Moving WordPress Plugin Compute to Rust FFI: An Experience Report on Sitemaps, Related Posts, and Redirects

Jermaine Harris
Hutts Enterprises / Ferrus Labs
2026-09-20 · CC BY 4.0
Keywords: cs.SE, cs.PF, WordPress, Rust, PHP FFI
Source git 6ef9c2d · technical report

Abstract

Hutts WordPress plugins can call Rust shared libraries through PHP FFI and still run a PHP fallback when the .so is missing. That split is an operational convenience. It is not, by itself, a performance result. This report times three kernels that already exist in hutts-wp-rust-plugins: XML sitemap generation, related-posts scoring, and redirect lookup.

On one workstation (Intel Core i5-12600KF, PHP 8.3.6 with FFI enabled, rustc 1.97.1) we time a WordPress-free PHP fallback copied from the plugins, the same work via PHP FFI into libhutts_*.so, and the native Rust APIs. Workloads are synthetic: 10,000 sitemap URLs; 2,000 related-post candidates; 5,000 redirect rules and 1,668 lookup paths. Each cell is 2 warmups plus 12 timed repetitions.

Native Rust builds sitemap XML faster than the PHP loop (0.74 ms vs 1.74 ms mean). The PHP FFI path is slower than PHP because it JSON-encodes the URL list on every call. Related-posts PHP is an order of magnitude faster than Rust because it implements a weaker token-containment ranker, not mixed token and character n-gram cosine. Redirect lookup is the intended win: a radix trie in Rust is about 700x faster than a linear PHP scan on this fixture (0.22 ms vs 157 ms). FFI still beats PHP for redirects because the trie is built once.

We do not claim a new ranking algorithm, and we did not cut over production WordPress. The practical lesson is that moving plugin compute to Rust is an API and safety project first. Speed shows up only when the data structure changes and marshalling stays off the hot path.

1. Introduction

WordPress plugins in this fleet stay hook-compatible. hutts-rust-loader.php locates libhutts_*.so, caches FFI::cdef, and lets each plugin fall back to PHP when the extension, the library, or a symbol is missing. Operators can ship a PHP-only site and later drop in a release .so without changing hooks.

That story is easy to oversell as “Rust is faster.” Three things get mixed together:

- Language and optimizer (PHP bytecode vs --release Rust).
- Algorithm (linear scan vs radix trie; token count vs cosine).
- Boundary cost (JSON in, char* out, FFI::string, free).

This report separates those for three kernels that already exist as companions, not toy

ports.

Research question. On the same hardware and the same synthetic corpus, how do wall time and error behavior change for sitemap generation, related-posts ranking, and redirect lookup when the PHP companion runs alone versus Rust (FFI or native)?

Contributions:

- C1. A WordPress-free harness (harness/wp-rust-ffi/) that never opens MySQL or WP_Query.
- C2. Paired timings for three kernels and three implementations, including population standard deviation (Table 1).
- C3. An explicit finding that “Rust FFI” can lose to PHP when the PHP path is a cheaper algorithm or when JSON/FFI overhead dominates.

2. Operator context

The plugins under test are production-shaped, not extracted from a blog post. Sitemap generation is hooked on init and answers /sitemap.xml from published posts. Related posts score a query against candidate titles, bodies, and category IDs. Redirect lookup sits in front of request routing: every miss in a linear PHP scan walks the whole rule list.

We kept WordPress out of the timed path on purpose. A full request includes object cache, permalinks, and PHP-FPM bootstrap. Those costs dwarf a 1 ms XML builder and would hide the kernel comparison. The numbers here are lower bounds on kernel work, not end-to-end TTFB.

Production .so files were not replaced. Date-randomizer and other plugins that mutate post_date were not part of this campaign.

3. Architecture of the FFI boundary

Each crate is a cdylib. PHP passes a UTF-8 C string. Rust deserializes JSON, runs the kernel, and returns a heap char* that PHP copies with FFI::string and frees. Sitemap and related posts are stateless per call: the entire payload is re-parsed every time. Redirects expose an engine:

- hutts_redirect_engine_new(rules_json) builds a radix trie once.
- hutts_redirect_lookup_eng(engine, path) walks that trie.
- hutts_redirect_engine_free drops it.

