

Ultrafast¹ Lua JSON Parsing

Writing a Lua/JSON encoder + decoder as a LuaJIT module



Adam Ivora

FOSDEM 2026

¹At the time of submitting the talk proposal, writing "Fastest" might have been false advertising.

About me

Software engineer at BeamNG:

- ▶ Soft-body physics vehicle simulator.
- ▶ Closed-source C++ engine, Lua code open-source (not free as in *free beer*).

I focus on:

- ▶ Linux port,
- ▶ Sandboxing untrusted Lua code,
- ▶ **Hacking on LuaJIT.**



LuaJIT

Just-In-Time Compiler (JIT) for the Lua programming language.

- ▶ Lua is a small embeddable scripting language popular in game development.
- ▶ Comes with a high-performance interpreter.
- ▶ Compatible with Lua 5.1 API.

Why do we use LuaJIT?

1. It is **small** (we run a VM for every vehicle in a single thread).
2. It is **fast** (some BeamNG systems run 2000 times per second).

The problem

- ▶ **JSON**: human-readable data interchange format.
- ▶ **Lua**: small embeddable scripting language popular in game development.

JavaScript Object Notation (string)

```
"{\"foo\":3.5,\"arr\":\" +  
\"[\"x\",true,{}]}"
```

↔

Lua table (object)

```
{  
  ["foo"] = 3.5,  
  ["arr"] = {"x", true, {}}  
}
```

- ▶ Concerned not only about parsing, but also construction of Lua table.

There's no need to reinvent the wheel...

1. C++ libraries:

- ▶ RapidJSON: <https://github.com/Tencent/rapidjson>
- ▶ simdjson: <https://github.com/simdjson/simdjson>

2. Pure Lua libraries:

- ▶ json.lua: <https://github.com/rxi/json.lua>
- ▶ lunajson: <https://github.com/grafi-tt/lunajson>

3. Lua libraries using C API:

- ▶ Lua CJSON: <https://github.com/mpx/lua-cjson>
- ▶ RapidJSON bindings: <https://github.com/xpol/lua-rapidjson>
- ▶ lua-simdjson: <https://github.com/FourierTransformer/lua-simdjson>
- ▶ jit-cjson from OpenResty
 - ▶ not open-source :/

Solution: Pure Lua libraries

json.lua, lunajson, ...

- ▶ easy integration,
- ▶ easily extensible,
- ▶ quite slow (JSON → Lua less than 100 MB/s).

Solution: C++ libraries

rapidjson, **simdjson**, ...

- ▶ very fast (simdjson advertises gigabytes per seconds JSON parsing),
- ▶ extra Lua $\leftrightarrow$ C++ bindings needed,
 - ▶ this can make it slower
- ▶ simdjson not easily extensible (and only supports *strict* JSON decoding).

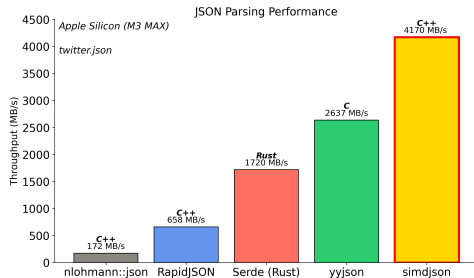


Figure: JSON Parsing Performance from <https://simdjson.org/>.

Solution: Lua libraries using C API

Lua CJSON, RapidJSON bindings, lua-simdjson

- ▶ easy integration,
- ▶ as extensible as the underlying C++ library,
- ▶ Lua C API is (surprisingly?) not very performant.

Time to reinvent the wheel.

Who put comments in my JSON?! aka JBeam (SJSON)

```
brakepads.jbeam:
{
  //BASIC BRAKE PADS
  "brakepad_F": {
    "information":{
      "authors"="BeamNG"
      name:"Basic Front Brake Pads",
      "value":[150 3 16],
    },
    ...
  }
}
```

This is a valid JBeam file!

Who put comments in my JSON?! aka JBeam (SJSON)

```
brakepads.jbeam:
{
  //BASIC BRAKE PADS
  "brakepad_F": {
    "information":{
      "authors"="BeamNG"
      name:"Basic Front Brake Pads",
      "value":[150 3 16],
    },
    ...
  }
```

JBeam (SJSON) format

Like a JSON, but:

- ▶ single-line `//` and multi-line `/* */` comments are allowed,
- ▶ object keys do not have to be enclosed in quotes,
- ▶ all commas are optional,
- ▶ `=` can be used instead of a colon `:`.

