Parsing Millions of URLs per Second

Yagiz Nizipli[1] and Daniel Lemire[1]

[1]Universite TELUQ

November 18, 2024

Abstract

URLs are fundamental elements of web applications. By applying vector algorithms, we built a fast standard-compliant C++ implementation. Our parser uses three times fewer instructions than competing parsers following WHATWG URL standard (e.g., Servo's rust-url) and up to eight times fewer instructions than the popular curl parser. The Node.js environment adopted our C++ library. In our tests on realistic data, a recent Node.js version (20.0) with our parser is four to five times faster than the last version with the legacy URL parser.

RESEARCH ARTICLE

Parsing Millions of URLs per Second

Yagiz Nizipli[1] | Daniel Lemire[1*]

[1]Data Science Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence
Daniel Lemire, Data Science Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: daniel.lemire@teluq.ca

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2017-03910

URLs are fundamental elements of web applications. By applying vector algorithms, we built a fast standard-compliant C++ implementation. Our parser uses three times fewer instructions than competing parsers following WHATWG URL standard (e.g., Servo's rust-url) and up to eight times fewer instructions than the popular curl parser. The Node.js environment adopted our C++ library. In our tests on realistic data, a recent Node.js version (20.0) with our parser is four to five times faster than the last version with the legacy URL parser.

KEYWORDS
URL, Text Processing, Vectorization, Performance

1 | INTRODUCTION

A Uniform Resource Locator (URL) is a unique identifier that defines a resource on the web. A typical URL might provide a protocol, a domain and a path. We offer several URL examples in Table 1.

Berners-Lee et al. (2005) defined the URL syntax was defined in the Request for Comments (RFC) 3986 [1]. However, the standard evolved organically through various implementations over the years. To address the growing difference between existing standards and practice—most notably in web browsers—the Web Hypertext Application Technology Working Group proposed the WHATWG URL Standard in 2012 [?]. Most of the popular web browsers (e.g., from Apple, Mozilla or Google) abide by the WHATWG URL standard. Unfortunately, many standard libraries still fail to follow the WHATWG URL standard: we verified that support is missing in the Java standard library (`java.net.URL`), in the Go standard library (`net/url`) and in Python's `urllib` library. Furthermore, popular URL parsers often differ in how they interpret URL strings [2, 3].

URL parsing consists in taking an input string and identifying the various components while normalizing them as needed. For example, the input string `http://你好你好.在/./a/../b/./c` should be normalized to the string

```
https://xn-6qqa088eba.xn-3ds/b/c
```

where `https:` represent the protocol, `xn-6qqa088eba.xn-3ds` is the host, and `/b/c` is the path. We may also need to parse a URL string relative to another string. For example, given the base string `http://example.org/foo/bar`, the relative string `http:/example.com/` leads to the final URL `http://example.org/example.com/`. We should also be able to modify the various components of a URL (protocol, host, username, etc.). To illustrate the complexity, our C++ software library implementing the WHATWG URL standard—and little else—has approximately 20 000 lines of code.

The WHATWG URL standard follows the robustness principle (Postel's law): *be conservative in what you send, be liberal in what you accept*. Parsing URLs using the WHATWG URL standard can be more challenging than using the earlier standard (RFC 3986). For example, consider the string `https://\tlemire.me/en/` where `\t` is the tabulation character. The WHATWG URL standard requires us to ignore the tabulation characters. A conventional URL parser following RFC 3986 (e.g., curl[1]) would reject such a string.

There are many components that impact the performance of a web application but URL parsing is practically always required. URL parsing is relatively expensive. Parsing a single URL may take 4 μs on average in a system like Node.js. In our tests, the popular curl library can parse about half a million URLs per second, yet a fast C++ number parser—converting ASCII number strings to binary floating-point numbers—can process more than 50 million numbers per second [4]. Thus we can parse almost 100 floating-point numbers in the time it takes curl to parse a single URL.

We think that popular systems such as Node.js should be able to parse several million URLs per second on modern systems without sacrificing correctness or safety. We present our work on the efficient implementation of the current WHATWG specification. Our implementation is freely available.[2] We provide benchmarks and comparisons with other fast and popular URL parsers in C, C++, and Rust, whether they follow RFC 3986 [1] (curl and Boost.URL) or WHATWG URL (Servo rust-url). We review various strategies that are efficient when parsing strings.

