mergelog-rs: Rewriting a Year-2000 C Tool in Rust — and Making It 2.26× Faster

2026-03-15

A Rust rewrite of mergelog 4.5, a C tool from 2000 that merges Apache log files chronologically. Six optimizations — a k-way heap merge, a hand-rolled CLF parser, String buffer reuse, mimalloc, 4 MiB read buffers with SIMD newline search, and zstd support — took the Rust binary from slower than the original to 2.26× faster.

---

mergelog is a small but handy tool: it takes multiple Apache log files from servers behind a round-robin DNS and merges them into a single chronologically sorted stream. The original version 4.5 was written in C around 2000–2001, weighs in at ~450 lines, and does exactly what it promises — reliably and fast.

This post documents the journey from mergelog-4.5 to mergelog-rs: what changed, what was surprising, and how six targeted optimizations made the Rust binary **2.26× faster** than the C original.

Full respect and honour to **Bertrand Demiddelaer**, who created mergelog back in 2000. Writing a tool this fast, this correct, and this compact in C — with no external dependencies beyond zlib — is genuinely impressive. mergelog-rs stands on his shoulders.

The Original: mergelog 4.5

The C program reads any number of log files in *Common Log Format* (CLF) / *NCSA Combined Format*:

```
%h %l %u %t "%r" %>s %b "%{Referer}i" "%{User-agent}i"
```

Example line:

```
93.184.216.34 - - [15/Mar/2026:12:00:00 +0100] "GET /index.html HTTP/1.1"
200 1234 "-" "curl/7.88"
```

The algorithm in 4.5 is straightforward: find the earliest timestamp across all files, then iterate second-by-second from there to the end, writing all matching lines at each step. That is **O(timespan × K)** — slow when logs span multiple years or many files are involved.

Gzip support exists, but as a separate binary (`zmergelog`) compiled with `-DUSE_ZLIB`.

The Rust Rewrite: What Changed

1. Algorithm: K-Way Merge with a Binary Heap

Instead of time-based iteration, mergelog-rs uses a **k-way merge with a min-heap** (`std::collections::BinaryHeap` with reversed ordering):

- Seed the heap with the first line from each file
- Main loop: pop the smallest timestamp, write the line, push the next line from the same file
- Complexity: **O(N log K)** — independent of the time span of the logs

```
while let Some(Entry { mut line, mut reader, file_idx, .. }) = heap.pop()
{
    out.write_all(line.as_bytes())?;
    line.clear();
    if fill_line(&mut *reader, &mut line)? {
        let ts = parse_clf_timestamp(&line).unwrap_or_else(sentinel);
        heap.push(Entry { ts, file_idx, line, reader });
    }
}
```

2. Magic-Byte Detection Instead of File Extension

The original detects gzip by binary type (`zmergelog` vs. `mergelog`). mergelog-rs reads the first 6 bytes of each file and detects the format automatically:

Format	Magic Bytes
gzip	`1F 8B`
bzip2	`42 5A 68` (BZh)
xz	`FD 37 7A 58 5A 00`
zstd	`28 B5 2F FD`

```
pub fn detect_from_bytes(magic: &[u8]) -> Compression {
    if magic.len() >= 2 && magic[..2] == [0x1f, 0x8b] { Compression::Gz }
    else if magic.len() >= 3 && magic[..3] == [0x42, 0x5a, 0x68]
{ Compression::Bz2 }
    else if magic.len() >= 6 && magic[..6] == [0xfd, 0x37, 0x7a, 0x58,
0x5a, 0x00] { Compression::Xz }
    else if magic.len() >= 4 && magic[..4] == [0x28, 0xb5, 0x2f, 0xfd]
{ Compression::Zstd }
    else { Compression::None }
}
```

One binary, all formats, no file extension required. The same peek-based detection also works for stdin — `BufReader::fill_buf()` inspects the first bytes without consuming them, so no seek is needed.

3. Timezone-Aware Timestamp Comparison

The C original ignores the timezone offset in timestamps — it converts date components to local time via `mktime()`. For logs from servers in different timezones this can produce incorrectly ordered output.

mergelog-rs converts every timestamp to UTC before comparison. A `+0900` timestamp and a `-0500` timestamp are ordered correctly, regardless of the system's local timezone.

Profiling: Where Does the Time Go?

Since `perf` was locked down on the system (`perf_event_paranoid = 4`), a dedicated timing binary was built that measures each phase manually with `std::time::Instant`:

```
─── profile_timing (41.65M lines) ──────────────────
  I/O   (read_line)       3.91 s   32.4%
  Parse (jiff strptime)   5.52 s   45.7%  ← bottleneck
  Heap  (push/pop)        1.97 s   16.3%
  Write (write_all)       0.69 s    5.7%
  ──────────────────────────────────────────────────
  Σ                      12.08 s
```

Nearly half the time was in the parser. The reason: `jiff::Zoned::strptime` is a full parsing pipeline — format-string interpretation, timezone database lookup, error handling, heap allocation. Across ~42 million lines that adds up fast.