Also called Simplified JSON: <https://github.com/Autodesk/sjson>.

JBeam parsing alternatives

- ▶ Rigid parsers (simdjson) are not easily usable.
- ▶ We rolled our own parser written in pure Lua.

beamng-json.lua

- ▶ Pure battle-tested Lua implementation, 300 lines of code.
- ▶ Hand-written recursive descent parser.
- ▶ As JBeam is a superset of JSON, it is also a JSON parser.
 - ▶ Not a validating parser though!
- ▶ Written to be JIT friendly as we use LuaJIT.

beamng-json.lua

```
1  local function decode(str)
2      if str == nil then return nil end
3      gcrunning = collectgarbage("isrunning")
4      collectgarbage("stop")
5      s = str
6      local c, i = skipWhiteSpace(1)
7      local result = peekTable[c](i)
8      s = nil
9      if gcrunning then collectgarbage("restart") end
10     return result
11 end
```

beamng-json.lua: pausing GC

```
1  local function decode(str)
2    if str == nil then return nil end
3    gcrunning = collectgarbage("isrunning")
4    collectgarbage("stop") -- disable garbage collection
5    s = str
6    local c, i = skipWhiteSpace(1)
7    local result = peekTable[c](i)
8    s = nil
9    if gcrunning then collectgarbage("restart") end
10   return result
11 end
```

beamng-json.lua: whitespace skipping

```
1  local function decode(str)
2    if str == nil then return nil end
3    gcrunning = collectgarbage("isrunning")
4    collectgarbage("stop") -- disable garbage collection
5    s = str
6    local c, i = skipWhiteSpace(1)
7    local result = peekTable[c](i)
8    s = nil
9    if gcrunning then collectgarbage("restart") end
10   return result
11 end
```

beamng-json.lua: skipWhiteSpace

```
1 local function skipWhiteSpace(i)
2     local c = byte(s, i); i = i + 1
3     -- matches space tab newline or comma
4     while (c ~= nil and c <= 32) or c == 44 do
5         c = byte(s, i); i = i + 1
6     end
7     if c == 47 then c, i = skipCommentSpace(i) end -- / -- read comment
8     return c, i - 1
9 end
```

- ▶ Whitespace skipping can take a substantial portion of the parsing time!

beamng-json.lua: peekTable

```
1  local function decode(str)
2      if str == nil then return nil end
3      gcrunning = collectgarbage("isrunning")
4      collectgarbage("stop") -- disable garbage collection
5      s = str
6      local c, i = skipWhiteSpace(1)
7      local result = peekTable[c](i)
8      s = nil
9      if gcrunning then collectgarbage("restart") end
10     return result
11 end
```

beamng-json.lua: What do we peek on?

- ▶ n: null
- ▶ t: true
- ▶ f: false
- ▶ I: Infinity
- ▶ 0-9, +, -: numbers
- ▶ ": strings
- ▶ /: comments²

²in SJSON/JBeam only.

beamng-json.lua: peekTable value

Some of them are simple:

```
1 peekTable[116] = function(si) -- t
2   local b1, b2, b3 = byte(s, si+1, si+3)
3   if b1 == 114 and b2 == 117 and b3 == 101 then -- rue
4     return true, si + 4
5   else
6     jsonError('Error reading value: true', si)
7   end
8 end
```

beamng-json.lua: peekTable object

```
1 peekTable[123] = function(si) -- {
2     local result = tablenew(0, 3)
3     local c, i = skipWhiteSpace(si + 1)
4     while c ~= 125 do -- }
5         key, i = readKey(i, c)
6         repeat c = byte(s, i); i = i + 1          -- skipWhitespace
7         until (c == nil or c > 32) and c ~= 44 -- whitespace or comma
8         result[key], i = peekTable[c](i - 1)
9         repeat c = byte(s, i); i = i + 1          -- skipWhitespace
10        until c ~= 44 and (c == nil or c > 32) -- whitespace or comma
11        if c == 47 then c, i = skipCommentSpace(i) end -- / -- read comment
12    end
13    return result, i + 1
14 end
```

