JSON parsing in OxCaml: fast, but not too fast

An experience report on building a simdjson-style parser with OxCaml's performance features

ARTEM PIANYKH*

We built a simdjson-style parser in OxCaml [1]. Its main API returns an owned, pattern-matchable `Json.t`. A lower-level borrowed tape provides a comparison with simdjson and isolates the cost of materializing the tree.

On simdjson's benchmark corpus, the owned parser reaches about 730 MB/s—five times the throughput of existing OCaml parsers—and the tape reaches 1.2 GB/s. Numeric variant blocks account for much of that difference. User-defined unboxed variants could let the owned API approach tape throughput.

SIMD vectors, unboxed types, and bit-manipulation intrinsics allowed to express the algorithm directly in OxCaml. Making it fast still required profiling, benchmarking, and reading disassembly. CPU-level techniques transferred from C++. Allocation strategies had to fit OCaml's GC. Some combinations of the new features also needed extensive annotations or library workarounds. The result was not C++ in OCaml syntax; it was a faster OCaml program with OCaml-shaped tradeoffs.

The source code, benchmarks, and investigation notes are available in the project repository [6].

1 A simdjson primer

Simdjson separates byte classification from grammar parsing (Figure 1). Its scanner produces a structural index. A top-down parser follows that index and writes the parsed document to a flat tape. The library's DOM API is a navigation layer over the tape rather than a separately allocated tree.

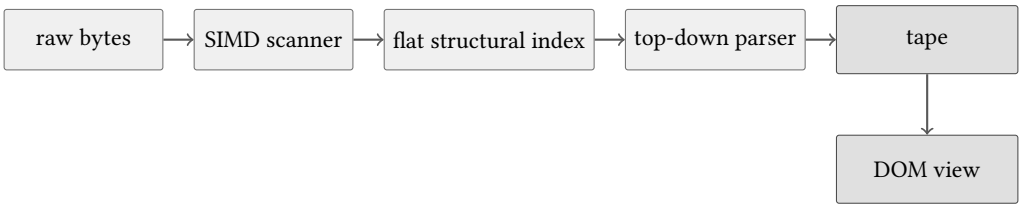

Fig. 1. The simdjson architecture. The DOM API navigates a tape stored in memory owned by the parser.

1.1 Scanner and structural index

The scanner processes 64-byte blocks. It tracks quotes and escapes, validates UTF-8, and classifies punctuation, whitespace, and the first byte of each number or literal. It uses bit masks to track strings and vector table lookups to classify bytes.

The output is a *structural index*: byte positions at which parsing decisions can occur. In Figure 2, it marks punctuation, the opening key quote, and the starts of `12` and `true`. It does not parse either atom. It lets the parser skip irrelevant positions.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
{ " x " : [ 1 2 , t r u e ] }
```

Fig. 2. The scanner marks the positions where parsing decisions can occur. Dark red characters are structural or pseudo-structural. They are not parsed values.

C++ simdjson expands those masks into a flat array of 32-bit byte positions. The parser can then jump directly from one relevant position to the next.

*August 2026. GitHub: https://github.com/artempyanykh. LinkedIn: https://www.linkedin.com/in/artempianykh.

1.2 Parser and tape

The second stage is a top-down parser over the structural stream. It recurses into containers and dispatches quotes and atoms to string, number, or literal routines. SIMD remains useful in string parsing. Grammar and numbers are mostly scalar.

The tape stores parsed entries in document order. An opening container points to the first entry after its scope, and its closing entry points back to the opening entry (Figure 3). Decoded strings live in a side byte buffer. These links let a DOM handle skip a complete value or iterate over a container without allocating a tree node for each value.

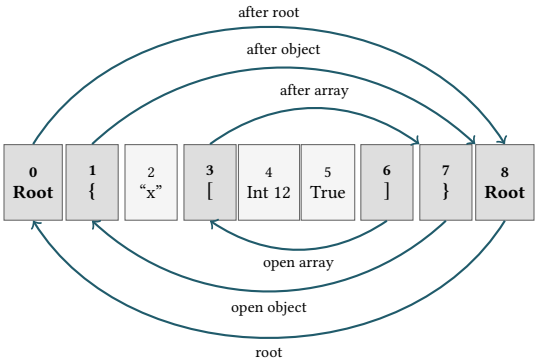

Fig. 3. Tape entries for `{"x":[12,true]}`. Opening entries point to the first entry after their scope. Closing entries point back to the matching opening entry. Payload packing is simplified.

Simdjson's DOM values are lightweight handles into the tape. The parser owns and reuses that storage, so parsing another document with the same parser invalidates the previous document and any values obtained from it [4].

