A bit is not a counter

Measuring how fast memory changes, when the kernel will only tell you whether it changed. A page-frequency instrument, its validation against known workloads, and the negative result that separates it from symbol profiling.

Matthew J. Harmon

2026-09-11

Machine speed is the one thing automation cannot hide. An agent can vary its timing, randomise its ordering, and spread its work across processes, but when it does something a million times a second, that rate is the work itself. If you want an instrument that notices automation without knowing in advance what the automation is, rate is a good place to point it.

This is an account of building one: a profiler that finds memory changing faster than the rest of its process, and the code responsible. It is also an account of three ways the obvious design is wrong, each of which produced confident, well-formatted, entirely meaningless output before it was caught. The negative result at the end is the most useful part.

All measurements below are from a single machine — AMD, kernel 7.0.0, `perf_event_paranoid=4`, `yama/ptrace_scope=1` — and every number is measured rather than estimated.

The problem with asking the kernel

Linux offers a cheap way to learn which pages a process has written. Write `4` to `/proc/PID/clear_refs` and the kernel clears a per-page *soft-dirty* bit and write-protects the pages; the next write to each page faults, sets the bit, and proceeds. Read `/proc/PID/pagemap` and bit 55 of each 8-byte entry tells you whether that page was written since the reset.

It is almost free. It costs no memory reads at all, so the observer barely perturbs the observed — which matters when the thing you are looking for is defined by its speed.

The obvious instrument writes itself. Clear the bits, wait, read the bitmap, count. Sample a process every 50 ms for eight seconds and rank pages by how often they came back dirty.

Here is what that produced against a synthetic target with a deliberately planted hot region:

```
pages written: 84   median hits/page: 156   MAD: 1

          address   rate   hits       z  kind       region
        0xb49000   100%    156     0.0  file-data  rw-p python3.12
        0xb8a000   100%    156     0.0  file-data  rw-p python3.12
       0x28073000  100%    156     0.0  heap       rw-p [heap]
    0x7bc9bf92c000 100%    156     0.0  anon       rw-p [anon]
```

Every page. One hundred percent. One hundred and fifty-six hits out of one hundred and fifty-six samples, a median of 156, a median absolute deviation of 1, and therefore a z-score of exactly 0.0 for every page in the process. The planted hot region is in that list, indistinguishable from the interpreter's own bookkeeping.

The failure is not a bug. **Soft-dirty is a bit, not a counter.** It records that a page *was* written since the last reset, never how many times. At a 50 ms window, any page touched at all comes back dirty, so a page written a hundred times a second and a page written ten million times a second produce identical output. The metric saturates, and saturation is invisible: the numbers look fine, the table sorts, the z-scores compute. Nothing about the output announces that the instrument has stopped measuring.

This is worth dwelling on, because it is the general shape of the problem. Binary-per-interval observations look like measurements and are not.

Sweeping the timescale

The information the bit loses within one window can be recovered by varying the window. For a page written at frequency f, sampled with interval T:

writer	probability the page is dirty
periodic	`f · T`, until it saturates at 1
randomly timed	`1 - exp(-f · T)`

Both say the same thing. The interval at which a page's hit rate *starts to fall off* is its write frequency. So sample the same process at 20 ms, 5 ms, 1 ms and 0.25 ms, and read the frequency off the rungs where it has not yet saturated.

A page still pinned at 100% in the shortest window is being written faster than the instrument can resolve. That is not a failure — it is the finding, reported as a floor rather than a number.

Frequency, unlike a hit rate, does not saturate, and it is comparable across processes and across machines.

Two details in the inversion turned out to matter more than they look.

Read the frequency off the largest unsaturated window, not the smallest. The shortest window has the fewest writes per sample and so the most discretization error. For a tier written at a true 200 Hz, the 0.25 ms rung read 6% and the 1 ms rung read 22%; the latter is the better-conditioned measurement.

