cost EMI

FREE delivery **Thu, 5 Dec**  
Or fastest delivery **Wed, 4 Dec**

Add to cart

Amazon's Choice



CORSAIR Vengeance 16GB RAM (1x16GB) DDR5 DRAM 5200MHz Memory Kit Black CMK16GX5M1B5200C40

★★★★☆ 14,179

**₹4,199** M.R.P: ₹9,000 (53% off)  
Save extra with No Cost EMI

FREE delivery **Wed, 4 Dec**  
Or fastest delivery **Tomorrow, 3 Dec**

Add to cart



Intel Core i9-14900Kf LGA 1700 New Gaming Desktop Processor 24 Cores (8 P-Cores + 16 E-Cores) - Unlocked

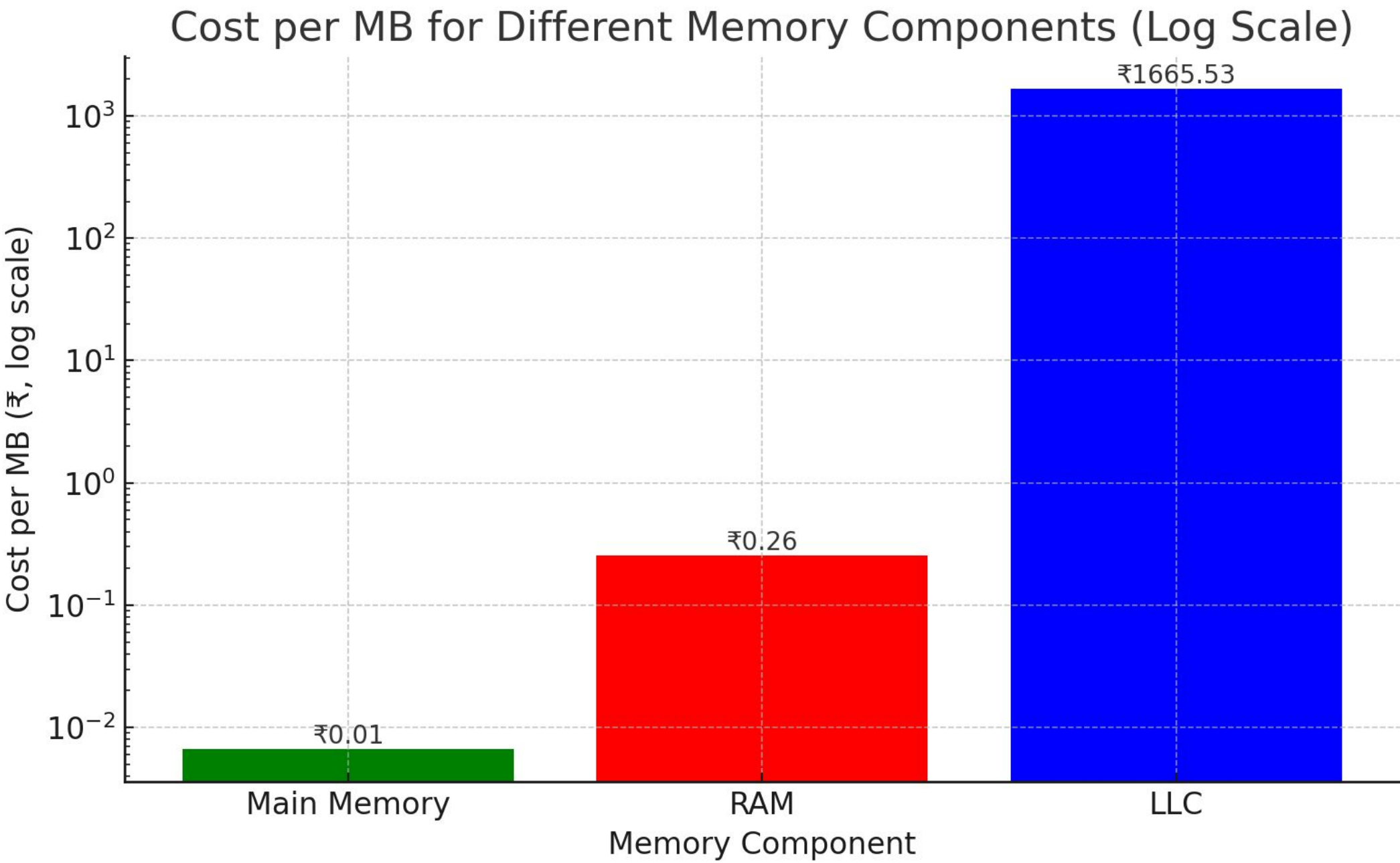
★★★★☆ 1,625

**₹53,297** M.R.P: ₹73,700 (28% off)  
Save extra with No Cost EMI

FREE delivery **Wed, 4 Dec**  
Or fastest delivery **Tomorrow, 3 Dec**

Add to cart

More Buying Choices  
₹53,249 (4 new offers)



