FROM PHYSICS TO EBPF: QUANTIFYING FLASH WEAR IN EMBEDDED SYSTEMS

BLAKE HILDEBRAND

```
> mmc extme read /dev/blake

eMMC Host Info       [EXT_ME_HOST]:        Blake Hildebrand - @bahildebrand
eMMC Job Title       [EXT_ME_JOB_TITLE]:   Software Engineer @NordicSemi

eMMC Work Experience [EXT_ME_WORK_EXP_A]:  microcontrollers
eMMC Work Experience [EXT_ME_WORK_EXP_B]:  embedded linux
eMMC Work Experience [EXT_ME_WORK_EXP_C]:  backend - distributed systems
eMMC Companies       [EXT_ME_COMPANIES]:   Garmin - Amazon Robotics - AWS

eMMC Current Gig     [EXT_ME_CURRENT_GIG]: Develop memfaultd and help embedded engineers ship faster
```

```
> mmc extme read /dev/blake

eMMC Host Info       [EXT_ME_HOST]:        Blake Hildebrand - @bahildebrand
eMMC Job Title       [EXT_ME_JOB_TITLE]:   Software Engineer @NordicSemi

eMMC Work Experience [EXT_ME_WORK_EXP_A]:  microcontrollers
eMMC Work Experience [EXT_ME_WORK_EXP_B]:  embedded linux
eMMC Work Experience [EXT_ME_WORK_EXP_C]:  backend - distributed systems
eMMC Companies       [EXT_ME_COMPANIES]:   Garmin - Amazon Robotics - AWS

eMMC Current Gig     [EXT_ME_CURRENT_GIG]: Develop memfaultd and help embedded engineers ship faster
```

```
> mmc extme read /dev/blake

eMMC Host Info       [EXT_ME_HOST]:        Blake Hildebrand - @bahildebrand
eMMC Job Title       [EXT_ME_JOB_TITLE]:   Software Engineer @NordicSemi

eMMC Work Experience [EXT_ME_WORK_EXP_A]:  microcontrollers
eMMC Work Experience [EXT_ME_WORK_EXP_B]:  embedded linux
eMMC Work Experience [EXT_ME_WORK_EXP_C]:  backend - distributed systems
eMMC Companies       [EXT_ME_COMPANIES]:   Garmin - Amazon Robotics - AWS

eMMC Current Gig     [EXT_ME_CURRENT_GIG]: Develop memfaultd and help embedded engineers ship faster
```

```
> mmc extme read /dev/blake

eMMC Host Info       [EXT_ME_HOST]:        Blake Hildebrand - @bahildebrand
eMMC Job Title       [EXT_ME_JOB_TITLE]:   Software Engineer @NordicSemi

eMMC Work Experience [EXT_ME_WORK_EXP_A]:  microcontrollers
eMMC Work Experience [EXT_ME_WORK_EXP_B]:  embedded linux
eMMC Work Experience [EXT_ME_WORK_EXP_C]:  backend - distributed systems
eMMC Companies       [EXT_ME_COMPANIES]:   Garmin - Amazon Robotics - AWS

eMMC Current Gig     [EXT_ME_CURRENT_GIG]: Develop memfaultd and help embedded engineers ship faster
```

WHAT WE'LL COVER

1. How NAND flash is organized
2. What "wear" actually is (the physics)
3. Wear leveling, and what the FTL does for us
4. Detecting and monitoring wear in production