beamng-json.lua: peekTable object – Preallocation

```
1 peekTable[123] = function(si) -- {
2     local result = tablenew(0, 3) -- narray = 0, nhash = 3
3     local c, i = skipWhiteSpace(si + 1)
4     while c ~= 125 do -- }
5         key, i = readKey(i, c)
6         repeat c = byte(s, i); i = i + 1          -- skipWhitespace
7         until (c == nil or c > 32) and c ~= 44 -- whitespace or comma
8         result[key], i = peekTable[c](i - 1)
9         repeat c = byte(s, i); i = i + 1          -- skipWhitespace
10        until c ~= 44 and (c == nil or c > 32) -- whitespace or comma
11        if c == 47 then c, i = skipCommentSpace(i) end -- / -- read comment
12    end
13    return result, i + 1
14 end
```

beamng-json.lua: peekTable object – Inlining

```
1 peekTable[123] = function(si) -- {
2     local result = tablenew(0, 3)
3     local c, i = skipWhiteSpace(si + 1)
4     while c ~= 125 do -- }
5         key, i = readKey(i, c)
6         repeat c = byte(s, i); i = i + 1      -- skipWhitespace
7         until (c == nil or c > 32) and c ~= 44 -- whitespace or comma
8         result[key], i = peekTable[c](i - 1)
9         repeat c = byte(s, i); i = i + 1      -- skipWhitespace
10        until c ~= 44 and (c == nil or c > 32) -- whitespace or comma
11        if c == 47 then c, i = skipCommentSpace(i) end -- / -- read comment
12    end
13    return result, i + 1
14 end
```

Benchmarking protocol

- ▶ LuaJIT v2.1 rolling (707c12b), Ryzen 5600G (3.9 GHz), Fedora 42
- ▶ Measuring full passes through the datasets.
- 1. Run passes until either 1000 passes are complete or 10 seconds pass.
 - ▶ Collect garbage before every pass.
 - ▶ Flush JIT cache before every pass.
- 2. Take the mean pass time.
- 3. Repeat previous steps 5 times, take the mean of the means and calculate throughput ($\frac{\text{time}}{\text{file size}}$).

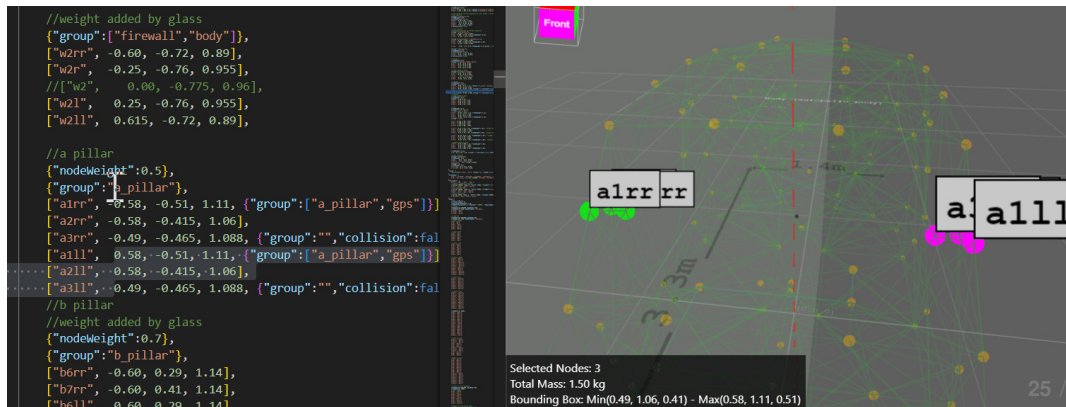
Further reducing variance:

- ▶ `nice -n -20`
- ▶ Forcing the same CPU core using `taskset`.

JBeam dataset

All 4,950 JBeam (SJJSON) files from the latest release of BeamNG.drive.

- ▶ Describes the physics properties and structure of all the vehicle parts.
- ▶ Full of numbers and very short strings.



Benchmark on the JBeam dataset (93.8 MB)

Our parser is the fastest!