Our work has been integrated into the popular Node.js JavaScript runtime environment over several versions, concluding with a final integration in Node.js version 20. We are therefore able to run JavaScript benchmarks before the inclusion of our fast parser (e.g., Node.js version 18) and after its complete integration (e.g., Node.js version 20). Though many factors contribute to improved performance, we estimate that the large performance gains in URL parsing are mostly the result of our work.

2 | RELATED WORK

Much of the academic research regarding URLs relates to security issues. For example, Ajmani et al. [2] as well as Reynolds et al. [3] test a wide range of popular URL parsers: they find many differences and discuss the security implications of these differences. In our work, we sought to provide complete and rigorous support to the WHATWG URL specification.

To our knowledge, there is no related work on the production of high-performance URL parsers. However, there is related work regarding the high-performance parsing of web formats. Park et al. [5] show that we can improve the performance of web applications by parsing JavaScript concurrently. XML parsing has received much attention: e.g., Van Engelen proposes fast XML parsing with deterministic finite state automata [6], Kostoulas et al. achieve higher XML parsing speed by avoiding unnecessary data copying and transformation [7], Cameron et al. show that we can parse XML faster using SIMD instructions [8]. There is also much work regarding JSON parsing: e.g., Langdale and Lemire show that we can parse gigabytes of JSON per second using branchless routines and vectorization [9]. Binary

[1] Curl stands for *command line tool and library for transferring data with URLs* and it is sometimes capitalized as *cURL* though the official documentation and website use a lowercase name: *curl*.

[2] `https://www.github.com/ada-url/ada`

TABLE 1 URL examples.

Long URLs	`http://nodejs.org:89/docs/latest/api/foo/bar/qua/13949281/0f28b//5d49/b3020/url.html#test?payload1=true&payload2=false&test=1&benchmark=3&foo=38.38.011.293&bar=1234834910480&test=19299&3992&key=f5c65e1e98fe07e648249ad41e1cfdb0`
Short URLs	`https://nodejs.org/en/blog/`
IDN	http://你好你好.在
File	`file:///foo/bar/test/node.js`
Websocket	`ws://localhost:9229/f46db715-70df-43ad-a359-7f9949f39868`
Authentication	`https://user:pass@example.com/path?search=1`
JavaScript	`javascript:alert("nodeisawesome");`
Percent Encoding	`https://%E4%BD%A0/foo`
Pathname with dots	`https://example.org/./a/../b/./c`

data is commonly published online using segments of base64 code: we can greatly accelerate the coding and decoding of these segments [10, 11].

3 | PARSING URL STRINGS

A URL string consists of many substrings. We refer to these substrings as components. Once normalized, URL strings are ASCII but a parser may receive a Unicode string.

A URL string typically begins with a protocol (or *scheme*) string: e.g., the string `http` in `http://google.com`. The WHATWG URL specification recognizes *special* protocols that are subject to different constraints: `ftp`, `file`, `http`, `https`, `ws`, and `wss`. The protocol string is terminated by the colon character (‘`:`’).

The protocol might be followed by a host. In such cases, the protocol-terminating colon character is followed by two slash characters ‘//’. A host might be preceded by *credentials*. Credentials in a URL define the username with an optional password split with the ‘`:`’ character. E.g., `postgresql://username:password@localhost:5432`. To have credentials, a URL string must not have the protocol `file` and it must have a non-empty host. A host begins with a hostname string, optionally followed by the colon character (‘`:`’) and a port number string. Thus, given the URL string `data://example.com:8080/pathname?search`, the host is `example.com:8080` whereas the hostname is `example.com`. URL strings with a special protocol must contain a *host*, whereas it is optional for other types of URL strings. Host names may be domain names, IPv4, or IPv6 addresses.