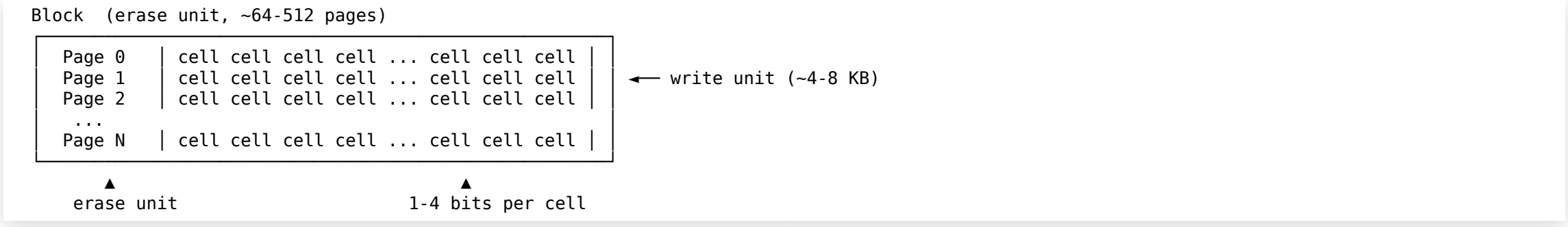
WHY DO WE CARE?

- NAND flash is everywhere, from nVMEs down to the eMMC in your embedded device
- It wears out, and eventually it stops working
- In a data center you swap the drive
- On a fleet of embedded devices, you can't

HOW NAND IS ORGANIZED

- **Page**: smallest writable unit (~4-8 KB)
- **Block**: smallest erasable unit (~64-512 pages)
- Programming a bit only moves it `1` → `0`
- To go back to `1`, you erase the **entire block**

BLOCK, PAGE, CELL



Writes happen one page at a time. Erases nuke the entire enclosing block.

THE ERASE-BEFORE-WRITE CONSTRAINT

Want to change 1 KB of a 4 KB page?

- Read the whole block into the controller
- Erase the block
- Write the modified block back

Bytes *requested* ≠ bytes *actually written*.

WRITE AMPLIFICATION FACTOR

WAF = bytes written to flash ÷ bytes the host requested

- Sequential, well-buffered writes: **1-2×**
- Lots of small random writes: **10×+**
- Workload-dependent, but always ≥ 1

MANAGED NAND (EMMC)

- NAND + a controller in one package
- Follows the JEDEC JESD84 spec
- Controller handles wear leveling and bad-block management
- Standardized register interface, including diagnostic ones we'll come back to

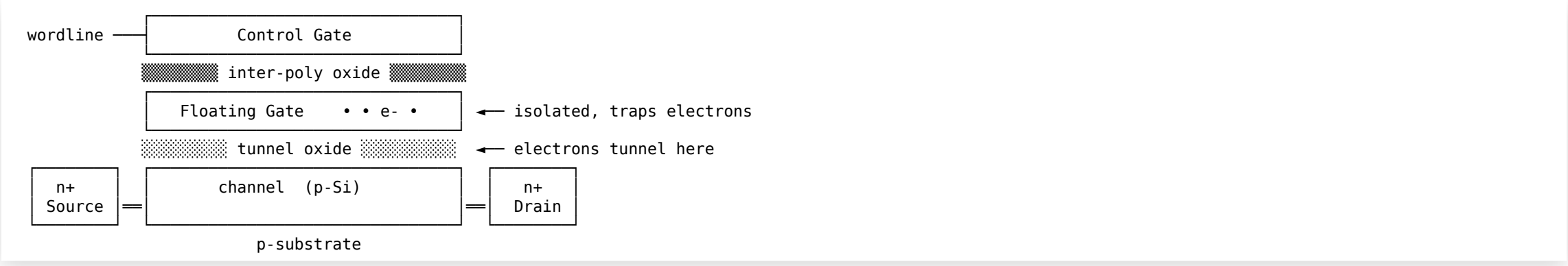
THE PHYSICS

CELLS, BRIEFLY

Type	Bits / cell	Endurance (P/E)
SLC	1	~100,000
MLC	2	~10,000
TLC	3	~3,000
QLC	4	~1,000

More bits/cell, more density, less endurance.

A FLOATING GATE CELL



Trapped electrons on the floating gate shift the threshold voltage. That shift *is* the stored bit.