# Proposed Methodology: AWARE

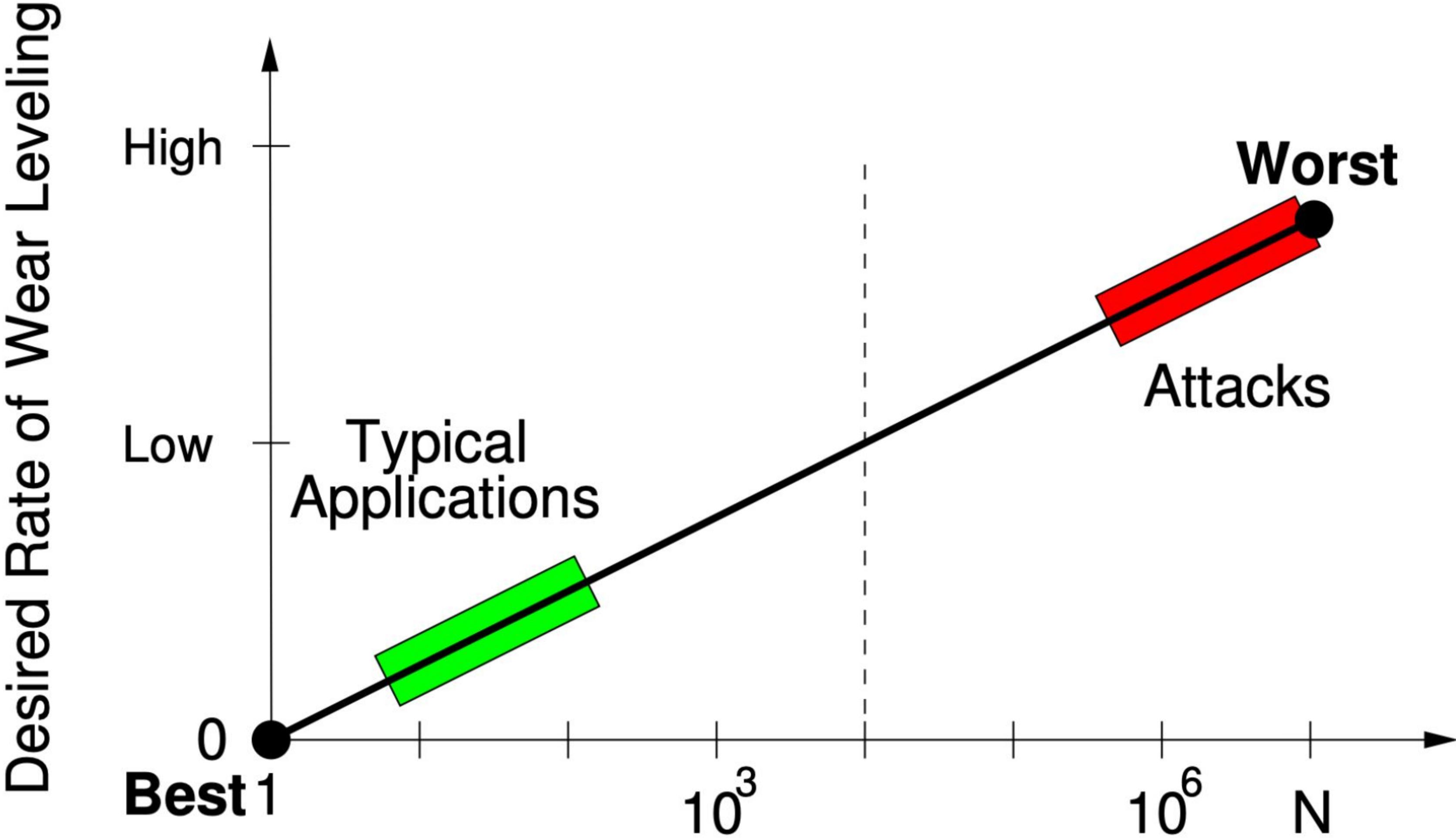


Fig. 1. Rate of wear leveling for different types of write stream

# Proposed Methodology: AWARE

**A**daptive **W**ear-levelling and **A**ttack **R**e-mapping **E**ngine



# Proposed Methodology: AWARE

**A**daptive **W**ear-levelling and **A**ttack **R**e-mapping **E**ngine



```
graph TD; A["Adaptive Wear-levelling and Attack Re-mapping Engine"] --> B["Adaptive Wear-levelling"]
```

The diagram consists of two rounded rectangular boxes. The top box contains the text "Adaptive Wear-levelling and Attack Re-mapping Engine". A curved arrow points from the bottom-left corner of this box to the top-left corner of a second box below it. The second box contains the text "Adaptive Wear-levelling".

**A**daptive **W**ear-levelling

# Proposed Methodology: AWARE

**A**daptive **W**ear-levelling and **A**ttack **R**e-mapping **E**ngine

```
graph TD; A["Adaptive Wear-levelling and Attack Re-mapping Engine"] --> B["Adaptive Wear-levelling"]; A --> C["Attack Re-mapping"];
```

**A**daptive **W**ear-levelling

**A**ttack **R**e-mapping

# Adaptive Wear-levelling : Estimating WinWD

$$\text{WinWD} = \frac{\text{Writes to a specific window}}{\text{Total number of writes to memory}}$$

$$\text{WayWD} = \frac{\text{Writes to a specific way}}{\text{Total number of writes to memory}}$$

# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100

# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

- 1) **Initialization.** Create an empty stack to hold up to  $s$  entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100



# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

- 1) **Initialization.** Create an empty stack to hold up to  $s$  entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.
- 2) **Processing Memory Writes.**
  - a) For each incoming memory write with address  $A$ , increment *WriteCounter* by 1.