Invert as periodic, not Poisson. Using `-ln(1-r)/T` on a periodic writer overestimates it by about a third — it read 268 Hz against a ground truth of 200. The linear form `r/T` is also the conservative choice: for a bursty writer it reads low, so a reported frequency is a floor rather than an inflated claim.

Sampling is not free, and the difference is not small

The sweep sleeps for the interval it wants. It also has to do work — a pagemap read, a `clear_refs` write — and that work is not instantaneous. Feeding the *requested* interval into

the frequency math scales every estimate by the error.

Measured on the synthetic target:

requested	actual	overhead
0.25 ms	0.36 ms	44%
1.00 ms	1.12 ms	12%
5.00 ms	5.19 ms	4%
20.0 ms	21.2 ms	6%

Correcting the math to use measured intervals moved a 200 Hz tier from 215 Hz to 196 Hz. For the portable collector described below, where a single sample costs tens of milliseconds, the same error would have been an order of magnitude.

The model can check itself

Below saturation, `r/T` should be the same at every rung. That is a testable claim about each page, and pages that fail it are not measuring a frequency.

A page whose hit rate stays *flat* as the window shrinks — 88%, 88%, 89%, 90% — is violating the model, and the instrument says so with a `?` rather than printing a confident number. In practice this catches a specific blind spot, described next.

Two collectors, one analysis

Soft-dirty is Linux-only. The analysis is not, so the collector sits behind a two-method interface — `reset()` and `sample(spans)` — with two implementations.

`softdirty` asks the kernel, as above. No memory reads, roughly 0.13 ms per sample on a small process.

`hashdiff` reads each page and hashes it, comparing against the previous sample. A page whose hash changed was written. This works anywhere memory can be read — `/proc/PID/mem` on Linux, `vm_read` on macOS, `ReadProcessMemory` on Windows — because the only platform-specific part is the read.

It has a real blind spot: a write that restores a previous value leaves the content unchanged, so the hash cannot see it. Soft-dirty sees the write regardless of the value. That blind spot is exactly what the model-fit check flags — in the validation run, six pages came back `?`, and all six were pages `softdirty` independently showed as saturated.

Two mechanisms that share only the analysis, disagreeing in a predictable direction, is a more useful arrangement than either alone.

Validation

A synthetic target writes four populations at **time-paced** frequencies. The pacing matters, and getting it wrong wasted a full cycle: an earlier version paced by loop count, and "every 50 iterations" in a Python loop turns out to be roughly 60 kHz. Nothing in that workload was slow, so nothing discriminated, and the tool looked broken when it was reporting correctly.

tier	pages	ground truth	`softdirty`	`hashdiff`
hot	4	as fast as the CPU allows	`> ceiling`	`> ceiling`
warm	8	200 Hz	**196 Hz**	**198 Hz**
cool	8	5 Hz	**5 Hz**	**5 Hz**
cold	2048	written once at startup	absent	absent

Both tiers were recovered as contiguous eight-page runs at the right addresses. The 2,048 cold pages, written once and then left alone, correctly never appear.

Two independent mechanisms converging on ground truth is the point. They share the inversion and the fit check; they share no part of the measurement.

What "normal" has to mean

Finding the fastest page in a process is not the same as finding one that is *faster than it should be*. The second needs a baseline, and a baseline for memory runs into a hard constraint immediately.

ASLR relocates every mapping on every run. The address `0x7d34c29b3000` in today's process is unrelated to the same number tomorrow. A profile cannot store addresses.

What survives a restart:

- **File-backed pages** have a stable identity. `libxul.so+0xa000` names the same data on every run, on every machine with that build.
- **Anonymous pages** have none. Heap and allocator arenas are laid out differently each time.

In a browser this bites hard. Of 1,170 written pages in a Firefox content process, **nine had stable identity**. Almost all write activity is anonymous, so anonymous memory cannot be treated as the leftover case.

The percentile trap

The first attempt scored each anonymous page against the 95th percentile of the baseline's pooled frequencies. It produced 29 findings on a process compared against its own baseline.