	JIT off	JIT on
beamng-json.lua	30.5 MB/s	222.6 MB/s
json.lua	—	—
lunajson	—	—
lua-simdjson	—	—

JSON dataset

20 files from <https://github.com/simdjson/simdjson-data>:

```
127275 apache_builds.json
2251051 canada.json
1727204 citm_catalog.json
65132 github_events.json
11812 google_maps_api_compact_response.json
26102 google_maps_api_response.json
3327831 gsoc-2018.json
220346 instruments.json
2983466 marine_ik.json
723597 mesh.json
1577353 mesh.pretty.json
150124 numbers.json
510476 random.json
11356 repeat.json
10075 twitter_api_compact_response.json
15253 twitter_api_response.json
562408 twitterescaped.json
631515 twitter.json
42233 twitter_timeline.json
533178 update-center.json
```

Benchmark on the JSON dataset (15.5 MB)

Our parser is not the fastest anymore :/

	JIT off	JIT on
beamng-json.lua	31.9 MB/s	240.7 MB/s
json.lua	16.1 MB/s	40.1 MB/s
lunajson	37.9 MB/s	43.7 MB/s
lua-simdjson	313.7 MB/s	324.4 MB/s

twitter.json dataset

A single file with Twitter statuses, JSON parsers commonly use it for testing, also part of the JSON dataset.

```
{ "statuses": [  
  {  
    "metadata": {  
      "result_type": "recent",  
      "iso_language_code": "ja"  
    },  
    "created_at": "Sun Aug 31 00:29:15 +0000 2014",  
    "id": 505874924095815700,  
    "text": "... <JAPANESE CHARACTERS>",  
    ...  
  },  
  ...  
]
```

```
}
```

Benchmark on twitter.json (631 kB)

	JIT off	JIT on
beamng-json.lua	34.2 MB/s	230.9 MB/s
json.lua	19.8 MB/s	78.4 MB/s
lunajson	70.6 MB/s	83.7 MB/s
lua-simdjson	501.7 MB/s	500.7 MB/s

Towards a faster JSON Lua parser

beamng-json.lua parser:

- ▶ Pure Lua, beats other pure Lua parsers in the benchmark.
- ▶ Performance heavily relies on JIT.
- ▶ Slower on JSON files than Lua bindings for simdjson.

Can going closer to the source help us?

LuaJIT source code

- ▶ C99 + hand-written assembly (we don't need to touch).
- ▶ 56 `lj_*.c` and 14 `lib_*.c` files.
- ▶ We are only interested in `lj_serialize.c` (serialization) and `lib_buffer.c` (string buffers).

LuaJIT string buffers

As LuaJIT strings are immutable and interned, the **string buffer** structure is there for efficient string manipulation. From https://luajit.org/ext_buffer.html:

The string buffer library allows high-performance manipulation of string-like data. Unlike Lua strings, which are constants, string buffers are mutable sequences of 8-bit (binary-transparent) characters. Data can be stored, formatted and encoded into a string buffer and later converted, extracted or decoded.

- ▶ The string buffer `sbx` consists of:
 - ▶ read pointer `const char *r`,
 - ▶ write pointer `char *w`.
- ▶ If `r == w`, buffer is empty.

LuaJIT buffer serialization

- ▶ Implemented in `lj_serialize.c`.
- ▶ Internal binary format, but we can borrow the code for creating a JSON encoder and decoder.

```
1 buf = require('string.buffer').new()
2 buf:encode({1, 2, 3}) -- appends to buffer (moves w ptr)
3 obj = buf:decode()    -- consumes from buffer (moves r ptr)
4 assert(obj[1] == 1 and obj[2] == 2 and obj[3] == 3)
```

beamng-json.c: initial version

- ▶ Almost one-to-one rewrite of beamng-json.lua.
- ▶ Bound checking has to be explicit now.

beamng-json.c: decode

```
1 char *serialize_json_get(char *r, SBufExt *sbx, TValue *o) {
2     char *w = sbx->w;
3     r = skip_white_space(r, w, sbx);
4     if (LJ_LIKELY(r < w)) {
5         switch (*r) {
6             case '{':
7             case '[':
8                 return peek_table(r, w, sbx, o);
9             break;
10        }
11    }
12 }
```

beamng-json.c: skipWhiteSpace

Naive but working implementation:

```
1 char *skip_white_space(char *r, char *w, SBufExt *sbx) {
2     while (LJ_LIKELY(r < w) && (*r <= ' ' || *r == ',')) {
3         r++;
4     }
5     if (LJ_LIKELY(r < w) && *r == '/') {
6         r = skip_comment_space(r + 1, w, sbx); // / -- read comment
7     }
8     return r;
9 }
```