The harness mirrors production: sitemap and related FFI re-encode JSON inside the timed loop; redirect FFI constructs the engine once (outside the inner lookup loop, still inside the timed repetition so construction is amortized across 1,668 paths, not across repetitions). Native Rust deserializes fixtures once before timing, then calls urlset, score_candidates, or RedirectEngine::lookup in-process. That is why native sitemap is not an apples-to-apples FFI comparison: it omits JSON parse and PHP request bootstrap.

An older .so on disk lacked hutts_redirect_engine_new. Release libraries were rebuilt from git SHA b2e08abc8f192719fbb302a46a13476879304ce3 so symbols match the source used in this report.

4. System under test

Repository hutts-wp-rust-plugins at that SHA. Kernels:

Kernel	Rust API	PHP fallback	Hot path
Sitemap	urlset / hutts_sitemap_generate	PHP XML loop	Build urlset
Related	score_candidates (cosine mix)	token containment	Rank 2k, keep top 5
Redirect	RedirectEngine trie	linear scan	1,668 lookups / 5k rules

Related-posts Rust mixes scores as 0.45 token_cosine + 0.55 char_ngram_cosine, then prefers category overlap. When a query title is present, title similarity is weighted 0.75 against 0.25 body. PHP counts query tokens longer than two characters that appear as substrings, then sorts by category overlap and that integer score. The PHP path is the correct compatibility fallback, not an optimized twin of the Rust ranker.

Sitemap PHP omits <lastmod> nodes that Rust emits when lastmod is present on the fixture. Output equality is therefore not claimed; both emit a well-formed urlset.

5. Method

Hardware. 12th Gen Intel Core i5-12600KF, 64 GiB RAM, Linux 6.17, 16 hardware threads. Operator box, not PHP-FPM on the WordPress origin.

Software. PHP 8.3.6 NTS, ffi.enable=true; rustc 1.97.1; --release native binary wp-rust-ffi-bench.

Workloads (generated, not production posts).

- Sitemap: 10,000 {loc, lastmod} objects (https://example.test/post-i).
- Related: one query (“honda civic brake pads maintenance”) plus 2,000 candidates cycling five lorem-like phrases; ngram=3, top_k=5, min_score=0.12.
- Redirect: 5,000 rules (/old-i -> /new-i; every 17th is prefix) and 1,668 lookup paths (every third old path, plus one miss and one prefix-style extra).

Protocol. Two warmup iterations discarded, then 12 timed repetitions. Wall time via hrtime(true) in PHP and std::time::Instant in Rust. Mean and population standard deviation (divide by n, not n−1). PHP fallbacks are WP-free copies of plugin loops (esc_url becomes htmlspecialchars with ENT_XML1).

What we did not do. Production plugin cutover; date-randomizer; nDCG or human quality evaluation of related posts; sampling RSS per kernel (runs are short and CPU-bound).

6. Results

Table 1. Mean wall time in milliseconds, n=12, population stdev. Source: results/01/kernel_times.csv.

Kernel	PHP fallback	PHP FFI	Rust native
Sitemap (10k URLs)	1.740 ± 0.123	3.319 ± 0.752	0.739 ± 0.065
Related (2k candidates)	2.762 ± 0.021	68.484 ± 1.325	65.644 ± 2.272
Redirect (5k rules, 1.7k lookups)	156.897 ± 6.057	0.648 ± 0.072	0.222 ± 0.009

No timed iteration returned an empty error document. Redirect FFI and native hit the trie; PHP scanned the rule list.

Table 2. Ratios on means (PHP fallback / other). Values below 1 mean PHP was faster.

Kernel	PHP / FFI	PHP / native
Sitemap	0.52	2.35
Related	0.040	0.042
Redirect	242	708

Sitemap FFI is noisier (stdev 0.75 ms on a 3.3 ms mean) than native XML (0.065 ms on 0.74 ms), consistent with allocator and JSON work. Redirect native is tight (0.009 ms stdev on 0.222 ms). Related native and FFI are within a few milliseconds of each other: once JSON is parsed, scoring dominates.

7. Interpretation by kernel

Sitemap. Native Rust wins a modest XML-builder comparison. FFI loses because the timed path JSON-encodes 10,000 URLs, crosses into Rust, deserializes, builds XML, and copies the result back. Operators who call FFI per request with large JSON payloads should not expect a win over a tight PHP loop. Chunking sitemaps (the plugin already has page / per_page) would shrink that marshalling tax; we did not time chunked FFI separately.