READING A CELL

- Apply a **reference voltage** at the control gate
- Trapped electrons on the floating gate oppose that field - they raise the threshold voltage (V_t) needed to turn the channel on
- Compare against fixed voltage bands → decode which state the cell is in

State	Floating gate	V_t	Bit (SLC)
Erased	no electrons	~ -3 V to -1 V	`1`
Programmed	many electrons	~ +1 V to +3 V	`0`

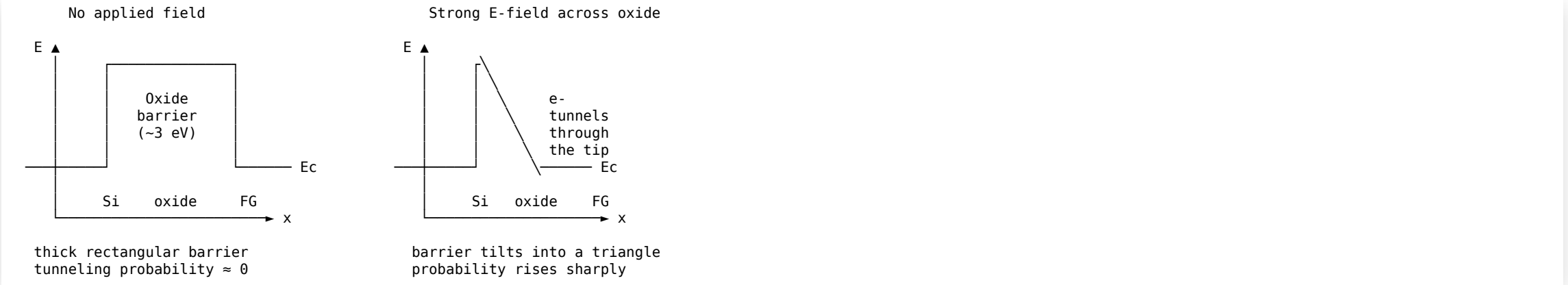
MLC/TLC/QLC slice the same voltage range into more bands. More bands → tighter margins → less tolerance for drift.

FOWLER-NORDHEIM TUNNELING

- Apply a strong electric field across the oxide
- The energy barrier tilts. What was a tall rectangle becomes a narrow triangle
- Electrons tunnel through the short triangular tip with much higher probability
- Reverse the field and they tunnel back out. That's the erase

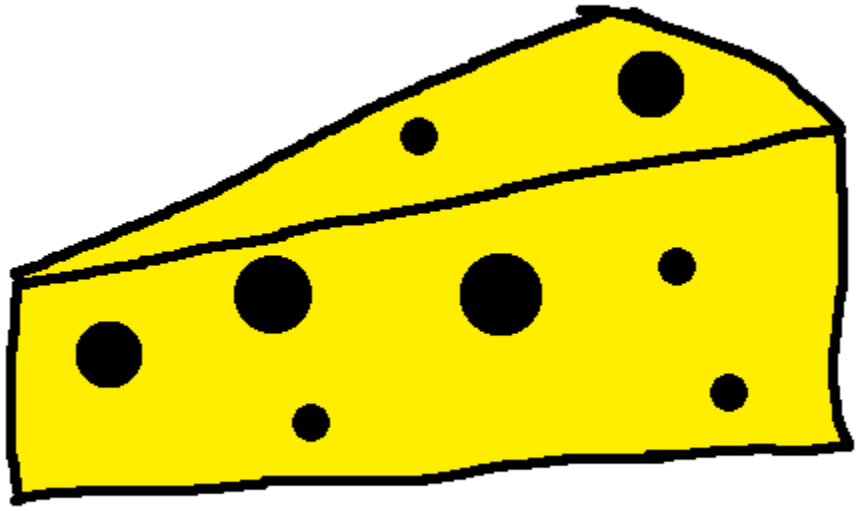
The barrier doesn't physically deform. It's the energy landscape that changes when the field is on.

FOWLER-NORDHEIM TUNNELING



Reversing the field flips the triangle to the other side - electrons tunnel back out. That's the erase.