```
simdjson::dom::parser parser;
auto doc = parser.parse(input).value();
auto name = doc["name"];

auto next = parser.parse(next_input).value();
// doc and name are now invalid
```

Simdjson benchmarks one warmed parser that reuses these buffers.

2 The simdjson-oxcaml parser

2.1 APIs and result

We applied this architecture in OxCaml, but made the main result an owned algebraic data type:

```
module Json : sig
  type t =
    | Null
    | Bool of bool
    | Int of int
    | Float of float#
    | String of string
    | Object of assoc array
```

```
    | Array of t array
  and assoc = { key : string; value : t }
end
```

The returned `Json.t` is independent of the parser: code can pattern-match on it and parse another document without invalidating it. A second API exposes the borrowed tape. The two paths share the scanner and parser, so their difference measures tree materialization.

Figure 4 gives the main result. The reusable and one-shot DOM rows return the same `Json.t`. Only the former retains grown scratch buffers between calls. Appendix A describes the benchmark and testing setup.

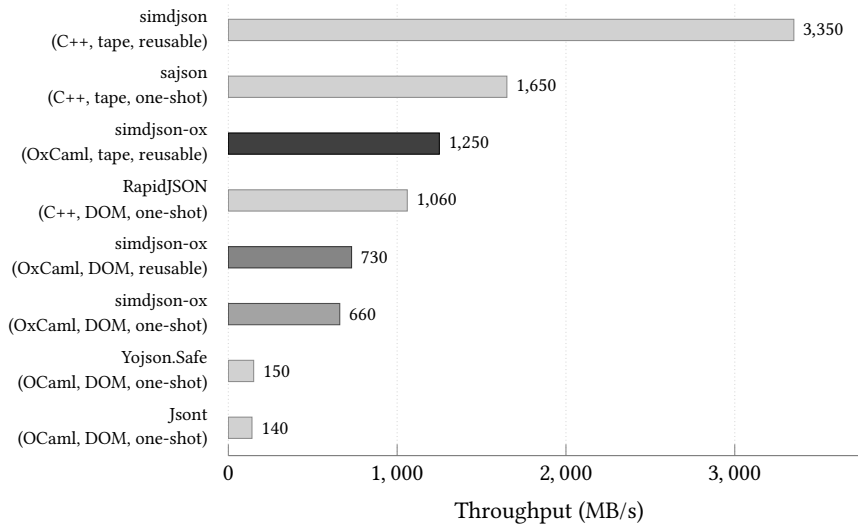

Fig. 4. Geometric-mean throughput over seven inputs, rounded to the nearest 10 MB/s. Labels give the implementation language, result representation, and usage mode. We use *DOM* only for a materialized tree. Parser-owned flat storage is labelled *tape*, including simdjson's DOM API.

The owned parser is five times faster than the existing OCaml rows. The tape is faster again but remains well behind C++ simdjson. The features behind the improvement are standard in systems languages. A five-fold gain from adding them suggests that regular OCaml can leave meaningful performance untapped when code cannot express the representations or operations it needs.

3 What OxCaml made possible

Four OxCaml features let us implement the complete parser in OCaml, including the SIMD-heavy scanner and string decoder.

3.1 Unboxed types

Unboxed types carry data outside the ordinary OCaml *value* layout. In the standard OCaml value representation [5], a standalone `float` normally points to a heap block. `float#` carries the payload directly (Figure 5). Unboxed products and records can likewise hold scanner state and results without intermediate blocks. Built-in variants such as `or_null` avoid an option block in the same way.

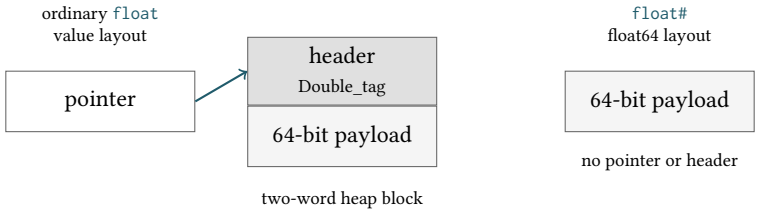

Fig. 5. Source-level representation of boxed and unboxed floating-point values on a 64-bit target. Optimizations may eliminate particular boxes.

3.2 SIMD support

OxCaml exposes built-in SIMD vector types. Loading 16 bytes produces one `int8x16#`. Comparing it with a vector containing 16 quote bytes performs the same comparison in every lane (Figure 6). Four such loads cover one 64-byte scanner block.

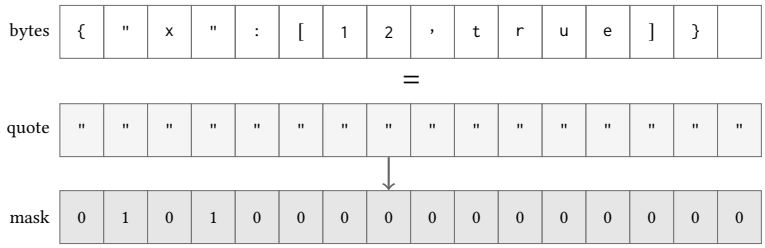

Fig. 6. A 16-lane byte comparison. Each output lane records whether the corresponding input byte is a quote. Later operations reduce the vector mask to bits.

3.3 Bit-manipulation intrinsics

Population count and count-trailing-zeros intrinsics count structural positions and extract the next set bit from a mask. When lowered correctly, each operation becomes a single CPU instruction: `popcnt` or `tzcnt`.

3.4 Stack allocation and allocation checks

`stack_` allocates a local value on the stack instead of the GC-managed heap. In the example below, both records are allocated, but the stack-allocated record is reclaimed when the function returns.