This full parsing pipeline is super handy, but in our use case it is not necessary — therefore we built a hand-rolled CLF timestamp parser.

Optimization 1: Hand-Rolled CLF Parser

The CLF timestamp format is completely fixed:

```
[DD/Mon/YYYY:HH:MM:SS ±HHMM]
 0123456789012345678901234 5
```

A hand-written parser only needs to: 1. Find `[` in the string 2. Read 6 integers from fixed byte positions 3. Compute the Unix timestamp via integer arithmetic

For step 3 we use Howard Hinnant's proleptic Gregorian algorithm — no branches, pure multiplication:

```rust
fn days_since_epoch(y: i64, m: i64, d: i64) -> i64 {
    let y = if m <= 2 { y - 1 } else { y };
    let era = if y >= 0 { y } else { y - 399 } / 400;
    let yoe = y - era * 400;
    let doy = (153 * (m + if m > 2 { -3 } else { 9 }) + 2) / 5 + d - 1;
    let doe = yoe * 365 + yoe / 4 - yoe / 100 + doy;
```

```
    era * 146_097 + doe - 719_468
}
```

Result: parse time dropped from 5.52 s to 1.50 s — **3.7× faster**.

Optimization 2: Reuse the String Buffer

The original implementation allocated a new `String` for every line read:

```
// before: fresh allocation per line
fn read_line(reader: &mut dyn BufRead) -> Result<Option<String>> {
    let mut line = String::new();  // ← new every time
    reader.read_line(&mut line)?;
    ...
}
```

The fix is simple: when an `Entry` is popped from the heap, you receive `line: String` with full ownership. Instead of dropping it, clear it and read the next line directly into the same buffer:

```
// after: capacity is retained
while let Some(Entry { mut line, mut reader, file_idx, .. }) = heap.pop()
{
    out.write_all(line.as_bytes())?;
    line.clear();  // ← keeps the heap allocation
    if fill_line(&mut *reader, &mut line)? {
        let ts = parse_clf_timestamp(&line).unwrap_or_else(sentinel);
        heap.push(Entry { ts, file_idx, line, reader });
    }
}
```

Since consecutive log lines are similar in length, the buffer stops growing after the first few lines — no `malloc` calls in the hot path.

Optimization 3: mimalloc as the Global Allocator

With most per-line allocations already eliminated, there are still smaller allocations happening — buffer growth during seeding, internal `BinaryHeap` resizing, and the occasional `String` capacity bump. Swapping out the default glibc allocator for mimalloc is a one-liner in Rust:

```
# Cargo.toml
mimalloc = { version = "0.1", default-features = false }
```

```
// main.rs
use mimalloc::MiMalloc;
#[global_allocator]
static GLOBAL: MiMalloc = MiMalloc;
```

That's it — no other code changes. mimalloc is designed for throughput-heavy workloads with many small, short-lived allocations, which matches our remaining allocation pattern well.

Result: 5.55 s → 5.39 s — a further ~3% improvement.

Not a dramatic win on its own, but it's essentially free: two lines of code, no logic changes, and it stacks on top of everything else.

Optimization 4: 4 MiB Read Buffers + SIMD Newline Search

After the previous three optimizations, profiling showed I/O was still the largest remaining cost at 47% of total time. Two changes address this together.

4 MiB read buffers — The previous `BufReader` used a 256 KiB internal buffer, meaning a syscall every ~256 KiB of data. Increasing to 4 MiB reduces syscall frequency by 16×, which shows up directly in system time.

SIMD newline search via `memchr` — Instead of `BufReader::read_line`, the line reader now uses `fill_buf()` + `memchr::memchr()` directly. The `memchr` crate uses AVX2/SSE2 to scan 16 or 32 bytes per cycle when searching for `\n`:

```
fn fill_line(reader: &mut dyn BufRead, buf: &mut String) -> Result<bool>
{
    loop {
        let available = reader.fill_buf()?;
        if available.is_empty() { return Ok(false); }
        match memchr(b'\n', available) {
            Some(pos) => {
                buf.push_str(std::str::from_utf8(&available[..pos +
1])?);
                reader.consume(pos + 1);
                return Ok(true);
            }
            None => {
                buf.push_str(std::str::from_utf8(available)?);
                let len = available.len();
                reader.consume(len);
            }
        }
    }
}
```

Result: 5.39 s → 4.43 s — **another 18% improvement**, bringing the total to **2.26× faster** than the C original.