TUNNEL OXIDE AFTER WEAR (ARTIST RENDITION)



WEAR LEVELING

P/E CYCLES

- "Program/Erase cycles": rated number of times a block can be cycled
- Manufacturer-provided, probabilistic in practice
- TLC eMMC: ~3,000 P/E
- You don't get to pick which block dies first

THE HOT-BLOCK PROBLEM

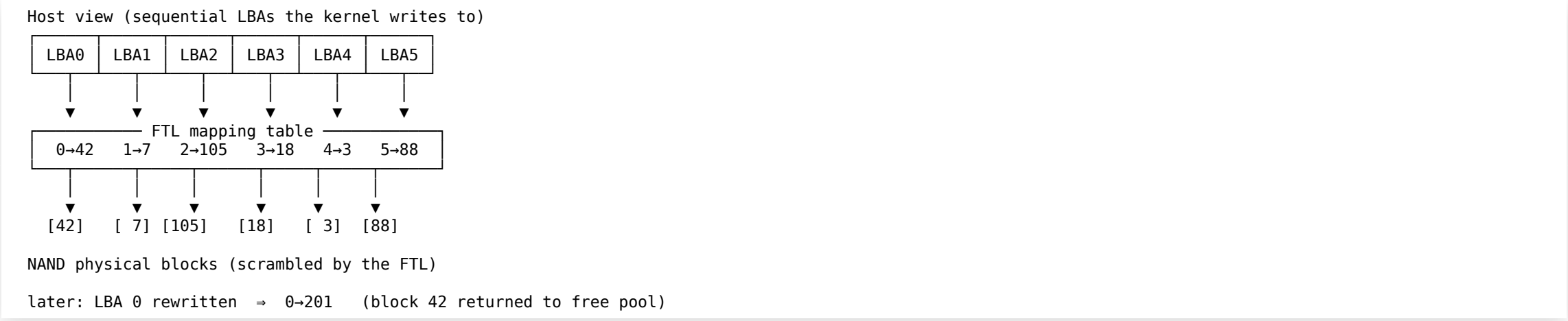
One config file gets rewritten every second.

- Same logical address → same physical block, every time
- That block burns through its P/E budget
- The other 99% of the device is barely touched
- Device fails, with 99% of its endurance unused

THE FLASH TRANSLATION LAYER

- The kernel sees logical block addresses
- The FTL (inside the eMMC controller) maps logical to physical
- Same logical address can land on a different physical block tomorrow
- This is what makes wear leveling possible