```
type point = { x : int; y : int }

let make_points () =
  let heap = { x = 1; y = 2 } in
  let local = stack_ { x = 3; y = 4 } in
  use local;
  heap
```

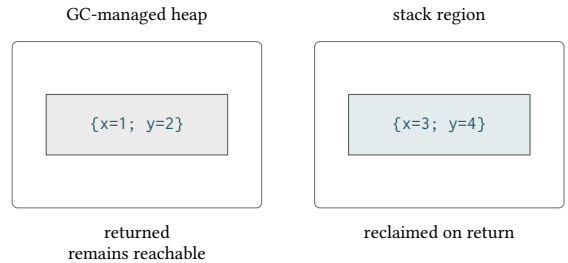

Fig. 7. Heap and stack allocation in one function. Stack allocation changes where a local value is stored. It does not remove the allocation.

`[@zero_alloc]` verifies that a function does not allocate on the OCaml heap. `[@zero_alloc opt]` performs the same check after optimization. We use the latter where inlining and scalar replacement can remove an intermediate aggregate. Section 4.2.4 shows why eliminating an aggregate mattered more than moving it to the stack.

4 Making the parser fast

The first faithful scanner implementation ran at 400 MB/s on `twitter.json`. Adding parsing on top would only make it slower. The current tape parser reaches 1.4 GB/s on the same input. Benchmarks, profiles, and generated code showed that the scanner loop allocated, small helpers remained calls, and some intrinsics became software helpers.

The work fell into three themes:

(1) CPU-level techniques transferred directly from C++ (Section 4.1).
(2) allocation and representation had to fit a generational collector (Section 4.2).
(3) some combinations of new features needed compiler annotations or library workarounds (Section 4.3).

The final scanner produces packed structural-index masks at 6 GB/s. A baseline that only traverses the input with the same SIMD loads and computes a checksum reaches 22 GB/s (Figure 8). This is the approximate throughput ceiling for the scanner on this machine. Classification, UTF-8 validation, quote tracking, and mask construction consume most of that headroom.

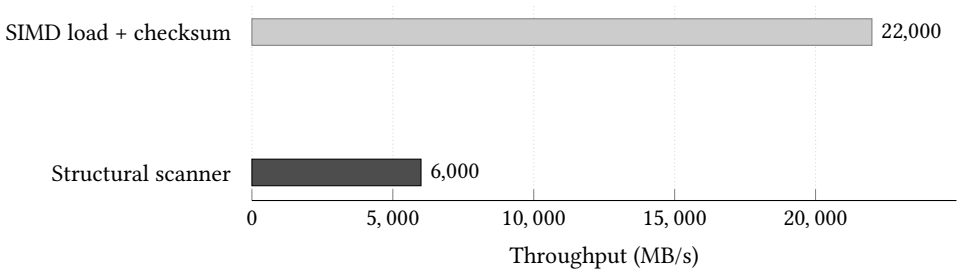

Fig. 8. Final packed structural-index production against a baseline that performs the same SIMD block loads and computes a checksum without classification.

4.1 CPU-level techniques transferred from C++

The usual CPU-level techniques worked: unroll common cases, outline rare ones, fuse passes, and interleave independent work. We kept changes only when benchmarks and profiles showed an improvement. The following examples show how these techniques appeared in the scanner and parser.

4.1.1 Interleaving scalar and vector work. Writing a structural mask also runs `popcnt` and updates the index count. The original loop did this after classifying block N. The revised loop loads N, writes and counts the mask for N-1, then classifies N (Figure 9). This exposes scalar work while vector loads are in flight.

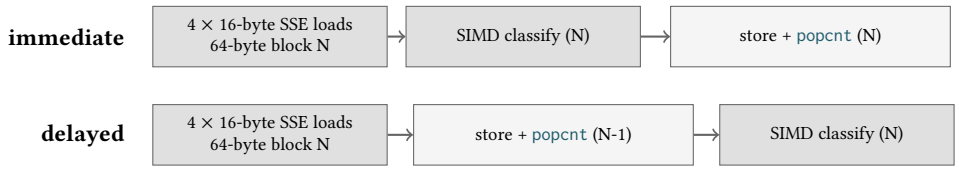

Fig. 9. Delaying structural-mask writes by one block exposes overlap between scalar bookkeeping and the next block's vector loads. On `citm_catalog`, the delayed schedule took 1.70 s versus 1.76 s for the immediate schedule and retired slightly more instructions.