- For non-ASCII domain names, we must follow RFC 2390 [12] which involves converting Unicode to punycode [13] and checking that various rules are satisfied.
- The IPv4 address is a 32-bit unsigned integer that identifies a network address. The WHATWG URL specification considers both `192.168.1.1` and `192.0x00A80001` as valid and equivalent IPv4 addresses. The normalized URL string is made of four decimal integers (`192.168.1.1`).
- The IPv6 address is a 128-bit unsigned integer that identifies a network address. It is represented as a list of eight 16-bit unsigned integers, also known as IPv6 pieces. We surround IPv6 addresses by square brackets: e.g.,

http://[c141:ffff:0:ffff:ffff:ffff:ffff:ffff].

- Port numbers are represented by 16-bit integers with a maximum value of 65536. Special protocol have default ports: e.g., the `http` protocol has default port 80. Default ports are omitted in the normalized string. The `file` protocol cannot have a port. It is also disallowed to have port without a hostname.

It is possible for a URL to have no host (and thus no credentials), in which case the protocol string is not followed by two slashes '`//`': e.g., `non-spec:/.//p`. The standard distinguishes between an empty host (e.g., `protocol:///mypath`) and a missing host (e.g., `protocol:/mypath`).

A URL string may contain a *pathname* after the protocol and (optional) host. If there is no host then the pathname is opaque: e.g., the URL `mailto:john@doe.com` has the opaque pathname `john@doe.com`. Otherwise a URL pathname starts with '`/`'. If the host is empty, there might be a sequence of three slash characters: e.g. `file:///file.txt`. The pathname is always optional. If the pathname contains non-ASCII characters, they are *percent encoded*: treating the characters as UTF-8 bytes, we replace non-ASCII characters with a sequence of '`%`' characters followed by two-character hexadecimal codes. For example, the character `é` is replaced by the sequence `%C3%A9`.

We may then have a search component (also called a *query*). A URL search component is represented by either null or an ASCII string and starts with the character '`?`'. It is usual for the search component to content a sequence of key-value pairs separated by the ampersand character '`&`' and linked by the equal sign '`=`': `?a=b&c=d`. The search component is percent-encoded as needed. Similarly, we may have a hash component (also called a *fragment*). The URL hash is the URL part that starts with the '`#`' character. It may also be percent-encoded.

4 | FAST PARSING

The WHATWG URL standard is specified as an algorithm following a state-machine. See Fig. 1. URL parsing begins in the *Scheme Start* state. The algorithm consumes one character at a time, and changes state according to state-specific rules. In certain scenarios, the URL state machine reverses the iteration and goes back, resulting in re-iterating the same character more than once.

We wrote our parser in C++ by initially following the finite-state design. However, the byte-by-byte processing implied by the standard is a poor choice for performance. Thus we adapted the design so that once we enter a state, we fully consume the relevant component of the URL string, as much as possible.

The standard also suggests that each component is parsed into a separate string instance. Though we optionally support this design, our default is to parse into a single string which constitutes the normalized string at the end of the parsing. We call the result an `url_aggregator` because the components are aggregated during parsing into a single buffer. Having a single buffer has several performance benefits:

- At the beginning of the parsing, we allocate a buffer that has the size of the input string, rounded up to the next power of two. Usually there is no need for further memory allocation or copying. By allocating less memory, we reduce the probability of incurring expensive cache misses.
- When querying for string components, or for the normalized string, there is no need to generate and allocate a new string instance. We may simply return an immutable view on the underlying buffer. It is made convenient by the introduction of the `string_view` class in C++17 but it is also convenient in other programming languages: Rust has string slices (`str`), Java has `CharSequence`, C# has `ReadOnlySpan<char>` and so forth.

We expect that most components consumed from input URL strings do not need to be modified and they may be copied as is. We optimized our code for this scenario by integrating tests leading to fast paths. For example, we must remove tabulation and newline characters from input strings, since they are ignored during the processing. However,