THE FLASH TRANSLATION LAYER

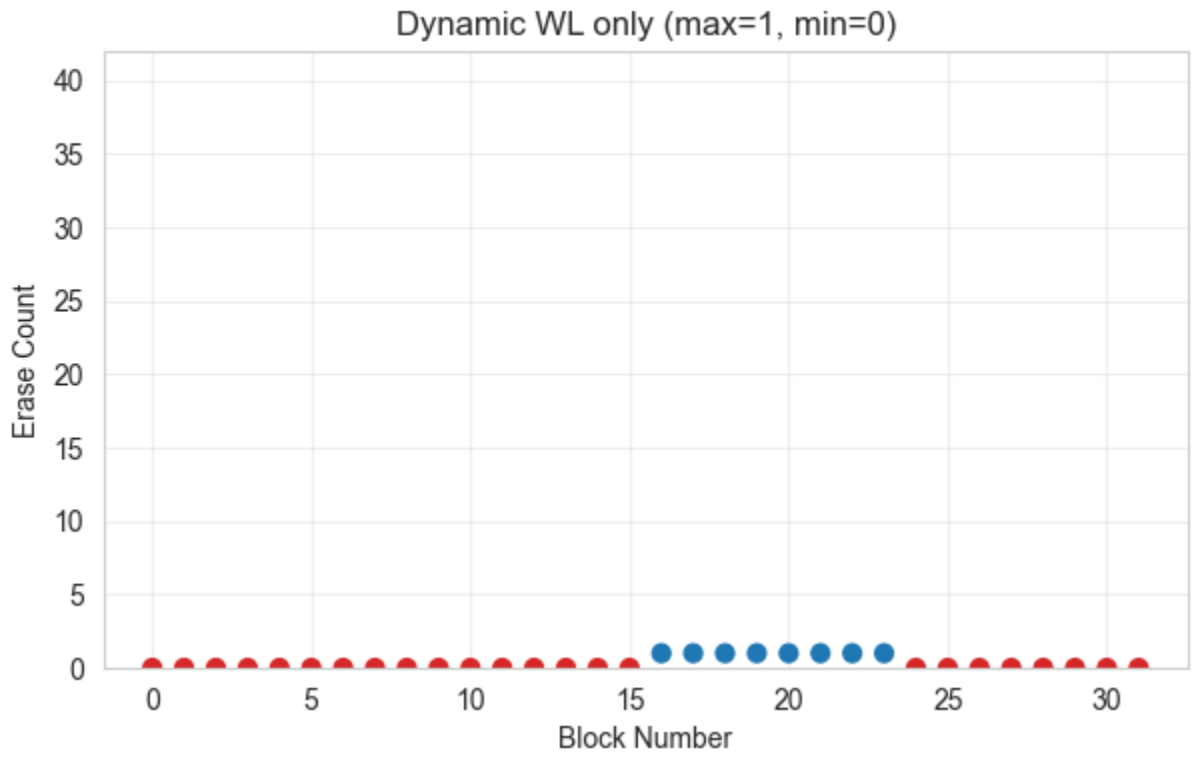


One mapping-table lookup is all that stands between a logical address and the physical block that gets the wear.

DYNAMIC WEAR LEVELING

- On every write, pick the freshest block from the free pool
- Rotate writes across blocks with the lowest P/E counts
- Works great for data that actually changes
- Cold data (config files, firmware) just sits there hogging fresh blocks