Backend stalls fell from 25% to 22%, even though the new loop completed ("retired") slightly more instructions. Exposing independent work mattered more than minimizing the count.

4.1.2 Fusing string scanning and copying. While parsing a JSON string, the parser must find its closing quote, decode any escapes, and store the decoded bytes. For the tape, those bytes go into parser-owned storage. C++ simdjson's `copy_and_find` loop[1] copies each SIMD block while looking for the closing quote or a backslash. Our first implementation had preserved the SIMD search but lost the fusion: it found the end of an ordinary span, then passed that span to `Stdlib.Buffer.add_substring` to read and copy it a second time.

On `twitter`, decoded strings average 15–20 bytes, and `add_substring`'s checks and separate runtime blit accounted for 8% of sampled cycles. A reusable `Bytes.t` exposed the write position and let each 16-byte iteration:

(1) load the 16-byte SIMD block from the JSON input.
(2) store all 16 bytes at the current output position.
(3) compute the special-byte mask and find its first set bit.
(4) advance the output position by only the bytes before that bit.

The store is speculative. Bytes after a quote or backslash are written but not committed and are overwritten later. A quote ends the string. A backslash enters the scalar escape decoder (Figure 10).

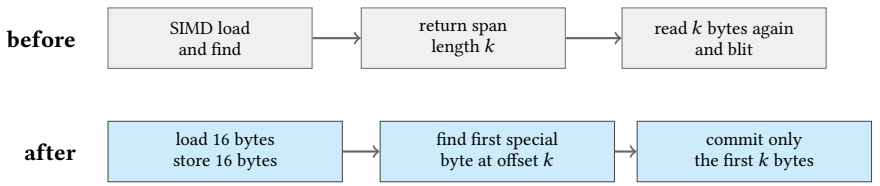

Fig. 10. Restoring simdjson's `copy_and_find` structure. The OxCaml tape decoder now stores each loaded SIMD block immediately and commits only the prefix that belongs to the string.

Figure 11 shows gains of 17–28% on string-heavy fixtures. The owned path still uses `String.sub` for ordinary strings because an owned OCaml string requires a final allocation and copy.

[1] simdjson, https://github.com/simdjson/simdjson/blob/master/include/simdjson/westmere/stringparsing_defs.h.

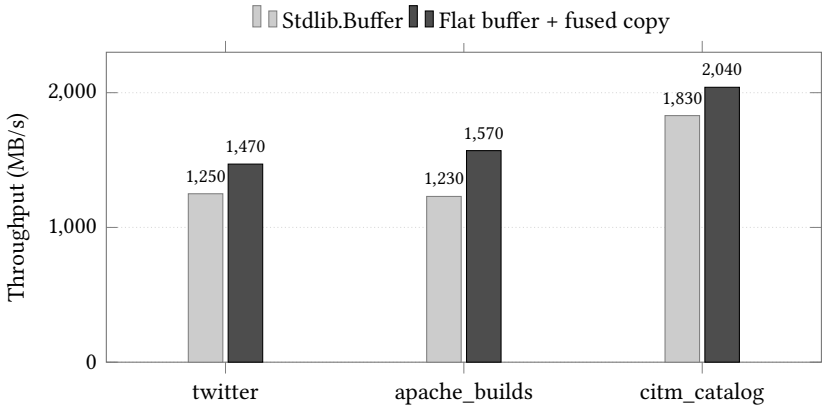

Fig. 11. Effect of the flat decoded-string buffer and fused scan-and-copy on string-heavy fixtures for simdjson-ox (OxCaml, tape, reusable).

4.1.3 The decimal-integer loop. The standard OCaml number-conversion functions were far too slow for this parser, so it uses a dedicated integer parser and the Clinger and Lemire fast paths for floating-point conversion [3]. Decimal integers start with an eight-digit SWAR ("SIMD within a register") fast path. A scalar loop handles the remaining digits. Figure 12 shows 19 instructions per digit in OxCaml against eight in C++. Both compute $v = 10v + d$. OxCaml also handles tagged positions and digits, an explicit bound, and a runtime poll—the safe-point check that lets the runtime interrupt long-running native code.

C++ simdjson — 8 instructions