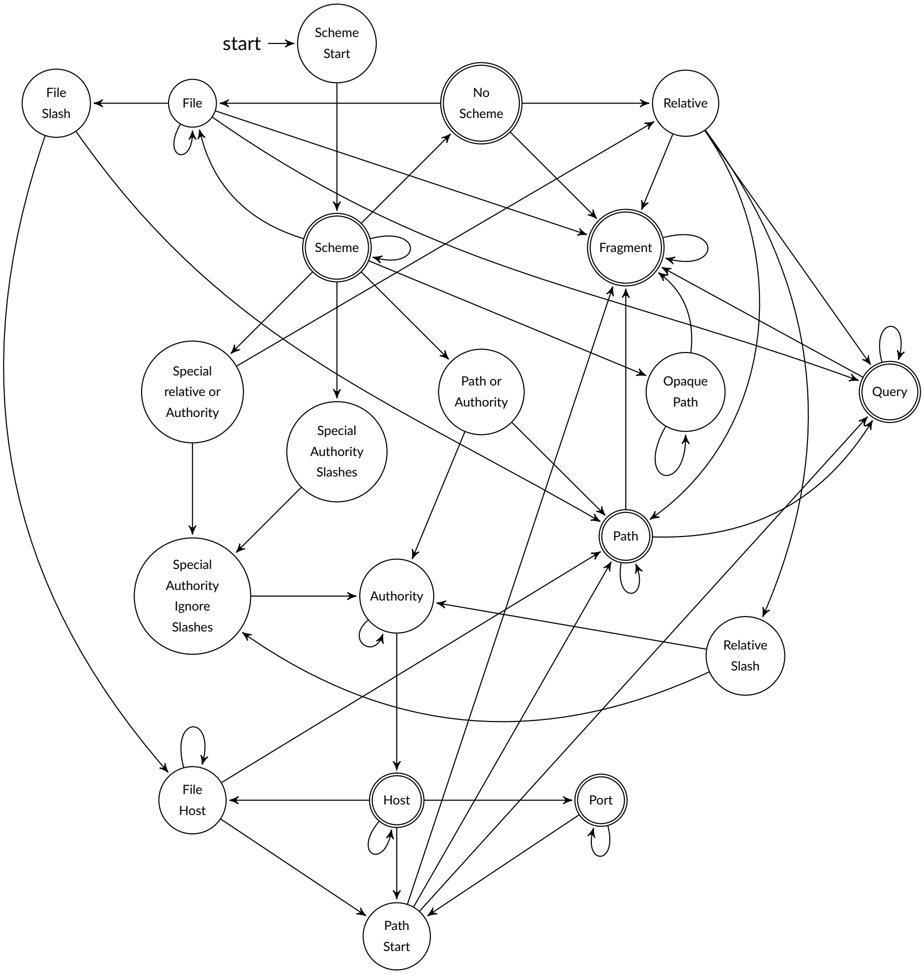


FIGURE 1 URL Parser State Machine

most input strings do not contain tabulation and newline characters. Thus we use a fast scanning function to verify that there are no such characters. Common processors (Intel, AMD, ARM, POWER) support single-instruction-multiple-data (SIMD) instructions. SIMD instructions operate on several words at once unlike regular instructions. Though different processors support different SIMD instruction sets, there is some common ground. The 64-bit processors from Intel and AMD (x64) are required to support SSE2 instructions while 64-bit ARM processors (Apple, Qualcomm, etc.) support NEON instructions. We can use these instructions through *intrinsic functions* in C and C++: these special functions often provide functionality similar to a given instruction (e.g., a NEON addition), without using assembly. Fig. 2 illustrates one function scan for characters under x64 processors, using SSE2 intrinsic function. We have also the equivalent function in NEON, as well as a fallback function for other processors. We detect the target family of processors at compile time. Effectively, the routine compares each input character with the newline and tabulation characters. When at one such character is found, we use a slow path where a temporary buffer is allocated. We write a version of the input string to the temporary buffer while omitting the newline and tabulation characters. We find in

```
bool has_tabs_or_newline(std::string_view user_input) {
  size_t i = 0;
  const __m128i mask1 = _mm_set1_epi8('\r');
  const __m128i mask2 = _mm_set1_epi8('\n');
  const __m128i mask3 = _mm_set1_epi8('\t');
  __m128i running{0};
  for (; i + 15 < user_input.size(); i += 16) {
    __m128i word = _mm_loadu_si128((const __m128i*)(user_input.data() + i));
    running = _mm_or_si128(
        _mm_or_si128(running, _mm_or_si128(_mm_cmpeq_epi8(word, mask1),
          _mm_cmpeq_epi8(word, mask2))),
          _mm_cmpeq_epi8(word, mask3));
  }
  if (i < user_input.size()) {
    uint8_t buffer[16]{};
    memcpy(buffer, user_input.data() + i, user_input.size() - i);
    __m128i word = _mm_loadu_si128((const __m128i*)buffer);
    running = _mm_or_si128(
        _mm_or_si128(running, _mm_or_si128(_mm_cmpeq_epi8(word, mask1),
          _mm_cmpeq_epi8(word, mask2))),
          _mm_cmpeq_epi8(word, mask3));
  }
  return _mm_movemask_epi8(running) != 0;
}
```