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100

# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

- 1) **Initialization.** Create an empty stack to hold up to  $s$  entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.
- 2) **Processing Memory Writes.**
  - a) For each incoming memory write with address  $A$ , increment *WriteCounter* by 1.
  - b) Check if the window corresponding to  $A$  exists in the stack:

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100

# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

- 1) **Initialization.** Create an empty stack to hold up to  $s$  entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.
- 2) **Processing Memory Writes.**
  - a) For each incoming memory write with address  $A$ , increment *WriteCounter* by 1.
  - b) Check if the window corresponding to  $A$  exists in the stack:
    - i) If yes, increment the *HitCounter* and also increment the frequency for that window.

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100

# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

- 1) **Initialization.** Create an empty stack to hold up to  $s$  entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.
- 2) **Processing Memory Writes.**
  - a) For each incoming memory write with address  $A$ , increment *WriteCounter* by 1.
  - b) Check if the window corresponding to  $A$  exists in the stack:
    - i) If yes, increment the *HitCounter* and also increment the frequency for that window.
    - ii) If no, insert the window into the stack with probability  $p$ .

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100



# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

- 1) **Initialization.** Create an empty stack to hold up to  $s$  entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.
- 2) **Processing Memory Writes.**
  - a) For each incoming memory write with address  $A$ , increment *WriteCounter* by 1.
  - b) Check if the window corresponding to  $A$  exists in the stack:
    - i) If yes, increment the *HitCounter* and also increment the frequency for that window.
    - ii) If no, insert the window into the stack with probability  $p$ .
  - c) If the stack is full, evict the window with the least frequency and replace it with the new window.

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100



# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

- 1) **Initialization.** Create an empty stack to hold up to  $s$  entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.
- 2) **Processing Memory Writes.**
  - a) For each incoming memory write with address  $A$ , increment *WriteCounter* by 1.
  - b) Check if the window corresponding to  $A$  exists in the stack:
    - i) If yes, increment the *HitCounter* and also increment the frequency for that window.
    - ii) If no, insert the window into the stack with probability  $p$ .
  - c) If the stack is full, evict the window with the least frequency and replace it with the new window.
- 3) **Aging Mechanism.** Periodically decrement the frequency of all windows in the stack to account for aging, ensuring that stale entries are naturally evicted over time.

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100

# Adaptive Wear-levelling : Estimating WinWD

**Efficient Estimation of WinWD.** The LLC is divided into  $n$  windows, and we maintain a stack of size  $s$  to track active windows (Table I shows an example stack.). Additionally, we use two counters: *WriteCounter* and *HitCounter*, initialized to zero. The estimation process proceeds as follows:

- 1) **Initialization.** Create an empty stack to hold up to  $s$  entries (each representing a window) and set both counters (*WriteCounter* and *HitCounter*) to 0.
- 2) **Processing Memory Writes.**
  - a) For each incoming memory write with address  $A$ , increment *WriteCounter* by 1.
  - b) Check if the window corresponding to  $A$  exists in the stack:
    - i) If yes, increment the *HitCounter* and also increment the frequency for that window.
    - ii) If no, insert the window into the stack with probability  $p$ .
  - c) If the stack is full, evict the window with the least frequency and replace it with the new window.
- 3) **Aging Mechanism.** Periodically decrement the frequency of all windows in the stack to account for aging, ensuring that stale entries are naturally evicted over time.
- 4) **Compute WinWD.** The *Write Density* for a window is calculated as:

$$\text{WinWD} = \frac{\text{HitCounter}}{\text{WriteCounter}}.$$

TABLE I  
EXAMPLE OF THE STACK TRACKING MECHANISM FOR WRITE DENSITY ESTIMATION

Window ID	Frequency (HitCounter)
W3	15
W7	12
W2	10
W5	8
W1	6
HitCounter (Sum)	51
Total Writes	100