beamng-json.c: peekTable value

```
1 char *peek_table(char *r, char *w, SBufExt *sbx, TValue *o) {
2     if (LJ_LIKELY(r < w)) {
3         switch (*r) {
4             case 't': {
5                 const char val[4] = "true";
6                 if (LJ_UNLIKELY(r + sizeof(val) > w ||
7                     strncmp(r + 1, val + 1, sizeof(val) - 1) != 0)) {
8                     lj_err_callerv(sbufL(sbx), LJ_ERR_BADJSON_INVALIDVAL, val);
9                 }
10                setboolV(o, 1);
11                return r + sizeof(val);
12            }
13            ...
14        } } }
```

beamng-json.c: peekTable value

```
1 char *peek_table(char *r, char *w, SBufExt *sbx, TValue *o) {
2     if (LJ_LIKELY(r < w)) {
3         switch (*r) {
4             case 't': {
5                 const char val[4] = "true";
6                 if (LJ_UNLIKELY(r + sizeof(val) > w
7                     strncmp(r + 1, val + 1, sizeof(val) - 1) != 0)) {
8                     lj_err_callerv(sbufL(sbx), LJ_ERR_BADJSON_INVALIDVAL, val);
9                 }
10                setboolV(o, 1);
11                return r + sizeof(val);
12            }
13            ...
14        } } }
```

beamng-json.c: peekTable array

```
1  case '[': {
2      GCTab *t = lj_tab_new(sbufL(sbx), 4, hsize2hbits(0));
3      int i = 1;
4      settabV(sbufL(sbx), o, t);
5      r = skip_white_space(r + 1, w, sbx);
6      while (LJ_LIKELY(r < w && *r != ']')) {
7          TValue *v = lj_tab_setint(sbufL(sbx), t, i++);
8          r = peek_table(r, w, sbx, v);
9          r = skip_white_space(r, w, sbx);
10     }
11     if (LJ_LIKELY(r < w && *r == ']')) {
12         return r + 1;
13     }
14 }
```

beamng-json.c: peekTable array – Preallocation

```
1  case '[': {
2      GCTab *t = lj_tab_new(sbufL(sbx), 4, hsize2hbits(0));
3      int i = 1;
4      settabV(sbufL(sbx), o, t);
5      r = skip_white_space(r + 1, w, sbx);
6      while (LJ_LIKELY(r < w && *r != ']')) {
7          TValue *v = lj_tab_setint(sbufL(sbx), t, i++);
8          r = peek_table(r, w, sbx, v);
9          r = skip_white_space(r, w, sbx);
10     }
11     if (LJ_LIKELY(r < w && *r == ']')) {
12         return r + 1;
13     }
14 }
```

Benchmark on the JBeam dataset (93.8 MB)

	JIT off	JIT on
beamng-json.lua	30.5 MB/s	222.6 MB/s
beamng-json.c		
• initial version	219.5 MB/s	216.3 MB/s

No speedup when rewriting to C.

- ▶ Optimized LuaJIT code can be quite fast.
- ▶ We're not done yet, time for optimizations.

Optimizations: Branch predictor hints

- ▶ LJ_LIKELY and LJ_UNLIKELY are existing macros from LuaJIT,
- ▶ We put the annotations to the places where the unlikely path is an error state.

```
#define LJ_LIKELY(x)    __builtin_expect(!!(x), 1)  
#define LJ_UNLIKELY(x) __builtin_expect(!!(x), 0)
```

Optimizations: Number parsing

Initial implementation:

```
1 char *read_number(char *r, char *w, SBufExt *sbx, TValue *o) {
2     char *rbegin = r; char back = *r;
3     r = ...; // find the end character of the number
4     TValue tmp;
5     StrScanFmt fmt = lj_strscan_scan(rbegin, r - rbegin, &tmp, ...);
6     if (fmt == STRSCAN_ERROR) {
7         lj_err_caller(sbufL(sbx), LJ_ERR_BADJSON_INVALIDNUM);
8         return NULL;
9     }
10    o->u64 = tmp.u64;
11    return r;
12 }
```