FIGURE 2 Scan for tabulation and newline characters using SSE2 intrinsic functions. The ARM NEON version is similar.

```
std::string_view is_special_list[] = {"http", " ",
   "https", "ws", "ftp", "wss", "file", " "};

enum type {
  HTTP = 0, NOT_SPECIAL = 1, HTTPS = 2, WS = 3,
  FTP = 4, WSS = 5, FILE = 6 };

type get_scheme_type(std::string_view scheme) {
  if (scheme.empty()) { return NOT_SPECIAL; }
  int hash_value = (2 * scheme.size() + (unsigned)(scheme[0])) & 7;
  std::string_view target = is_special_list[hash_value];
  if ((target[0] == scheme[0]) && (target.substr(1) == scheme.substr(1))) {
    return type(hash_value);
  } else { return NOT_SPECIAL; }
}
```

FIGURE 3 Analysis of the protocol string

practice that it is rarely needed.

Most of the strings begin with a protocol string (e.g., `file` or `https`). We must recognize a limited set of *special* protocols specified by the WHATWG URL standard. We identify that first occurrence of the colon character '`:`' and seek to recognize quickly the protocol. We expect most protocol strings to be special in practice: it is uncommon for protocols not to be one of `http`, `https`, `ws`, `wss`, `ftp` or `file`. We designed a perfect hash function [14] (see Fig. 3). Based solely on the length of the protocol string and the first character, we can distinguish any one of the special protocols. The perfect hash function returns an integer value between 0 and 6 inclusively. At several steps during the processing, the standard requires us to check the protocol. If we merely store a string value representing the protocol, then we may need to do a string-to-string comparison each time. Instead, for example, we can verify whether we have `file` protocol by comparing the protocol type with the integer value 6: an integer-to-integer comparison may result in a single instructions once compiled unlike a string comparison.

Most URL strings have a host string that must be processed. In the majority of cases, the host string requires no further processing: it is a lower-case ASCII string. We use the function of Fig. 4 to identify problematic characteristics. Effectively, it is a stream of table lookups with bitwise OR operations. Though we could use SIMD instructions for

```
static uint8_t is_forbidden_domain_code_point_table_or_upper[] = {
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

bool contains_forbidden_domain_code_point_or_upper(
    const char* input, size_t length) noexcept {
  ;
  uint8_t accumulator{};
  for (size_t i = 0; i < length; i++) {
    accumulator |=
        is_forbidden_domain_code_point_table_or_upper[uint8_t(input[i])];
  }
  return accumulator;
}
```

FIGURE 4 Function to detect forbidden of upper-case characters in host string

this purpose, the hostnames are relatively short. When the host string contains non-ASCII lower-case characters, we fall back on a relatively extensive normalization process which may include punycode encoding [12, 13]. About half of our C++ source code (or 10 000 lines) is dedicated to this normalization: thankfully it is rarely needed in practice. We also include a fast routine to detect IPv6 when the host string begins with the bracket '['. We also check for IPv4 by scanning for digits and the dot character '.'. As soon as an IPv6 or IPv4 address is found, we normalize it using a specialized routine.

We must then process the rest of the URL string, including the path, the search, and the hash substrings. These components may sometimes require percent-encoding. To avoid unnecessary percent-encoding, we search through each substring for the first character that might require percent encoding, when none is found, we can skip percent encoding entirely. Otherwise, we proceed with percent encoding from that character. We classify characters needing percent encoding using fast table lookups.