# Adaptive Wear-levelling : Gap-Window

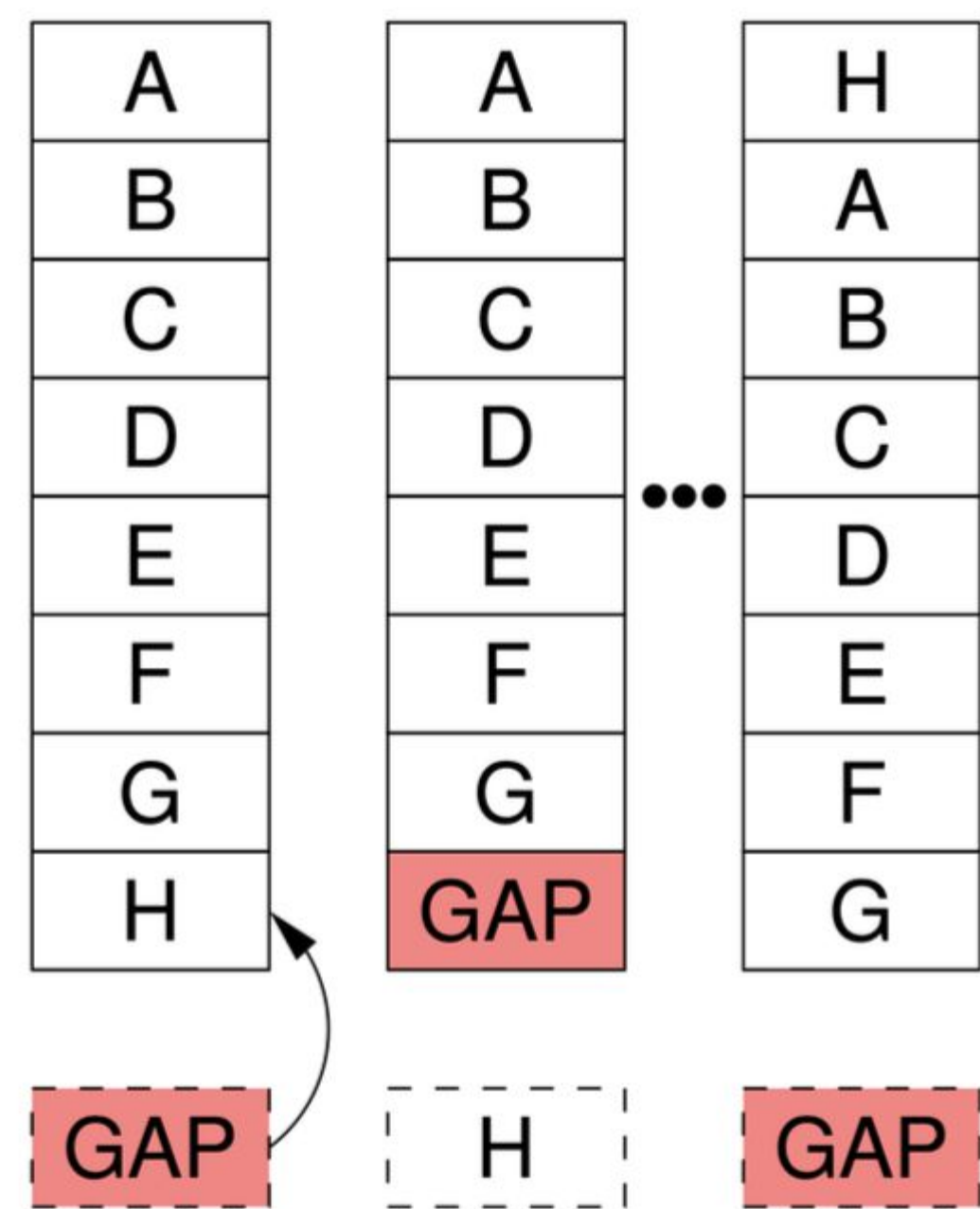


Fig. 3. Illustration of how Gap-Window works.