A Road Not Taken: Parallel Reader Threads

An obvious next step seemed to be spawning one thread per input file — each thread handles its own I/O and decompression, sending `(timestamp, line)` pairs to the merge thread via a bounded `mpsc` channel:

```
Thread 1: read + decompress + parse → channel ─┐
Thread 2: read + decompress + parse → channel ─┤→ Merge thread → stdout
...                                            ┘
```

For **compressed files** (bz2, xz) this is conceptually attractive: decompression is CPU-bound and can genuinely run in parallel.

For **plain-text files** on a single disk, it does not help — the bottleneck is disk read bandwidth, and parallel reads from the same physical device cause seek contention or simply saturate the same I/O bus. Benchmarking confirmed this: the threaded version achieved identical wall-clock time on plain-text files, while adding per-line heap allocation overhead (the channel requires sending owned `String` values, making buffer reuse impossible across the thread boundary). The result was slightly *slower* than the sequential version with large buffers.

Takeaway: threading is the right tool when the bottleneck is CPU-bound work that can be parallelised per-file (compressed files on a multi-core system). It is the wrong tool when the bottleneck is a shared I/O resource. Measure first.

Final Results

Measured on 7 log files × 1 GiB = 7 GiB total, 41.65 million lines:

Version	Wall time	User	System
mergelog-4.5 (C)	10.03 s	5.93 s	4.12 s
mergelog-rs v1 (jiff)	10.39 s	9.13 s	1.25 s
1. hand-rolled parser	5.99 s	4.75 s	1.23 s
1. buffer reuse	5.55 s	4.29 s	1.26 s
1. mimalloc	5.39 s	4.17 s	1.22 s
+ 4 MiB buffers + memchr	**4.43 s**	**3.61 s**	**0.81 s**

```
mergelog-rs ran 2.26 ± 0.02 times faster than mergelog-4.5
```

Memory and CPU Usage

Measured with `/usr/bin/time -v` on the same 7 × 1 GiB workload:

	mergelog-4.5	mergelog-rs
Wall time	9.63 s	4.28 s
User time	5.61 s	3.42 s
System time	4.01 s	0.84 s
CPU utilisation	99%	99%
Peak RSS	**1.6 MiB**	**30.7 MiB**
Involuntary ctx switches	105	122

Memory — The picture changed significantly with the 4 MiB read buffers. mergelog-4.5 uses 1.6 MiB; mergelog-rs now uses 30.7 MiB. The breakdown is straightforward: 7 files × 4 MiB read buffer = 28 MiB, plus the write `BufWriter`, the binary itself, and mimalloc metadata. This is a deliberate trade-off — RAM is cheap, syscalls are not. At 30 MiB to process 7 GiB of data the ratio is still extremely lean, and the buffer size is a single constant in `reader.rs` that can be tuned if memory is constrained.

System time — The most striking difference is kernel time: mergelog-4.5 spends 4.01 s in the kernel versus 0.84 s for mergelog-rs. Two factors: the C version calls `write(1, ...)` directly for every line (~42 million syscalls), and its read buffers are only 32 KiB. mergelog-rs batches writes via `BufWriter` and reads via 4 MiB `BufReader`, cutting kernel round-trips dramatically.

CPU — Both tools are single-threaded and max out one core at 99%.

Takeaways

Five lessons from this project:

1. Measure before optimizing. Without the timing binary the focus might have landed on the heap. But `jiff::strptime` was costing nearly 3× more than every other phase combined.

2. Libraries are great until they aren't. `jiff` is an excellent crate for correct timezone-aware date handling. But when you parse the same fixed-format string 42 million times, you pay a steep price for generality. A 30-line hand-rolled parser that exploits the known fixed layout beats it by 3.7×.

3. Sometimes the allocator is the last few percent. After eliminating almost all allocations from the hot path, swapping in mimalloc took two lines of code and gave another 3% for free. It won't rescue a poorly written program, but on an already optimized one it's a cheap final step worth trying.

4. Buffer size matters more than you think. Going from 256 KiB to 4 MiB read buffers — a one-line change — cut system time by 34% and wall time by 18%. Every syscall is a round-trip to the kernel; fewer is better.

5. Parallelism requires the right bottleneck. Spawning reader threads per file is conceptually appealing for compressed logs where decompression is CPU-bound. But for plain-text files on a single disk, the bottleneck is I/O bandwidth — adding threads just adds channel overhead and per-line heap allocation, making things slightly *slower*. Threading is a tool, not a universal solution.

6. Ownership is not a burden. Buffer reuse in C requires careful explicit lifetime management and is easy to get wrong. In Rust, the ownership system makes it the obvious and safe solution — `line.clear()` instead of `free(line); line = malloc(...)`, with the compiler guaranteeing no use-after-free.

The code is available under GPL-3.0-or-later. The original mergelog 4.5 was released as GPL-2.0-or-later, which explicitly permits upgrading to any later version of the GPL. GPL-3.0 adds patent protection clauses and anti-tivoization provisions that GPL-2.0 lacks — a straightforward upgrade with no downsides for a command-line tool like this.