A challenge with path processing is that while most path strings require little to no work, some of them require potentially expensive processing. The fastest case is when there is no character needing percent encoding, no percent character, no backslash, and no dot character. In this fastest case, we may copy the path string as is. The second fastest case is when we can do without percent-encoding and there are no backslash nor percent characters. In this second case, we may still need to do some processing (e.g., convert `/././` to /), but it is still relatively simple. These two cases amount to the majority of path strings found in realistic scenarios. Finally, we fall back on the complete case where there might be backslashes and character encoding. The general case includes cases such as `http://www.google.com/path/%2e./` which must be normalized to `http://www.google.com/` (maybe surprisingly) because `%2e.` is equivalent to '`..`' which instructions us to shorten the path. To quickly classify the path strings, we use the algorithm of Fig. 5 which identifies the type of characters present:

- The first bit is set whenever there is a forbidden character that needs percent-encoding.
- The second bit is set whenever the backslash character is present.
- The third bit is set whenever a dot character is present.
- The fourth bit is set whenever the percent character is present.

We call the result of the function a path signature. We could use SIMD instructions for the computation of the path signature—it would be beneficial for long paths—but our signature routine is already efficient.

```
static uint8_t path_signature_table[256] = {
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0,
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

uint8_t path_signature(std::string_view input) {
  uint8_t accumulator{};
  for (size_t i = 0; i < input.size(); i++) {
    accumulator |= path_signature_table[uint8_t(input[i])];
  }
  return accumulator;
}
```

FIGURE 5 Path-signature function

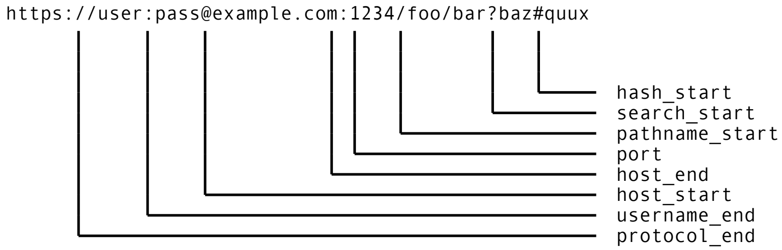


FIGURE 6 Component indexes

As we parse the input strings, we store the components (e.g., protocol, hostname) on a single buffer that becomes our normalized string. To record the location of the components, we use a convention similar to other parsers (e.g., Servo rust-url): counting the normalized string length, we only need nine integers to characterize a parsed URL. See Fig. 6. In our actual implementation compiled with GCC 12 under Linux, we use 80 bytes per URL (not counting the dynamic memory allocation), of which 32 bytes are used by the `std::string` instance that we use as our buffer. Though our memory usage could be further optimized, it is clear that storing multiple `std::string` instances would use much more memory.

4.1 | JavaScript Integration

In a system like Node.js, calling C++ from JavaScript can be relatively expensive. Indeed, creating a new JavaScript string instance from C++ data can be a costly operation. With our design where we have a single normalized string, we just need to additionally pass some integer offsets to indicate the position of the components in the string. We also provide JavaScript with the protocol type as an integer, which allows (for example) to check that we have a file URL with a single integer comparison. Components such as protocol, hostname, pathname, search, hash, etc. are computed as needed as substrings of the normalized string from within JavaScript. In effect, we reduce as much as possible the need to copy strings between C++ and JavaScript, relying instead on integer values.

5 | BENCHMARKS

For C++ benchmarking, we use the release 2.4.1 for the Ada library. Our implementation is *safe* and *correct* in the sense that it has undergone thorough testing, including extensive tests with random inputs (fuzzing).

To directly compare our C++ implementation, we use the following competitors:

- A high-quality WHATWG URL C++ library published as open-source software by Misevičius.[3] We use a code snapshot from January 26 2023.
- We use the Boost.URL C++ library version 1.81.0.
- We use the rust-url library (version 0.1.0) from the Servo web browser engine[4], building it with Rust 1.65. The Firefox browser relies on the rust-url library.
- We also use curl 7.81.0.

Both curl and Boost.URL follow the RFC 3986 standard [1] so direct comparisons must be done with care. We believe that the WHATWG URL standard is more demanding: we expect that all RFC 3986 URLs are valid WHATWG URLs.