```
.loop:
 imul   $0xa,%rdx,%rdx   # value *= 10
 movzbl %cl,%ecx         # digit
 add    $0x1,%rax        # next byte
 add    %rcx,%rdx        # value += digit
 movzbl (%rax),%r12d     # load byte
 lea    -0x30(%r12),%ecx # byte - '0'
 cmp    $0x9,%cl         # digit?
 jbe    .loop
```

OxCaml tape — 19 instructions

```
.loop:
 mov    %rsi,%rax
 sar    $1,%rax              # [R] untag pos
 movzbq 0(%rbp,%rax),%rax    # load byte
 lea    -0x5f(%rax,%rax),%r8 # [R] tagged digit
 cmp    $1,%r8
 jl     .done
 cmp    $0x13,%r8
 jg     .done                # digit range
 add    $2,%rsi              # [R] tagged pos++
 cmp    %r13,%rsi
 setl   %al
 movzbq %al,%rax             # [R] bound value
 sar    $1,%r8               # [R] untag digit
 imul   $0xa,%rdi,%rdi
 add    %r8,%rdi             # value = 10v+d
 cmp    (%r14),%r15          # [P] runtime poll
 jbe    .poll
 test   %rax,%rax            # [R] test bound
 jne    .loop
```

Fig. 12. Exact decimal-integer loop bodies from the linked x86 binaries. `[R]` marks representation or explicit-control work. `[P]` marks the runtime poll. The remaining instructions perform the parse.

The OxCaml loop clearly does more work, but it does not necessarily take 19/8 as long. Modern out-of-order CPUs can execute independent instructions in parallel. For example, some representation and bounds-checking work can overlap the integer multiplication. Instruction count exposes the extra work, while benchmarks determine its actual cost. Unrolling the first scalar digit shows the difference: it increased the number of retired instructions but removed a common back edge and runtime poll, producing the largest gain in Table 1.

Table 1. Retained number-parser checkpoints on `numbers.json`. The largest throughput gain increased the retired-instruction count.

Checkpoint	Tape MB/s	Instructions	Branches
Investigation baseline	840	45.1B	8.0B
Remove redundant finite checks	860	43.9B	7.5B
Unroll first scalar tail digit	890	44.3B	7.4B
Outline cold paths, trim scan result	900	42.1B	7.2B

Outlining rare integer and float paths also reduced register pressure and code size. Routing every float through substring allocation and `float_of_string_opt`, by contrast, reduced `numbers.json` to 170 MB/s. Native decimal conversion is essential.

4.2 Memory behaviour did not transfer

OCaml allocates into a minor heap and promotes survivors. Old-to-young pointers require a write barrier, and references left in reusable buffers keep objects reachable for longer. The structural index, the parser's internal buffers for accumulating array and object elements, and the result tree therefore need different allocation strategies.

4.2.1 Removing allocation from the scanner loop. The first scanner expanded each mask into positions and appended them to `Dynarray`. Each live slot is an `Elem` block. Clearing replaces it with `Empty`, so refilling allocates a block per index even when capacity is retained.

Packed masks remove per-index appends: input length determines an exact `int64# array`, with one word per block (Figure 13).

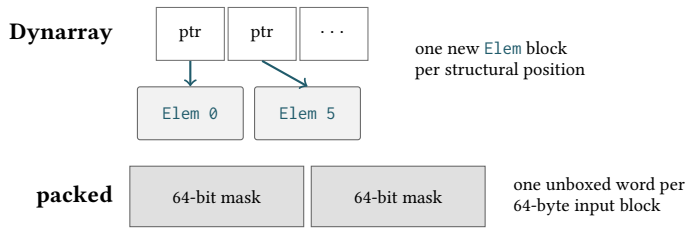

Fig. 13. Replacing expanded structural positions with packed masks. Retaining Dynarray capacity does not retain its `Elem` blocks. Refilling it allocates again. The packed array is allocated once at its final size.

The fold still allocated two boxed records per 64 bytes:

```
Simd.Int8x64.String.fold_blocks input
  ~init:{ state = initial_state; block_idx = 0 }
  ~f:(fun { state; block_idx } block ->
    let result = scan_block state block in
    Structure.set_mask structure block_idx
      result.structural_mask;
    { state = result.state; block_idx = block_idx + 1 })
```

Unboxed products kept both records' fields in loop state. Together with packed output, this moved the scanner to 2 GB/s.

4.2.2 Representing the structural index. The scanner naturally produces masks. The parser consumes positions. We compared packed masks with C++ simdjson's flat 32-bit indices (Figure 14). Dynarray remains in the benchmark only as the clearly slower baseline. Packed won overall, although flat was faster when its worst-case allocation was already warm.

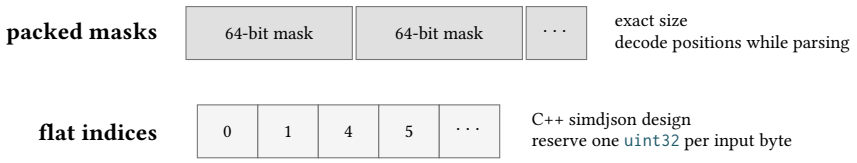

Fig. 14. The two viable scanner-to-parser representations. Packed keeps the scanner's native masks. Flat expands them to direct positions.

Table 2. Structural-index throughput in GB/s, geometric mean.

Allocation	Workload	Packed	Flat	Dynarray
Reusable	produce	6.0	4.1	1.0
Reusable	produce + consume	3.2	3.5	0.9
One-shot	produce	4.9	1.1	0.4
One-shot	produce + consume	2.9	1.0	0.3

After warmup, packed and flat allocate no reported words. Flat is about 10% faster when consuming every position, but reserves four bytes per input byte against one bit for packed. On `canada.json`, that is 9 MB versus 280 KB.

End to end, flat gains 4% with a reusable parser but loses 25% one-shot because each call initializes its worst-case reservation (Figure 15). Packed uses 32 times less capacity and works well in both modes.

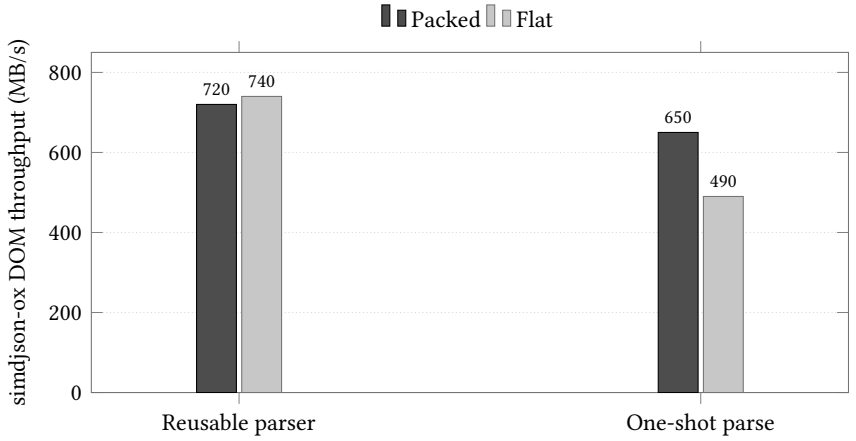

Fig. 15. The same representation choice in the two public usage modes. Flat helps only when its four-bytes-per-input-byte reservation is already warm.

C++ simdjson benchmarks a warmed parser, making flat preallocation look almost free. The one-shot result shows why usage mode matters.

4.2.3 The cost of owning `Json.t`. After warmup, the 1,250 MB/s tape reports no minor or major allocation: its words and string buffer contain no OCaml pointers. Materializing `Json.t` drops throughput to 730 MB/s. On `canada.json`, an earlier profile attributed 40% of cycles to allocation and GC runtime.

Numeric leaves. Numbers expose the largest obvious opportunity. On `numbers.json`, the owned parser reaches 560 MB/s against 890 MB/s for the tape. `float#` avoids a separate float box, but `Json.Float` and `Json.Int` still allocate one variant block per leaf. Those blocks remain reachable through the result tree and may be promoted by the GC.

User-defined unboxed variants could flatten numeric alternatives while preserving the pattern-matchable API. OxCaml currently provides only built-ins such as `or_null`.[2] This remains an unmeasured but likely large opportunity.

Accumulating arrays and objects. Containers add a different source of allocation and GC work. Container length is unknown until the closing bracket. The parser therefore marks a shared scratch buffer, appends elements, copies the completed slice into the newly allocated array or object in the resulting `Json.t`, and restores the mark (Figure 16).

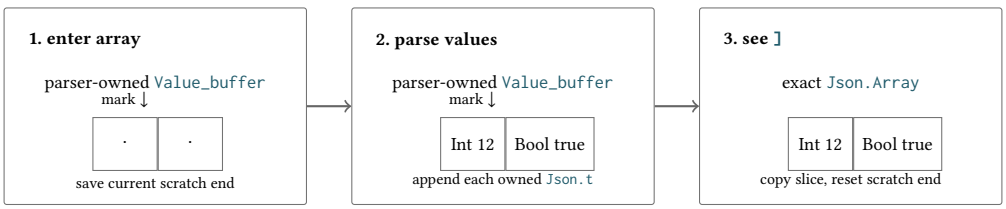

Fig. 16. Building an array whose length is unknown at the opening bracket. A mark lets nested containers share one parser-owned scratch buffer.

A reusable parser amortizes growth, but its scratch arrays eventually become old. Appending a young `Json.t` then triggers the write barrier and retains the value until its slot is cleared. Each element is also written to scratch and to the final result. This works well for long containers but poorly for small ones.

The parser therefore keeps up to four elements in local variables and constructs the result array or object directly. Larger containers spill to scratch. Despite the extra branching in the code, this was still a win because it avoided scratch writes, write barriers, and stale buffer references that could keep completed values alive. This improved `canada` and `citm_catalog` by 14%. For spills, `Array.sub` allocates and copies the result in one operation. A high-water mark also lets the parser clear stale references once per parse rather than once per container, improving representative reusable DOM rows by 8%.

We rejected several alternatives after benchmarking them. Pre-sizing 4,096 pointer slots made `twitter.json` one third slower by placing the buffer directly in the major heap. Fresh or growing per-container buffers created more minor-heap traffic. A 32M-word minor heap changed optimized `canada.json` throughput by only 1%. Pointer placement and reachability matter more than collector tuning.

[2] OxCaml, "Unboxed types," section "Unboxed variants," https://github.com/oxcaml/oxcaml/blob/main/jane/doc/proposals/unboxed-types/index.md#unboxed-variants.

4.2.4 Stack allocation removed heap traffic, not the work. The number parser returns short-lived records for digit runs and number parts. Adding `stack_` removed their accidental heap allocation, but not record construction or calls.

With inline annotations, scalar replacement removes the records. Without them, the records allocate and throughput falls. Restoring only `stack_` gives zero heap words but remains slower because the aggregate work survives (Table 3).

Table 3. Inlining and allocation experiment, MB/s. Values are medians of three pinned-CPU runs. Checksums were identical.

Build	DOM numbers	DOM twitter	Tape numbers	Tape twitter
Manual inlining (default build)	510	1,070	900	1,480
Automatic inlining (`-O3`)	370	960	660	1,210
plus explicit `stack_`	—	—	630	1,210

The fast case is not "the same allocation, on the stack." It is no allocation and no aggregate work at all. Heap allocation puts a ceiling on throughput. inlining and scalar replacement are what remove the execution cost.

OxCaml does not provide a simple escape hatch from the GC. Stack allocation and unboxed types provide more control over allocation and representation, but the collector still shapes the design. Fast code must account for generations, write barriers, promotion, reachability, inlining, and scalar replacement. OxCaml makes that code easier to express, but using its low-level features still requires understanding and sometimes working around the runtime.

4.3 Some feature combinations were still rough

The core features worked, but combinations of new layouts, SIMD values, and templates exposed optimizer and library gaps.

4.3.1 Inlining needed extensive manual guidance. The scanner ultimately needed twenty-six forced-inline annotations. together they roughly doubled its throughput. One surprising miss was a tiny local helper inside `Chars.process`. The listing highlights `make_group_mask`. Both call sites are immediately below it.