# Adaptive Wear-levelling : Gap-Window

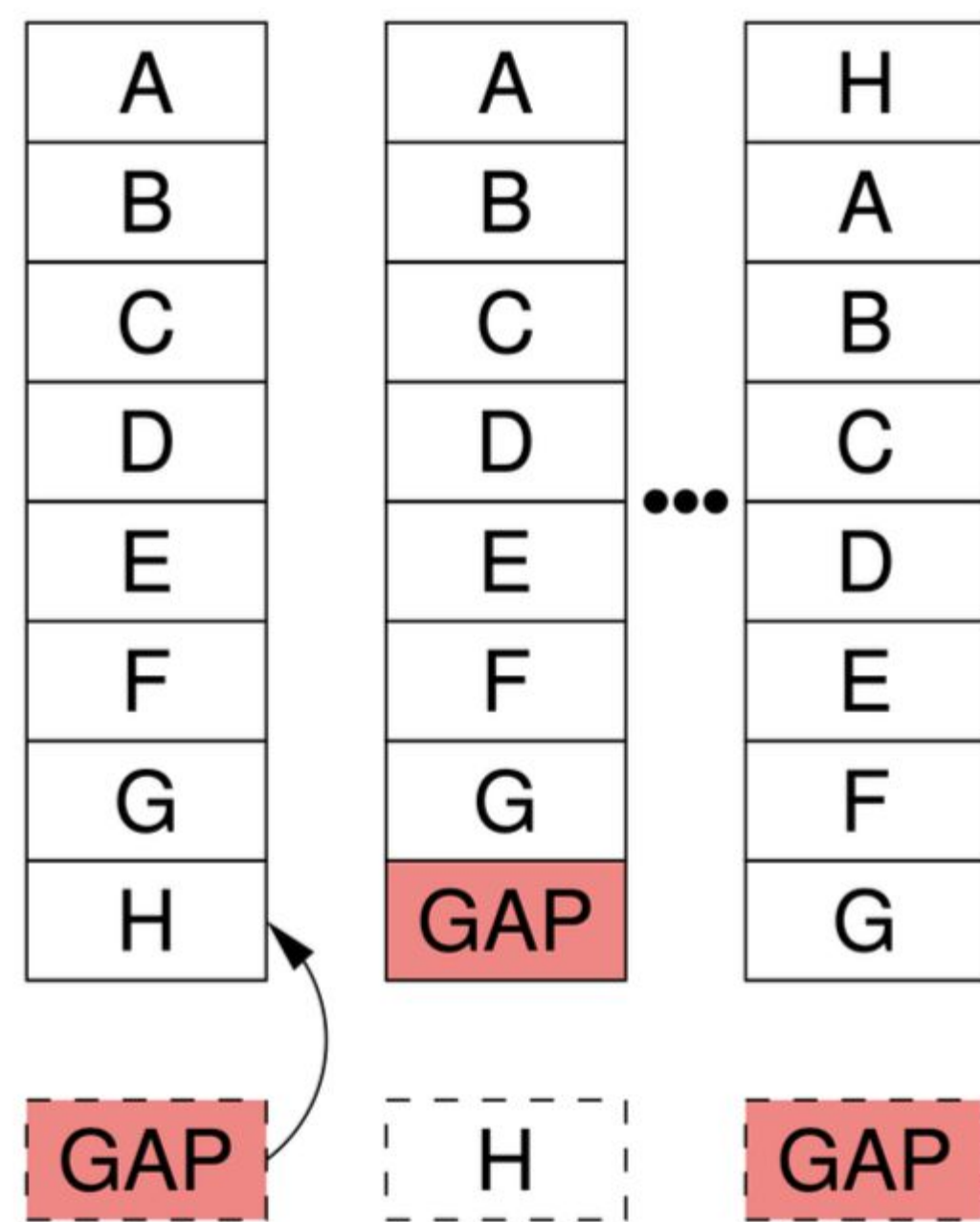
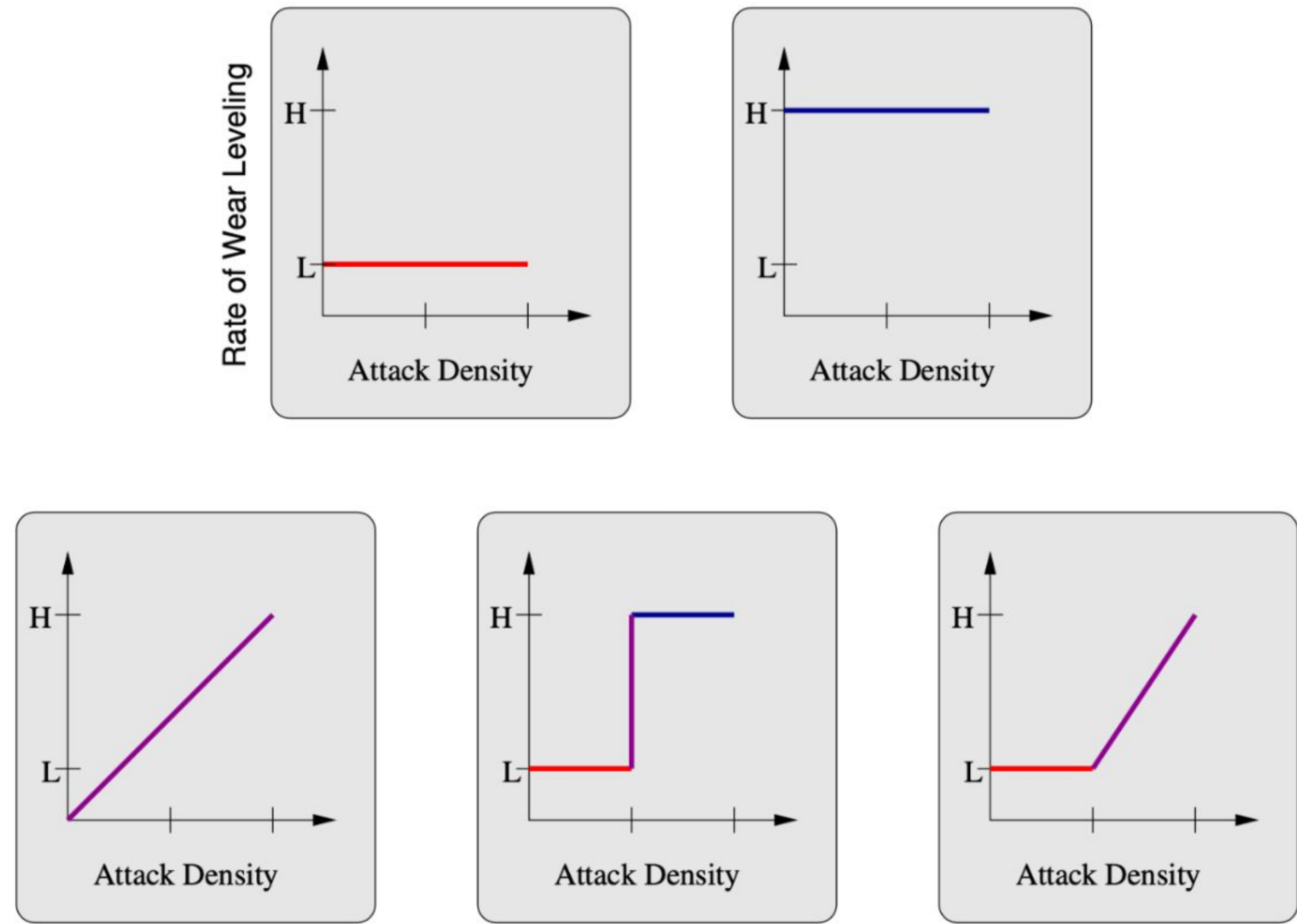


Fig. 3. Illustration of how Gap-Window works.





# Attack Re-mapping

The attack re-mapping mechanism consists of the following steps:



# Attack Re-mapping

The attack re-mapping mechanism consists of the following steps:

- 1) **Initialization.** Create an empty stack capable of holding up to  $q$  PC entries. Each entry consists of:
  - **PC:** The program counter associated with the instruction.
  - **Frequency:** The number of times the PC has been associated with a memory write.
  - **Valid:** A flag indicating whether the PC is still actively tracked or has been invalidated.