We considered adding URL parsers from major browsers (Chrome, Safari, etc.), but we were not able to use them as standalone components. Some of the browsers rely on customized memory allocators and other specialized code that is difficult to remove or isolate. We found other URL parsers, but we believe that the standalone parsers we have selected are representative of the state-of-the-art: all of them are well maintained, reasonably fast, and well documented.

Our benchmark code consumes the URLs taken from large datasets: we ask each parser to normalize the strings. We use Google Benchmarks to derive accurate timings. We also add additional code to capture CPU performance counters (cycles and instructions retired).

We gathered a collection of realistic URLs for benchmarking purposes and we make them freely available.[5]

- The wikipedia 100k dataset contains 100 000 URLs from a snapshot of all Wikipedia articles as URLs (collected March 6th, 2023).
- The top 100 dataset contains 100 031 URLs found in a crawl of the top 100 most popular websites on the Internet. It contains some invalid URLs: 26 URLs according to the WHATWG URL specification are invalid. The curl parser finds 130 invalid URLs whereas the Boost.URL parser identifies 201 invalid URLs. We make freely available the JavaScript software we used to construct this dataset.[6] Fig. 7 presents two histograms regarding this dataset. The first histogram shows that the size of the host in bytes ranges roughly between 10 and 30 bytes, with some outliers. The total size of URL string ranges between a few bytes and hundreds of bytes. Most URLs use between 50 and 100 bytes.
- The Linux files dataset contains all files from a Linux system as URLs (169 312 URLs).
- The userbait dataset contains 11 430 URLs from a phishing benchmark.[7]

In some experiments, we also include another dataset: the kasztp dataset is made of 48 009 URLs from a URL shortener benchmark.[8]

When they are not ASCII, all URLs are processed as UTF-8 strings. The conversion from UTF-16 inputs to UTF-8 would be take little computation [15]. We assume that all inputs are valid Unicode, validation would be similarly

[3] https://github.com/rmisev/url_whatwg

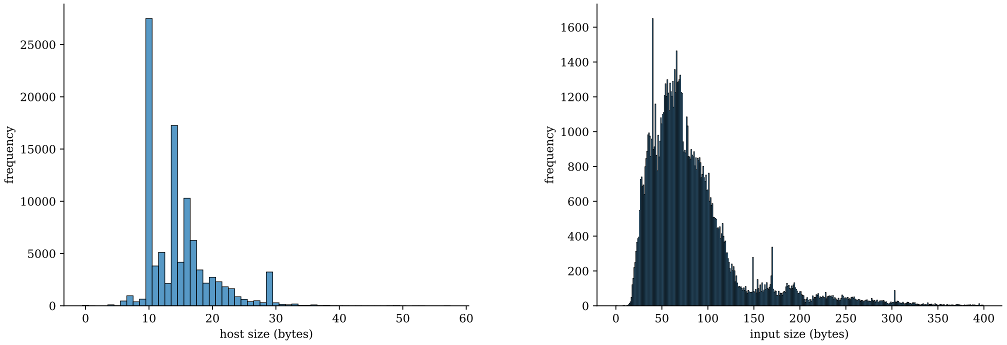
[4] https://servo.org

[5] https://github.com/ada-url/url-various-datasets

[6] https://github.com/ada-url/url-dataset

[7] https://github.com/userbait/phishing_sites_detector

[8] https://github.com/kasztp/URL_Shortener



(a) host-size histogram

(b) total-size histogram)

FIGURE 7 Histograms for the top 100 URL dataset (100 031 URLs)

TABLE 2 Systems

Processor	Frequency	Microarchitecture	Memory	Compiler
AMD EPYC 7262	3.4 GHz	Zen 2 (x64, 2019)	DDR4 (3200 MT/s)	GCC 11
Apple M2	3.0 GHz	Avalanche (aarch64, 2022)	LPDDR5 (6400 MT/s)	Apple/LLVM 14