They were all false. The flagged pages were allocator arenas running at 24–27 Hz — entirely ordinary behaviour — sitting above a percentile dominated by roughly 1,600 near-idle pages. Comparing individuals against a pooled percentile flags the top 5% *by construction*. The test could not not fire.

The fix is to compare **curves rather than points**. Store each baseline run as its own sorted frequency curve, and judge the live capture's k-th fastest page against the fastest k-th page any baseline run produced. A process that is merely busier shifts the whole curve and is judged as such. A single injected hot page spikes one rank and stands out.

Same captures, same data: 29 false positives to **zero**.

A signal that survives the ceiling

There is one more comparison worth making, because it does not depend on frequency at all: **how many pages sit above the resolution ceiling**. Unlike a frequency, that count is not capped by the ceiling, so it keeps working on large processes where everything interesting saturates.

It was exactly 17 in both baseline runs of the synthetic target, and 18 with an injection present.

Detection, with a control

The test workload gained two pages written at 800 Hz — a small, fast region against a busy background, which is the shape of the thing worth finding.

run	novel pages	over baseline	ceiling population
control — clean vs its own baseline	0	0	17 (baseline 17–17)
detection — injected vs clean baseline	0	**1, at 14×**	**18 (outside 17–17)**

The injected region was caught at rank 8, 2,769 Hz against a 192 Hz baseline, and independently by the ceiling population. The control reported *nothing outside the baseline*.

Repeating the detection against a freshly started process — new PID, new ASLR layout, the hot region at a completely different address — reproduced both signals. The baseline is genuinely address-independent.

Profiling by library and function

Addresses are unstable and anonymous memory is anonymous, but **code has stable names**. A function sits at a fixed offset inside its library, so `libxul.so!AppendElement` is the same identity on every run.

Hardware sampling reports the instruction pointer and the data address in the *same sample*, which is enough to key a profile on which function touched which kind of region:

```
samples   per sec  region      perms  function
    810       202  [stack]     rw-p   python3.12![unknown]
    367        92  [stack]     rw-p   python3.12!_PyEval_EvalFrameDefault
    361        90  [anon]      rw-p   python3.12!_PyEval_EvalFrameDefault
```

An anomaly is then a pair the baseline has never seen — some function writing somewhere it has never written before. That is a far sharper statement than "page 0x7d34 is busy", and it is legible without a debugger.

Getting the samples required working around the tooling. On this hardware `perf mem record` cannot attach to a running process: it wraps AMD IBS as `ibs_op/ldlat=0/`, which is rejected for an existing PID. The raw event works — `perf record -e ibs_op// -d -p PID` — and the

`-d` is essential, because without it every sample carries no data address at all and `perf script -F addr` refuses outright. Stores are separated from loads afterwards, from the `data_src` field.

Kernel-context samples are dropped. Their data addresses are kernel addresses, absent from the process's maps, and not a property of the application being profiled. Leaving them in produced three spurious "never seen before" findings in the first control run.

Baseline stability is good: independent runs of the same workload produced 37 distinct function-to-region pairs, twice.

The negative result

The symbol profiler did not detect the injection.

This deserves to be stated plainly, because the two instruments look interchangeable and are not.

	injected `[anon]` store rate	baseline
measured	118/s	112/s, 121/s

Statistically identical. The control was clean and the detection was clean, and the second of those is a miss.

The reason is structural. **IBS samples store volume.** Eight hundred stores per second of injected activity is invisible against millions of interpreter stores per second — it does not shift the sampled distribution at all. Soft-dirty counts **pages touched**, and two extra hot pages are obvious against a working set of thirty-five. Worse for the symbol view, an interpreter funnels everything through one eval loop, so the function key carries almost no information about this workload.