TABLE II  
EXAMPLE OF A PC-STACK USED IN ATTACK RE-MAPPING

PC	Frequency	Valid
0x400123	15	Yes
0x400456	12	Yes
0x400789	8	Yes
0x400ABC	5	No
0x400DEF	3	Yes

# Attack Re-mapping

The attack re-mapping mechanism consists of the following steps:

- 1) **Initialization.** Create an empty stack capable of holding up to  $q$  PC entries. Each entry consists of:
  - **PC:** The program counter associated with the instruction.
  - **Frequency:** The number of times the PC has been associated with a memory write.
  - **Valid:** A flag indicating whether the PC is still actively tracked or has been invalidated.
- 2) **Processing Memory Writes.** For each incoming memory write associated with address  $A$ :
  - a) Identify the PC responsible for the write operation.

TABLE II  
EXAMPLE OF A PC-STACK USED IN ATTACK RE-MAPPING

PC	Frequency	Valid
0x400123	15	Yes
0x400456	12	Yes
0x400789	8	Yes
0x400ABC	5	No
0x400DEF	3	Yes

# Attack Re-mapping

The attack re-mapping mechanism consists of the following steps:

- 1) **Initialization.** Create an empty stack capable of holding up to  $q$  PC entries. Each entry consists of:
  - **PC:** The program counter associated with the instruction.
  - **Frequency:** The number of times the PC has been associated with a memory write.
  - **Valid:** A flag indicating whether the PC is still actively tracked or has been invalidated.
- 2) **Processing Memory Writes.** For each incoming memory write associated with address  $A$ :
  - a) Identify the PC responsible for the write operation.
  - b) Check if the PC exists in the PC-Stack:

TABLE II  
EXAMPLE OF A PC-STACK USED IN ATTACK RE-MAPPING

PC	Frequency	Valid
0x400123	15	Yes
0x400456	12	Yes
0x400789	8	Yes
0x400ABC	5	No
0x400DEF	3	Yes

# Attack Re-mapping

The attack re-mapping mechanism consists of the following steps:

- 1) **Initialization.** Create an empty stack capable of holding up to  $q$  PC entries. Each entry consists of:
  - **PC:** The program counter associated with the instruction.
  - **Frequency:** The number of times the PC has been associated with a memory write.
  - **Valid:** A flag indicating whether the PC is still actively tracked or has been invalidated.
- 2) **Processing Memory Writes.** For each incoming memory write associated with address  $A$ :
  - a) Identify the PC responsible for the write operation.
  - b) Check if the PC exists in the PC-Stack:
    - i) If it exists, increment its frequency counter.

TABLE II  
EXAMPLE OF A PC-STACK USED IN ATTACK RE-MAPPING

PC	Frequency	Valid
0x400123	15	Yes
0x400456	12	Yes
0x400789	8	Yes
0x400ABC	5	No
0x400DEF	3	Yes



# Attack Re-mapping

The attack re-mapping mechanism consists of the following steps:

- 1) **Initialization.** Create an empty stack capable of holding up to  $q$  PC entries. Each entry consists of:
  - **PC:** The program counter associated with the instruction.
  - **Frequency:** The number of times the PC has been associated with a memory write.
  - **Valid:** A flag indicating whether the PC is still actively tracked or has been invalidated.
- 2) **Processing Memory Writes.** For each incoming memory write associated with address  $A$ :
  - a) Identify the PC responsible for the write operation.
  - b) Check if the PC exists in the PC-Stack:
    - i) If it exists, increment its frequency counter.
    - ii) If it does not exist, search for an invalid entry to replace. If no invalid entries are found, evict the PC with the lowest frequency and replace it with the new PC.

TABLE II  
EXAMPLE OF A PC-STACK USED IN ATTACK RE-MAPPING

PC	Frequency	Valid
0x400123	15	Yes
0x400456	12	Yes
0x400789	8	Yes
0x400ABC	5	No
0x400DEF	3	Yes



# Attack Re-mapping

The attack re-mapping mechanism consists of the following steps:

- 1) **Initialization.** Create an empty stack capable of holding up to  $q$  PC entries. Each entry consists of:
  - **PC:** The program counter associated with the instruction.
  - **Frequency:** The number of times the PC has been associated with a memory write.
  - **Valid:** A flag indicating whether the PC is still actively tracked or has been invalidated.
- 2) **Processing Memory Writes.** For each incoming memory write associated with address  $A$ :
  - a) Identify the PC responsible for the write operation.
  - b) Check if the PC exists in the PC-Stack:
    - i) If it exists, increment its frequency counter.
    - ii) If it does not exist, search for an invalid entry to replace. If no invalid entries are found, evict the PC with the lowest frequency and replace it with the new PC.

- 3) **Aging Mechanism.** Periodically decrement the frequency of all PCs in the stack to account for aging. This mechanism ensures that stale entries are evicted over time, prioritizing recently active malicious PCs.