Related posts. These numbers are not a language comparison. Rust computes mixed cosine similarity over tokens and character 3-grams for every candidate. PHP counts a handful of query tokens. The PHP path is cheaper because it does less ranking work. A speed paper that ignored this would be misleading. If the product requirement is the Rust ranker, compare FFI to native, not to PHP: FFI is only about 4% slower than native here (68.5 ms vs 65.6 ms), so marshalling is not the story.

Redirects. This matches the folklore: O(path) trie versus O(rules) scan. A worst-case linear scan of 5,000 rules on 1,668 paths is about 8.3 million string comparisons. Mean PHP time 157 ms implies roughly 50 ns per comparison on this CPU, which is plausible for str_starts_with on short paths. The trie walks bytes of the path; 1,668 lookups at 0.22 ms is about 130 ns per lookup, independent of rule count except for cache effects. FFI still beats PHP because lookup does not re-encode 5,000 rules each time. The remaining 0.65 vs 0.22 ms gap is FFI string copies and PHP loop overhead, not the trie.

8. Worked example (redirect)

Let R = 5,000 rules and P = 1,668 paths. PHP does, on a miss, a scan of R comparisons per path. Prefix rules (i % 17 == 0) can hit early; exact hits on /old-i also stop the inner loop. Even with early exits, the measured 157 ms is two orders of magnitude above the trie. That

is why C3 (algorithm) dominates C2 (language) for redirects, while sitemap shows only a 2.35x XML-builder difference and related posts invert if you naively compare languages.

9. Discussion and threats to validity

The right question for a new kernel is: will the PHP fallback stay a weaker twin, and will FFI see a large JSON blob every request? If yes, measure that pairing. Do not publish a “Rust FFI speedup” from native in-process times.

Threats. (1) Synthetic URLs and lorem-like candidate text; production post HTML and permalinks may change constants, not the ranking of implementations. (2) Native Rust omits PHP request bootstrap and JSON parse (except that FFI includes JSON). (3) Single machine, no production PHP-FPM, no opcache warmup of a full WordPress stack. (4) PHP sitemap fallback omits lastmod nodes that Rust emits. (5) Related-posts quality is unevaluated; faster PHP may return worse neighbors. (6) Redirect prefix density (1/17) is synthetic. (7) Short CPU-bound runs were not RSS-sampled.

10. Related work

PHP FFI is documented in the PHP manual. WordPress plugin I/O remains PHP; hooks, options, and $wpdb are not replaced. The sitemaps protocol is independent of language. We do not cite unpublished “Rust is fast” blogs as evidence.

Closest engineering neighbors are other embed paths: PHP extension modules, HHVM-era JIT folklore, and rust-php / ext-php-rs style bindings. Those change the boundary (Zend API vs FFI::cdef). This report stays on the FFI+JSON ABI the fleet actually ships, because that is what operators enable with ffi.enable=true without compiling a Zend extension.

11. Ethics and operational safety

Fixtures never touched the WordPress fleet. No post_date mutation. Production .so files were not replaced. The harness refuses to boot WordPress; there is no database DSN in run_bench.py.

12. Conclusion

Rust helped where the data structure changed (redirect trie) and helped modestly for sitemap XML when marshalling was out of the timed path. It did not help related-posts latency versus a weaker PHP fallback. Moving plugin compute to Rust is an API and safety project first; speed requires paired algorithms and attention to FFI JSON.

For this fleet: keep the PHP fallbacks; rebuild cdylibs from a pinned SHA before timing; prefer a long-lived redirect engine over per-request JSON of the full rule set; do not expect sitemap FFI to beat PHP until the URL catalog is not re-encoded on every call.

Appendix A. Artifact

```
python3 harness/wp-rust-ffi/run_bench.py
```

Requires PHP FFI, libhutts_{sitemap,related_posts,redirect_manager}.so in the plugins

target/release, and cargo build --release -p wp-rust-ffi-bench. CSV SHA is the git blob of results/01/kernel_times.csv in this repository. The ZIP supplement beside this PDF includes the harness, fixtures, results, and this source.

References

- PHP FFI. https://www.php.net/manual/en/book.ffi.php
- Sitemaps XML format. https://www.sitemaps.org/protocol.html
- WordPress Plugin Handbook. https://developer.wordpress.org/plugins/
- hutts-wp-rust-plugins (pinned). git SHA b2e08abc8f192719fbb302a46a13476879304ce3