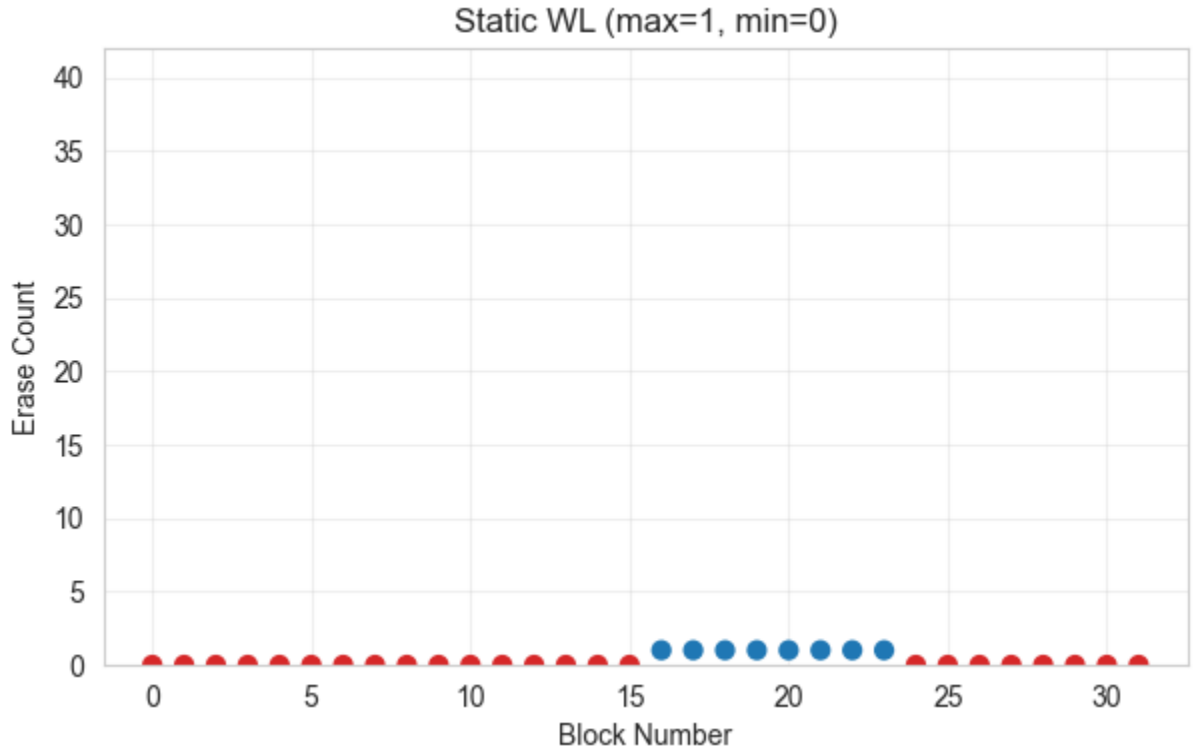
DYNAMIC WEAR LEVELING



STATIC WEAR LEVELING

- Watch the gap between most- and least-worn blocks
- When the gap crosses a threshold, *move cold data into worn blocks*
- Costs you write amplification, since you're writing data nobody asked you to write

STATIC WEAR LEVELING



HYBRID IN PRACTICE

- Real controllers do both
- Dynamic for the steady state, static when the gap grows
- Tuning lives entirely inside the controller firmware
- From the host side: it's a black box. We can't watch it work.

DETECTING & MONITORING WEAR

TERABYTES WRITTEN (TBW)

The standard lifetime projection:

```
TBW = Device Capacity (GB) × P/E Cycles ÷ WAF × 1000
```

- Manufacturer rating on the datasheet
- Tells you when the device *statistically* hits end-of-life
- Useless unless you can actually measure bytes written

WHAT `/proc/diskstats` GIVES US

```
179       0 mmcblk0  57839 46346 3776180 24107 2868024 458991 39327918 1458693 ...
             ^^^^^^^          ^^^^^                                  ^^^^^^^^
             device           reads completed                        sectors written
```

- Field 1: reads completed
- Field 5: writes completed
- Field 7: **sectors written**. Multiply by 512 for bytes
- Sample at an interval, diff, you have a rate

READING DISKSTATS IN PRACTICE

It's a plain text file. awk is enough.

```
$ awk '$3 == "mmcblk0" { print $10 * 512 }' /proc/diskstats
20136093696    # bytes written since boot
```

- Sample at t, again at $t + \Delta t$, subtract → bytes/sec
- Field 6 (sectors read) gives you the read side the same way
- No root, no kernel modules, no daemon - just open the file

This is what `iostat`, `sar`, `collectd`, and Prometheus' node_exporter are all doing under the hood.

EMMC EXT_CSD: THE LIFETIME BYTES

The eMMC controller exposes 512 bytes of EXT_CSD over JEDEC. Four bytes matter for wear:

Byte	Field	Meaning
192	EXT_CSD_REV	Spec rev (must be ≥ 7)
267	PRE_EOL_INFO	Normal / Warning / Urgent
268	DEVICE_LIFE_TIME_EST_TYP_A	MLC / TLC pool
269	DEVICE_LIFE_TIME_EST_TYP_B	SLC cache pool

Lifetime bytes encode 10% buckets: 0x01 = 0-10% used, 0x0A = 90-100%, 0x0B = exceeded.