TABLE II  
EXAMPLE OF A PC-STACK USED IN ATTACK RE-MAPPING

PC	Frequency	Valid
0x400123	15	Yes
0x400456	12	Yes
0x400789	8	Yes
0x400ABC	5	No
0x400DEF	3	Yes

# Attack Re-mapping

The attack re-mapping mechanism consists of the following steps:

- 1) **Initialization.** Create an empty stack capable of holding up to  $q$  PC entries. Each entry consists of:
  - **PC:** The program counter associated with the instruction.
  - **Frequency:** The number of times the PC has been associated with a memory write.
  - **Valid:** A flag indicating whether the PC is still actively tracked or has been invalidated.
- 2) **Processing Memory Writes.** For each incoming memory write associated with address  $A$ :
  - a) Identify the PC responsible for the write operation.
  - b) Check if the PC exists in the PC-Stack:
    - i) If it exists, increment its frequency counter.
    - ii) If it does not exist, search for an invalid entry to replace. If no invalid entries are found, evict the PC with the lowest frequency and replace it with the new PC.

- 3) **Aging Mechanism.** Periodically decrement the frequency of all PCs in the stack to account for aging. This mechanism ensures that stale entries are evicted over time, prioritizing recently active malicious PCs.
- 4) **Gap-Window Switching.** During the Gap-Window switching process:
  - Identify the PC with the maximum frequency in the PC-Stack and remap its associated memory writes to volatile DRAM.
  - Invalidate PCs with frequencies below a predefined threshold, freeing space in the PC-Stack for new entries.
  - When the volatile DRAM memory is filled, remap the entries back to the NVM, ensuring minimal disruption to system operations.

TABLE II  
EXAMPLE OF A PC-STACK USED IN ATTACK RE-MAPPING

PC	Frequency	Valid
0x400123	15	Yes
0x400456	12	Yes
0x400789	8	Yes
0x400ABC	5	No
0x400DEF	3	Yes

# Potential attack on AWARE and Countermeasure

To counter this vulnerability, we propose the following mitigation strategy: when a PC is observed to cycle through the remapping process (from NVM to volatile memory and back to NVM) a certain predefined number of times, the PC should be flagged as malicious. Once flagged:

# Potential attack on AWARE and Countermeasure

To counter this vulnerability, we propose the following mitigation strategy: when a PC is observed to cycle through the remapping process (from NVM to volatile memory and back to NVM) a certain predefined number of times, the PC should be flagged as malicious. Once flagged:

- All subsequent writes originating from this PC should be bypassed.



# Potential attack on AWARE and Countermeasure

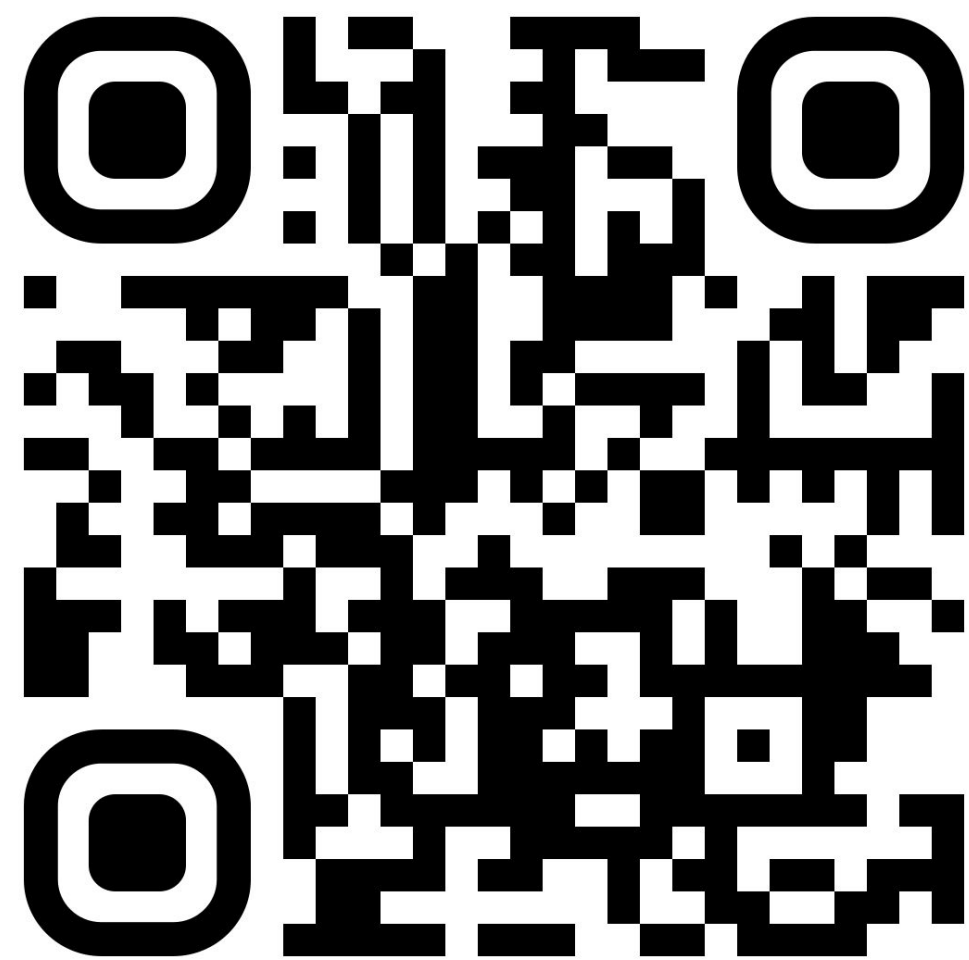
To counter this vulnerability, we propose the following mitigation strategy: when a PC is observed to cycle through the remapping process (from NVM to volatile memory and back to NVM) a certain predefined number of times, the PC should be flagged as malicious. Once flagged:

- All subsequent writes originating from this PC should be bypassed.
- The flagged PC should be reported to the user or system administrator for further action.