Optimizations: Number parsing

Simplify the fast path:

```
1 char *read_number(char *r, char *w, SBufExt *sbx, TValue *o) {
2     char *rbegin = r;
3     TValue tmp; tmp.n = 0.0;
4     while (LJ_JSON_LIKELY(r < sbx->w)) {
5         const unsigned char digit = (unsigned char)(*r - '0');
6         if (digit > 9) break;
7         tmp.n = 10 * tmp.n + digit;
8         r++;
9     }
10    ...
11 }
```

Optimizations: Number parsing

```
1  case '.': {
2      lua_Number f = 0, scale = 0.1;
3      r++;
4      while (LJ_JSON_LIKELY(r < sbx->w)) {
5          const unsigned char digit = (unsigned char)(*r - '0');
6          if (digit > 9) break;
7          f += digit * scale;
8          scale *= 0.1;
9          r++;
10     }
11     tmp.n += f;
12     break;
13 }
```

Optimizations: Number parsing

- ▶ For parsing floats of the form `6.02+e23`, `lj_strscan_scan` is still used.
- ▶ Focus on optimizing the common code path.

Optimizations: Inlining

LuaJIT contains the appropriate macros again:

```
#define LJ_INLINE inline  
#define LJ_AINLINE inline __attribute__((always_inline))
```

- ▶ LJ_AINLINE is used for most functions in the parser.

Optimizations: Scratch buffer

- ▶ Use a profiler to check bottlenecks.
 - ▶ I used `perf` and Visual Studio profiler.
- ▶ Substantial amount of time was spent in `lj_alloc_realloc`, which is called when we need to grow a Lua table.
- ▶ But we don't know the size of the table before we parse all its children.
 - ▶ Also this happens at every recursion level at the same time.
- ▶ Behold the scratch buffer optimization!

Optimizations: Scratch buffer

```
1 char *lj_json_read_array(char *r, SBufExt *sbx, uint32_t scr) {
2     while (LJ_JSON_LIKELY(r < sbx->w) && *r != ']') {
3         uint32_t v = lj_json_scratch_pushn(L, 1);
4         r = lj_json_deserialize_peek(r, sbx, v);
5         r = lj_json_skip_white_space(r, sbx);
6         asize++;
7     }
8     t = lj_tab_new_ah(L, asize + 1, 0);
9     TValue *base = &lj_json_scratch[lj_json_scratch_popn(L, asize)];
10    TValue *array = tvref(t->array) + 1;
11    memcpy(array, base, asize*sizeof(TValue));
12    settabV(L, &lj_json_scratch[scr], t);
13    return r;
14 }
```

Optimizations: Scratch buffer

```
1  TValue *lj_json_scratch = lj_mem_newvec(L, 32, TValue);
2  // allocate space for n TValues and return index of the first one on the scratch
3  uint32_t lj_json_scratch_pushn(lua_State *L, uint32_t n) {
4      if (LJ_UNLIKELY(lj_json_scratch_count + n >= lj_json_scratch_capacity)) {
5          lj_mem_growvec(L, lj_json_scratch, lj_json_scratch_capacity, ...);
6      }
7      uint32_t scr = lj_json_scratch_count;
8      lj_json_scratch_count += n;
9      return scr;
10 }
11 // pop n TValues on the stack and return index of the first one
12 uint32_t lj_json_scratch_popn(lua_State *L, uint32_t n) {
13     lj_json_scratch_count -= n;
14     return lj_json_scratch_count;
15 }
```

Optimizations: Scratch buffer

- ▶ Dynamic stack for temporary Lua values.
- ▶ We only deal with one dynamically allocated scratch buffer.
- ▶ While parsing arrays/objects we can always create a table of exact size.
- ▶ TValues are still copied from the scratch buffer, but `realloc` copy is avoided.
- ▶ You cannot do this on Lua level (at least I tried and it doesn't bring performance improvement).

Optimizations: Intrinsics

- ▶ SSE2, SSE4, ARM Neon versions.
- ▶ Two primitives:
 1. Skip until single character → find end of single-line or multi-line comments.
 2. Skip to string end → find next quote or \ character.
- ▶ Similar to RapidJSON implementation.