```
let process block =
  let low_class = lookup ~table:low_lut block in
  let high_class =
    block |> high_nibbles |> lookup ~table:high_lut
  in
  let klass = low_class land high_class in
  let make_group_mask group =
    equal (klass land group) (zero ())
    |> Int64_u.lnot
  in
  let operator_mask =
    make_group_mask vector_group_operators
  in
  ...
```

It survived as a call in the hot loop, as did similar boundaries across the scanner. `-O3` did not substitute for the annotations: removing them lost 10–30% depending on path and fixture (Table 3).

A parser example showed the same problem as an allocation. The original literal matcher contained a recursive local helper:

```
let expect_literal state ~start literal value =
  let literal_length = String.length literal in
  let rec loop offset =
    if offset = literal_length then
      require_boundary state (start + offset)
    else if state.input.[start + offset] = literal.[offset]
    then loop (offset + 1)
    else invalid_syntax ()
  in loop 0; value
```

Flambda2 did not lift `loop`, so each literal allocated a closure. Moving it to the top level removed the allocation.

4.3.2 Intrinsics did not always lower to the instructions they name. In the installed package snapshot,[3] the popcount and count-trailing-zeros externals lacked `[@@builtin]`. The backend supported both, but popcount became an OCaml runtime call and then libgcc's `__popcountdi2`.

A local `[@@builtin]` declaration produced one `popcnt` and improved the scanner by 10%. Count-trailing-zeros needed the same fix. Later package updates restored the metadata. Only machine-code inspection exposed the issue.

4.3.3 Non-value layouts need library support. The first object representation used `assoc# array`. Here `assoc#` is an unboxed pair-like record that stores the key and JSON value directly with layout `value & value`, removing one record per object field. While parsing an object of unknown size, however, the association scratch buffer must append fields, grow when full, copy the completed slice into the owned object, clear references to parsed values, and reuse its capacity for the next parse.

Neither Stdlib nor Base exposed bulk copy and fill for this layout. A custom binding to internal `%arrayblit` expanded to an OCaml loop with two `caml_modify` calls per product store. Clearing required another such loop because no matching fill was available.

Boxed associations restore ordinary value arrays: slicing becomes one `caml_array_blit`, clearing one `caml_array_fill`, and the runtime batches barrier work.

Depending on the fixture, unboxed associations ranged from a 10% loss to a 10% win. A better element layout helps only when libraries support its full lifecycle.

Templates can define operations for several layouts, but case lists must repeat product kinds: the grammar contains kind-abbreviation syntax that is not exposed to users. Support for non-value layouts remains spotty, and complete template families are cumbersome to write and maintain.

4.3.4 Practical guidance is sparse. The new features are documented individually, but there is little practical material on using them together with Flambda2 for performance work. Much of this investigation started with `ocamlopt -help`, compiler source, profiles, and disassembly. End-to-end examples of performance engineering with these features would make the toolchain easier to learn.

[3] Jane Street, ocaml_intrinsics_kernel, version v0.18~preview.130.83+317, https://github.com/janestreet/ocaml_intrinsics_kernel.

5 Where the remaining gap comes from

The tape runs 2.7 times slower than C++ simdjson. Some of that gap is elbow grease: simdjson has years of target-specific tuning, while Table 1 shows that our newer implementation was still finding gains. Tuning is not the whole explanation, however. The OxCaml parser also executes more work.

On a complete `canada.json` parse, the OxCaml tape path executes about 33 instructions per input byte against about 16 for C++ simdjson (Figure 17). The instruction-count ratio does not identify where the cycles are spent, but it shows that the OxCaml parser performs about twice as much work per byte. The decimal loop in Section 4.1.3 shows one source: representation work and a runtime poll around the same digit recurrence.

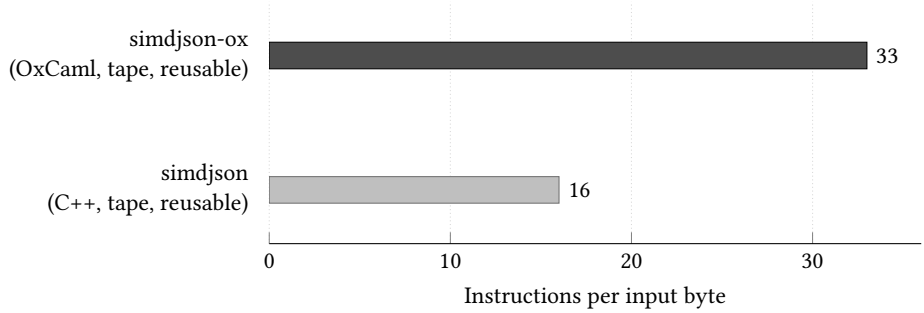

Fig. 17. Whole-document instruction count on `canada.json`. The OxCaml tape path retires about twice as many instructions per input byte.

Instruction count does not predict time, but the ratio confirms extra work across scalar parsing, representation, and runtime bookkeeping. Tuning can shrink some of it. The current OCaml representation and runtime require some of it.

6 Conclusions