# Citations

- M. K. Qureshi, A. Sez nec, L. A. Lastras, and M. M. Franceschini, “Practical and secure pcm systems by online detection of malicious write streams,” HPCA 2011.



Rahul Vigneswaran K

## AWARE: Adaptive Wear-levelling and Attack Re-mapping Engine

Rahul Vigneswaran  
Indian Institute of Technology Hyderabad  
India  
cs23mtech02002@iith.ac.in

**Abstract**—Non-Volatile Memory (NVM) technology faces significant adoption challenges in Last-Level Caches (LLCs) due to its inherent wear vulnerability, reducing durability and exposing systems to adversarial attacks. To address these issues, we propose AWARE (Adaptive Wear-levelling and Attack Re-mapping Engine), a novel framework that seamlessly integrates adaptive wear-leveling with attack mitigation through intelligent remapping. Central to AWARE is the introduction of WinWD, a pivotal metric that distinguishes normal workloads from malicious activities. Using this metric, AWARE employs an efficient estimation mechanism and a dynamic Gap-Window strategy to balance memory utilization and enhance wear resilience. Furthermore, AWARE features an attack re-mapping subroutine that isolates malicious instructions to volatile memory, thereby safeguarding NVM-based LLCs from prolonged wear-induced failures and attacks.

### I. INTRODUCTION

Non-Volatile Memory (NVM) technology has garnered significant attention due to its high read speeds, making it an attractive option for storage applications. However, a critical drawback of NVM cells is their susceptibility to wear, which limits their durability and long-term reliability. This inherent vulnerability poses challenges across various use cases. For instance, a customer nearing the end of their warranty period might deliberately execute repeated writes to accelerate wear and claim a replacement device. Even more concerning, malicious actors could exploit this weakness for unethical purposes, potentially causing significant damage.

While the issue of wear is problematic in external main memory, such components are relatively inexpensive and straightforward to replace. However, the challenge becomes far more severe when NVMs are considered for use in Last-Level Caches (LLCs). LLCs are typically embedded within the processor chip, meaning that wear-induced failure necessitates replacing the entire system—a prohibitively costly affair. This makes NVMs a less attractive option for manufacturers despite their substantial advantages, such as energy efficiency and performance.

Without addressing the wear issue, NVM-based LLCs are unlikely to see widespread adoption. To unlock the potential of NVM in LLCs, it is imperative to develop effective mechanisms that mitigate wear while preserving their inherent benefits. This challenge of balancing wear mitigation and

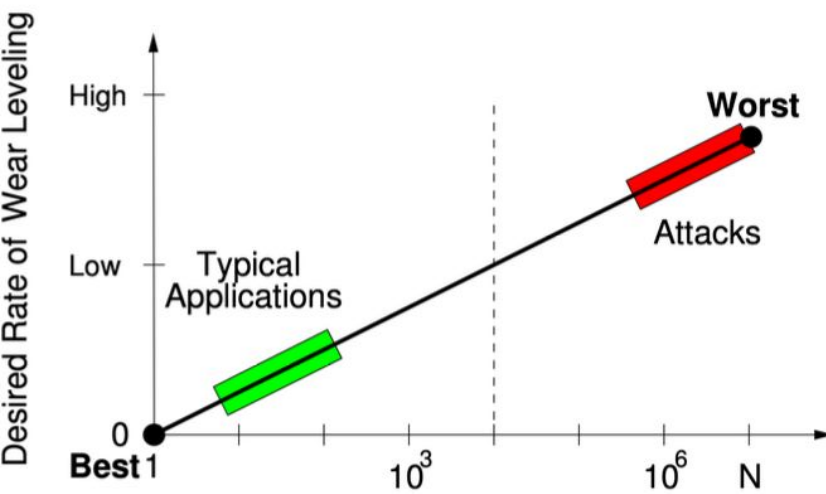


Fig. 1. Rate of wear leveling for different types of write stream

performance optimization poses a natural and compelling research problem, warranting innovative solutions.

To differentiate between adversarial attacks and general application workloads, a reliable metric is essential. Inspired by [1], we propose the metric of Write Density, which serves as a robust indicator to distinguish between attack patterns and normal application behaviors within the LLC. As illustrated in Figure 1 and discussed in [1], there is stark difference between the rate of wear-levelling required for attacks and a normal application. We use Write Density as a proxy to effectively distinguish between the behaviors of normal applications and adversarial attacks. This differentiation, which is critical for ensuring the reliability of NVM-based LLCs, is elaborated upon in subsequent sections.

Building upon this metric, we introduce AWARE (Adaptive Wear-levelling and Attack Re-mapping Engine), an adaptive wear-leveling algorithm designed to dynamically adjust its leveling subroutine based on observed Write Density. Furthermore, AWARE isolates repeat offenders by redirecting their requests to a separate volatile cache, thereby mitigating their impact on the NVM-based LLC and enhancing overall system resilience.

Our core contributions in this work are as follows:

- 1) We propose *Window Write Density (WinWD)*, a practical proxy metric that adjusts the wear-leveling rate dynamically, ensuring balanced memory utilization.
- 2) We develop *AWARE*, an adaptive framework that integrates Write Density with dynamic wear-leveling

Done as part of the coursework for Advanced Computer Architecture (CS5363).