So the two instruments answer different questions, and the difference is not cosmetic:

- **Page frequency** is sensitive to a *small region changing fast.* It is the right first instrument for machine-speed activity.
- **Symbol sampling** is sensitive to a *shift in where the bulk of memory traffic happens*, and to code appearing where it has no business being — a new library writing somewhere, injected code carrying its own symbol, any function writing into an executable mapping.

A detector that had only been tested on the case it catches would have shipped with the wrong claim attached.

They compose in one direction

The miss points at the fix. Sampling is proportional to traffic volume, so a low-volume region is invisible in a whole-process profile — but not in a profile narrowed to that region and sampled hard.

```
# 1. where - page frequency finds the anomaly
memwatch.py compare myapp cap.json
   -> 0x71feaa875000   2,798 Hz   baseline 192 Hz @rank 8
```

```
# 2. who - hardware sampling attributes it
memwatch-sym.py attribute $PID 0x71feaa875000+2 -d 15 --period 2000
   -> 8,690 stores   python3.12!_PyEval_EvalFrameDefault
```

Narrowed to two pages, 17,515 of 261,613 addressed samples landed inside the range — 6.7% of all sampled memory traffic, from a region that was undetectable in the whole-process view minutes earlier.

Find the *where* with the instrument that measures rate. Attribute the *who* with the instrument that measures volume. Neither substitutes for the other.

What it costs, and where it stops

The cheap path has a scaling property worth knowing before trusting its output. **`clear_refs` walks every page-table entry in the process**, so its cost tracks the whole address space rather than the part under investigation.

target	mapped pages	`clear_refs`	full pagemap read	usable ceiling
synthetic	7,309	0.06 ms	0.07 ms	**~2,800 Hz**
Firefox	~3,666,000	5.2 ms	38.0 ms	**~160 Hz**

Narrowing the sweep to candidate pages removes the 38 ms read entirely. It **cannot touch the 5.2 ms reset.** On a multi-gigabyte process, anything above roughly 160 Hz reads as "at ceiling", and going finer requires the hardware path — which is precisely where the ceiling-population signal earns its place.

On that same Firefox process the instrument still did its job: two anonymous pages pinned above the ceiling at a robust z-score of 108, standing clear of a ~25 Hz background of allocator and library churn. Saturated, but unambiguous.

The rest of the limits, stated once:

- Soft-dirty attributes a write to a page, never to code.
- `hashdiff` misses writes that restore a previous value, and costs a full read of the working set per sample. The `?` flag marks where this has happened.
- Hardware sampling is statistical. A rate from it is *samples* per second, not stores per second — comparable only against captures taken at the same sample period.
- Both tools observe a running process. Neither is a forensic dump analyser.
- `os.access()` is not a sufficient permission check for either path. It consults file permissions only, while `pagemap` additionally requires `PTRACE_MODE_READ` and `/proc/PID/mem` requires `PTRACE_MODE_ATTACH`, which `yama/ptrace_scope  1` denies for any process that is not a descendant. The tools probe by opening for real.
- **macOS and Windows collectors are not implemented.** The analysis is platform-independent and `hashdiff` is the shape they would take, but that code is not written, and neither platform has a soft-dirty equivalent — both would run at hashdiff cost with hashdiff's blind spot.

What generalises

Three things here are not about memory.

A binary observation per interval is not a rate. It looks like one, it tabulates like one, and it saturates without saying so. Wherever an instrument reports whether something happened during a window, the window is part of the measurement, and one window is not enough.

Sweeping a parameter recovers what a single setting destroys. The frequency was never in any individual sample; it was in how the answers changed as the window shrank.

A detector should be tested against the case it misses. The control run and the detection run together are the experiment — either alone supports a claim that the other refutes. The symbol profiler's clean control looked like success until the injected workload produced the same clean result.

For the adversarial case specifically, the most durable signal turned out to be the crudest one: not how fast a page changes, which the instrument can only measure up to its own ceiling, but **how many pages are changing faster than the instrument can follow**. That count is not capped, it is stable across runs of the same workload, and it moved the moment something started operating at a speed the baseline had never contained.