LIFETIME VIA SYSFS (KERNEL 4.19+)

The MMC driver decodes EXT_CSD for you and surfaces the wear bytes in sysfs:

```
$ ls /sys/block/mmcblk0/device/
...  life_time  pre_eol_info  ...

$ cat /sys/block/mmcblk0/device/life_time
0x01 0x01

$ cat /sys/block/mmcblk0/device/pre_eol_info
0x01
```

- `life_time`: two bytes - TYP_A, TYP_B
- `0x01..0x0A` → 0-10%, 10-20%, ..., 90-100% used; `0x0B` = past EOL
- `pre_eol_info`: `0x01` Normal, `0x02` Warning, `0x03` Urgent

LIFETIME VIA `mmc-utils`

No sysfs entry? Ship `mmc-utils`. Same bytes, userspace tool:

```
$ mmc extcsd read /dev/mmcblk0
...
eMMC Life Time Estimation A [EXT_CSD_DEVICE_LIFE_TIME_EST_TYP_A]: 0x01
eMMC Life Time Estimation B [EXT_CSD_DEVICE_LIFE_TIME_EST_TYP_B]: 0x00
eMMC Pre EOL information    [EXT_CSD_PRE_EOL_INFO]:               0x01
```

- Issues `MMC_SEND_EXT_CSD` (CMD8) via the `MMC_IOC_CMD` ioctl
- Requires JEDEC `EXT_CSD_REV` ≥ 7 (eMMC 5.0, ~2014)
- `0x00` means "not implemented for this part" - common for TYP_B

WHY PER-PROCESS ATTRIBUTION

diskstats tells you: *"mmcblk0 wrote 42 GB today."*

It does not tell you: *"...and 38 GB of that was our log daemon retrying a broken upload."*

- Total bytes is a fleet-level alarm
- Per-process bytes is a fixable bug
- eBPF gets us there with negligible overhead

EBPF IN 30 SECONDS

- Tiny sandboxed programs that run *inside the kernel*, attached to hooks like tracepoints, kprobes, and the network stack
- Loaded from userspace, verified at load time (bounded loops, no wild pointers) so they can't crash the kernel
- Communicate with userspace via **maps**: shared key/value tables the kernel program writes and userspace reads
- No kernel module, no recompile, no reboot

EBPF IN 30 SECONDS

For our use case: hook the kernel's writeback path, increment a per-process counter, drain it from userspace every few seconds.

WHERE TO HOOK

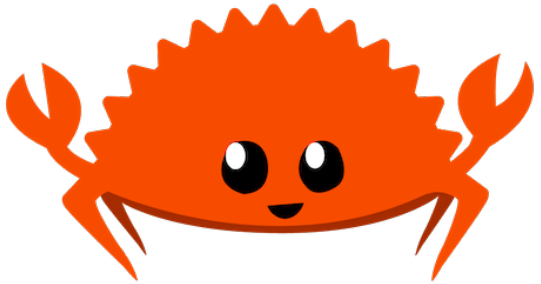
We need **pid** + **device**, surviving short-lived processes.

- **Block layer:** too late, `current` is a kworker
- **`/proc/pid/io`:** one scalar across every device, misses short-lived tasks
- **Page cache dirty:** writer is still `current`, folio carries bdi + inode

We count dirties, not flash bytes. Truncates inflate, repeated overwrites deflate, `O_DIRECT` bypasses. Good enough to *rank*.

WHERE TO HOOK

One hook, in the writer's context, before the kworker takes over.



RUST + AYA = ♥

EBPF: TRACKING WRITES

```
// writeback_dirty_page became writeback_dirty_folio in kernel 5.16.
// Userspace tries the old name first, falls back to the new one.

SEC("tracepoint/writeback/writeback_dirty_page")
int handle_writeback_dirty_page(struct trace_event_raw_writeback_dirty_page *ctx) {
    struct disk_write_key key = {};
    key.tgid = bpf_get_current_pid_tgid() >> 32;
    bpf_core_read(key.dev_name, sizeof(key.dev_name), &ctx->name);
    if (key.dev_name[0] == '\0') return 0;
    update_map_u64(&DISK_IO_WRITE_STATS, &key, PAGE_SIZE);
    return 0;
}
```

One event = one 4 KB page dirtied. Userspace attaches whichever tracepoint the kernel exposes.