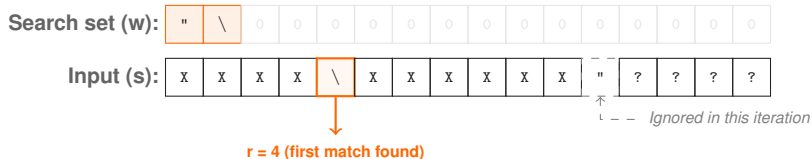
Optimizations: Intrinsics – SSE4 skipToStringEnd

Acts on 16-byte chunks.

```
1 char *lj_json_skip_to_string_end_simd(char *p, SBufExt *sbx) {
2     ...
3     const char strend[16] = "\\\"\\\"";
4     const __m128i w = _mm_loadu_si128((const __m128i *)&strend[0]);
5     for (; p <= sbx->w - 16; p += 16) {
6         const __m128i s = _mm_loadu_si128((const __m128i *)p);
7         const int r = _mm_cmpistri(w, s,
8             _SIDD_UBYTE_OPS | _SIDD_CMP_EQUAL_ANY | _SIDD_LEAST_SIGNIFICANT);
9         if (r != 16) return p + r;
10    }
11    return NULL;
12 }
```

Optimizations: Intrinsics – SSE4 skipToStringEnd

`__mm_cmpistri` does all the heavy lifting:



Let's benchmark again.

Benchmarking protocol

- ▶ LuaJIT v2.1 rolling (707c12b), Ryzen 5600G (3.9 GHz), Fedora 42
- ▶ Measuring full passes through the datasets.
- 1. Run passes until either 1000 passes are complete or 10 seconds pass.
 - ▶ Collect garbage before every pass.
 - ▶ Flush JIT cache before every pass.
- 2. Take the mean pass time.
- 3. Repeat previous steps 5 times, take the mean of the means and calculate throughput ($\frac{\text{time}}{\text{file size}}$).

Further reducing variance:

- ▶ `nice -n -20`
- ▶ Forcing the same CPU using `taskset`.

Benchmark on the JBeam dataset (93.8 MB)

Our parser is the fastest again!

	JIT off	JIT on
beamng-json.lua	30.5 MB/s	222.6 MB/s
beamng-json.c		
• initial version	219.5 MB/s	216.3 MB/s
• final version	368.6 MB/s	370.0 MB/s

Benchmark on the JBeam dataset (93.8 MB) – Ablations

	JIT off	JIT on
beamng-json.lua	30.5 MB/s	222.6 MB/s
beamng-json.c		
• initial version	219.5 MB/s	216.3 MB/s
• final version ³	368.6 MB/s	370.0 MB/s
• no branch hints	356.0 MB/s	353.2 MB/s
• no scratch buffer	288.0 MB/s	292.3 MB/s
• no inlining	337.8 MB/s	335.4 MB/s
• no intrinsics	335.8 MB/s	336.1 MB/s
• SSE4 intrinsics	369.0 MB/s	379.7 MB/s

³SSE2 intrinsics.

Benchmark on the JSON dataset (15.5 MB)

Also beats `lua-simdjson` on the JSON parsing dataset.

	JIT off	JIT on
<code>beamng-json.c</code>	704.7 MB/s	698.3 MB/s
<code>beamng-json.lua</code>	31.9 MB/s	240.7 MB/s
<code>json.lua</code>	16.1 MB/s	40.1 MB/s
<code>lunajson</code>	37.9 MB/s	43.7 MB/s
<code>lua-simdjson</code>	313.7 MB/s	324.4 MB/s

Benchmark on twitter.json (631 kB)

	JIT off	JIT on
beamng-json.c	864.8 MB/s	856.8 MB/s
beamng-json.lua	34.2 MB/s	230.9 MB/s
json.lua	19.8 MB/s	78.4 MB/s
lunajson	70.6 MB/s	83.7 MB/s
lua-simdjson	501.7 MB/s	500.7 MB/s

Results from the benchmark

JBeam dataset:

- ▶ 45% speedup over pure Lua parser when JIT is on.
- ▶ **More than 10x** speedup over pure Lua parser when JIT is off.

JSON dataset:

- ▶ At least 70% faster than `lua-simdjson`.
- ▶ **More than 10x** times faster than pure Lua implementations.

Tips

- ▶ Try to combine multiple optimization tricks.
- ▶ Focus on the happy common path, uncommon features can be slower (string with escape characters, scientific numbers).
- ▶ Some optimization attempts actually slow the code down, measure regularly.
- ▶ Use the profiler to find hotspots.
- ▶ Minimize data copying (realloc).