SIMD, unboxed types, and integer intrinsics let us implement simdjson directly in OxCaml. The owned parser is five times faster than existing OCaml parsers, which suggests that regular OCaml can leave meaningful performance untapped when code cannot express the representations or operations it needs. Numeric variant blocks remain costly. User-defined unboxed variants may bring the parser closer to ten times the OCaml baseline, but that estimate is unmeasured.

The algorithm mapped naturally, but the first version was still 3.5 times slower than the current one. CPU-level techniques transferred from C++. Finding where to apply them required benchmarks, profiles, and generated code.

Memory-related choices were runtime-specific. Stack allocation removed GC traffic, but inlining and scalar replacement made code fast by removing aggregates altogether. Reusable storage suited the tape. The owned tree needed small-container specialization and buffer strategies designed around the GC. Writing fast OxCaml therefore requires understanding both how the generated machine code runs on the CPU and how the OCaml runtime manages allocation and pointers. More predictable inlining and broader library support for new layouts would make that work easier.

A Benchmarking and testing setup

A.1 Machine and harness

Unless noted, figures use the 15 August snapshot on Fedora 43 and OxCaml 5.2.0+ox. Benchmarks were pinned to one Golden Cove performance core, with the performance governor and ASLR disabled. Boost remained enabled, so small

differences are noise. OCaml rows use Bechamel and C++ rows use Google Benchmark. Hardware counters use the same core. Throughput is rounded to 10 MB/s, except the structural-index table in GB/s.

A.2 Usage modes

Reusable creates one parser and retains its grown buffers. C++ simdjson benchmarks this mode. *One-shot* creates parser state per document, including buffer allocation and initialization. Both return the same `Json.t` (Section 4.2.2).

A.3 What the rows measure

Yojson, Jsont, simdjson-ox DOM, and RapidJSON return owned trees. The OxCaml, simdjson, and sajson tape rows return handles into parser-owned storage. Every row completes parsing, but ownership differs. Driver checksums catch missing work. They are not cross-language equivalence tests for integer or UTF-8 policy.

The seven-fixture geometric mean is a useful summary but hides real variation (Table 4). The borrowed tape is ahead of RapidJSON on object- and string-heavy fixtures and behind it on `numbers`, `canada`, and `marine_ik`. Those are also the cases where C++ simdjson loses much of its lead, because number parsing is scalar work.

Table 4. Per-fixture throughput from the August snapshot, rounded to the nearest 10 MB/s. OCaml rows use reusable parsers.

Fixture	Ox DOM	Ox tape	RapidJSON	C++ simdjson
`apache_builds`	680	1,560	870	5,840
`citm_catalog`	1,600	2,070	1,650	5,560
`gsoc-2018`	1,090	2,260	1,090	6,520
`twitter`	1,110	1,430	1,050	5,520
`numbers`	560	890	1,050	1,640
`canada`	430	760	990	1,530
`marine_ik`	350	670	890	1,610
Geometric mean	730	1,250	1,060	3,350

A.4 Correctness

The parser passes all 105 vendored JSONChecker cases. Tests compare tape with tree, reusable with one-shot, and SIMD with an independent scalar scanner. Quickcheck covers complete values, strings, numeric edge cases, and invalid syntax. Errors lack stable positions and categories.

A.5 Use of AI tools

AI coding agents were part of the implementation workflow, primarily Codex with the ocaml-codex plugin. They worked best on tasks with an automatic check: tests, benchmark and corpus wiring, and well-specified experiments. They were also useful as fuzzy search across compiler and package sources. Tasks that depended on judgement, such as API design, representation choices, and deciding what to measure, were a poor fit. Even checkable work required steering. Tests, benchmarks, profiles, and disassembly decided which changes stayed.

A.6 Scope and limitations

The public library needs API polish, better errors, broader platform support, and production use. The scanner uses four 128-bit x86 SSE vectors per block. Wider AVX types need runtime dispatch, while ARM support waits on upstream SIMD interfaces.

References

[1] Jane Street. OxCaml. https://oxcaml.org/.
[2] Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *The VLDB Journal*, 28(6):941–960, 2019. https://doi.org/10.1007/s00778-019-00578-5.
[3] Daniel Lemire. Number parsing at a gigabyte per second. *Software: Practice and Experience*, 51(8):1700–1727, 2021. https://doi.org/10.1002/spe.2984.
[4] Daniel Lemire et al. The simdjson DOM API. https://simdjson.org/api/1.0.0/md_doc_dom.html.
[5] OCaml.org. Memory representation of OCaml values. https://ocaml.org/docs/memory-representation.
[6] Artem Pianykh. simdjson-oxcaml. https://github.com/artempyanykh/simdjson-oxcaml.