EBPF: ATTACHING FROM USERSPACE (RUST + AYA)

```rust
// memfaultd/src/ebpf/disk_io.rs

let mut ebpf = Ebpf::load(DISK_IO)?;

// Try the pre-5.16 name first, fall back to the folio variant
let page_attached = {
    let prog: &mut TracePoint = ebpf
        .program_mut("handle_writeback_dirty_page")?.try_into()?;
    prog.load()?;
    prog.attach("writeback", "writeback_dirty_page").is_ok()
};
if !page_attached {
    let prog: &mut TracePoint = ebpf
        .program_mut("handle_writeback_dirty_folio")?.try_into()?;
    prog.load()?;
    prog.attach("writeback", "writeback_dirty_folio")?;
}
```

EBPF: DRAINING THE MAP

```rust
pub async fn run_once(&mut self) -> Result<()> {
    for key in self.write_stats.keys().filter_map(Result::ok) {
        // Sum across all CPUs (per-CPU hash map)
        let total: u64 = self.write_stats.get(&key, 0)?.iter().sum();
        let _ = self.write_stats.remove(&key);
        if total == 0 { continue; }

        let proc_name = self.proc_name_cache.get_or_resolve::<P>(key.tgid)?;
        let dev_name  = key.dev_name_as_str();

        readings.push(build_write_metric(&dev_name, &proc_name, total));
    }
    self.metrics_mbox.send_and_forget(readings)?;
}
```

Emits: `diskstats/mmcblk0/<proc>/pages_dirtied`

ONE HOUR OF `wefault-disk-killer`

```
#!/bin/sh

INPUT="${STORAGE_DIR}/treasure_island.epub"
OUTPUT="${STORAGE_DIR}/treasure_island2.epub"

# Grab the ebook from Project Gutenberg
curl https://www.gutenberg.org/cache/epub/120/pg120-images-3.epub \
     -o "${INPUT}"

# Copy it on top of itself, forever, fsync'd to NAND
while true; do
  dd if="${INPUT}" of="${OUTPUT}" conv=fsync
  sleep 30
done
```

49 MB epub. One `curl` at startup, then a `dd` loop every 30 s with `conv=fsync` so each copy actually lands on flash.

ONE HOUR OF `wefault-disk-killer`

Workload: `dd` overwrites a 49 MB epub every 30 s with `conv=fsync`, plus a one-shot `curl` download at boot.

source	metric	1 h value
diskstats	`mmcblk0` bytes_written	~5.93 GB
eBPF	`dd` pages_dirtied	~1.45 M

diskstats says the device is being pounded. eBPF names `dd` as the source. Projected forward: ~142 GB/day, ~4.3 TB/month, all from one 49 MB file in a bash loop.

THE FULL MONITORING PICTURE

- **Device-level**: `lifetime_remaining_pct` from `EXT_CSD`. "Is this device dying?"
- **Fleet-level**: `bytes_written` from `/proc/diskstats`. "How hard is it being worked?"
- **Process-level**: `pages_dirtied` from eBPF. "Who is doing this?"

All three are needed. Each one answers a different question.

PRACTICAL RECOMMENDATIONS

WORKLOAD

- Batch small writes. Let the page cache do its job
- Append, don't rewrite-in-place
- Audit your loggers. They are usually the worst offender
- Move volatile state (caches, scratch) to `tmpfs` when you can

IN PRODUCTION

- Ship `lifetime_remaining_pct` as per device telemetry. Monitor percentage of fleet in bad states
- Track `bytes_written` across whole fleet. Watch for spikes after releases
- Run eBPF per-process attribution continuously. The noisy process is named in the metric, no repro needed

TAKEAWAYS

1. Flash wears probabilistically. Every P/E cycle does a little damage
2. The controller hides most of this, but exposes EXT_CSD over JEDEC
3. diskstats + EXT_CSD + eBPF gives you the full picture: *what*, *how much*, *who*
4. Catch wear in telemetry, not in the field

THANKS!

QUESTIONS?

More writing at:
`interrupt.memfault.com/authors/blake`