Limitations

- ▶ The benchmark measures repeated parsing performance, which is not the metric we really care about.
 - ▶ We usually parse each .json or .jbeam file once.
- ▶ Unstable performance numbers in some cases.
 - ▶ Check out the "How to Reliably Measure Software Performance" from this talk (already happened).

Are we finished?

JSON encoding

- ▶ Not our primary focus, decoding is used more often.
- ▶ Let's write a Lua → JSON encoder for completion⁴.

⁴And to fit the title of the talk :)

JSON encoding: It's really simple!

```
1 char *lj_json_serialize_put(char *w, SBufExt *sbx, cTValue *o) {
2     if (LJ_JSON_LIKELY(tvisstr(o))) {
3         w = lj_json_serialize_more(w, sbx, 1);
4         *w++ = '"';
5         w = lj_json_put_string(w, sbx, strV(o));
6         w = lj_json_serialize_more(w, sbx, 1);
7         *w++ = '"';
8     } else if (tvisnum(o)) {
9         w = lj_json_put_number(w, sbx, numV(o));
10    } else if (tvispri(o)) {
11        w = lj_json_put_bool(w, sbx, itype(o));
12    } ...
}
```

JSON encoding: It's really simple!

```
1 char *lj_json_serialize_put(char *w, SBufExt *sbx, cTValue *o) {
2     if (LJ_JSON_LIKELY(tvisstr(o))) {
3         w = lj_json_serialize_more(w, sbx, 1);
4         *w++ = '"';
5         w = lj_json_put_string(w, sbx, strV(o));
6         w = lj_json_serialize_more(w, sbx, 1);
7         *w++ = '"';
8     } else if (tvisnum(o)) {
9         w = lj_json_put_number(w, sbx, numV(o));
10    } else if (tvispri(o)) {
11        w = lj_json_put_bool(w, sbx, itype(o));
12    } ...
}
```

JSON encoding: Writing a value

```
1 char *lj_json_serialize_put(char *w, SBufExt *sbx, cTValue *o) {
2     if (LJ_JSON_LIKELY(tvisstr(o))) {
3         w = lj_json_serialize_more(w, sbx, 1);
4         *w++ = '"';
5         w = lj_json_put_string(w, sbx, strV(o));
6         w = lj_json_serialize_more(w, sbx, 1);
7         *w++ = '"';
8     } else if (tvisnum(o)) {
9         w = lj_json_put_number(w, sbx, numV(o));
10    } else if (tvispri(o)) {
11        w = lj_json_put_bool(w, sbx, itype(o));
12    } ...
```

JSON encoding: Writing a value is also really simple

```
1 char *lj_json_put_bool(char *w, SBufExt *sbx, uint32_t itype) {
2     w = lj_json_serialize_more(w, sbx, 5);
3     switch (itype) {
4     case LJ_TNIL:
5         memcpy(w, "null", 4); w += 4; break;
6     case LJ_TTRUE:
7         memcpy(w, "true", 4); w += 4; break;
8     case LJ_TFALSE:
9         memcpy(w, "false", 5); w += 5; break;
10    }
11    return w;
12 }
```

Let's benchmark again.

Benchmark on the JSON dataset – Encoding

Our encoder's performance is not bad.

	JIT off	JIT on
beamng-json.lua	42.6 MB/s	90.2 MB/s
beamng-json.c	224.0 MB/s	223.0 MB/s
json.lua	21.5 MB/s	26.9 MB/s
lunajson	32.8 MB/s	42.6 MB/s
lua-simdjson	—	—

Note: No optimizations were attempted on beamng-json.lua and beamng-json.c.

Benchmark on the JBeam dataset – Encoding

	JIT off	JIT on
beamng-json.lua	31.1 MB/s	31.2 MB/s
beamng-json.c	586.7 MB/s	590.7 MB/s
json.lua	36.6 MB/s	39.3 MB/s
lunajson	36.7 MB/s	39.4 MB/s

The End; Questions?

Benchmark code:

- ▶ <https://github.com/aivora-beamng/luajit-json-performance>

LuaJIT JSON module:

- ▶ <https://github.com/aivora-beamng/LuaJIT-json>

Slides: available at the FOSDEM talk page.

Contact:

- ▶ <https://github.com/aivora-beamng>
- ▶ <https://www.linkedin.com/in/adam-ivora/>