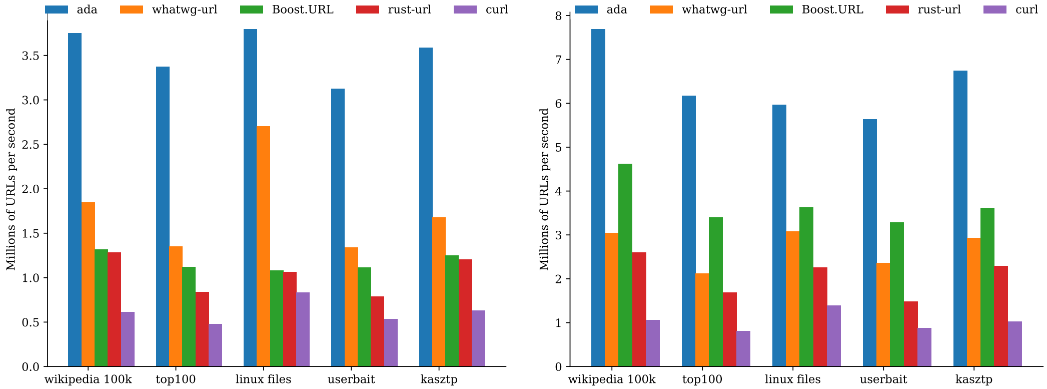
require little computation [16].

We also benchmark URL parsing within JavaScript runtime environments. We used Node.js which can run JavaScript on servers using the Google v8 JavaScript engine. It contains additional code written in C++ and JavaScript. Apart from the popular Node.js runtime environment, we selected two similar environments. Deno resembles Node.js in that it also relies on the v8 JavaScript engine; it is written in Rust instead of C++. Bun is another JavaScript environment but it replaces Google v8 with the WebKit's JavaScript engine (upon which Apple Safari is based). Bun is also written in part with C++ and Zig. We use deno (version 1.32.5), bun (version bun 0.5.9), and Node.js (versions 18.16.0 and 20.1.0). All systems run the same scripts, parsing the URLs from the top 100 dataset. We make our script available.[9]

We run our benchmarks on the two systems presented in Table 2. The AMD server runs Ubuntu 22.04 whereas the Apple processor is on a standard MacBook Air (2022). We monitor the effective frequency and find that the MacBook Air remains at 3.0 GHz whereas the AMD servers maintain 3.4 GHz. We find little variation in the effective frequency between tests (within 1 %). Our benchmark should not be interpreted as an assessment of the performance of ARM versus x64, or of AMD versus Apple. We use different hardware systems, released at different times, to arrive at a more robust comparison of the software.

Fig. 8 gives the number of millions of URLs processed per second for different datasets and different software libraries. Our parser (Ada) dominates, being often twice as fast as other parsers. It is consistently faster than 3 million URLs per second on the AMD system, and faster than 5 million URLs per second on the Apple system. On the Linux files dataset, the WHATWG URL C++ parser has excellent performance on the AMD system, exceeding 2.5 mil-

[9] `https://github.com/ada-url/js_url_benchmark`



(a) AMD Rome (x64)

(b) Apple M2 (ARM64)

FIGURE 8 Millions of URLs processed per second in C++ benchmarks

lion URLs per second. The curl parser is the slowest in our tests: its performance is approximately 0.5 million URLs per second on the AMD system, and nearly 1 million URLs per second on the Apple system.

Similarly, Fig. 9 gives the number of millions of URLs processed per second for different datasets and different JavaScript systems. Node.js 20, with our Ada URL parser has the best performance. However, bun also provides excellent performance, especially on the line files dataset where it comes close to Node.js 20. Roughly speaking, compared to the C++ benchmarks (Fig. 8), the speeds are about half: Node.js 20 is consistently faster than 1.5 million URLs per second on the AMD system, and faster than 2.5 million URLs per second on the Apple system. It suggests that about half of the processing is tied to the JavaScript system, some of it spent in C++, and the rest in JavaScript. The most important difference is between Node.js 20 and Node.js 18 (which lacked Ada): Node.js 20 is four times faster on the Apple system and five times faster on the AMD system. We believe that it is essentially attributable to the replacement of the legacy URL parser by Ada. Node.js went from having the worst performance on URL parsing to the best performance compared to bun and deno.

Table 3 presents the collected performance counters while running the C++ benchmark, while Table 4 has the performance counters for the Apple system. Ada requires consistently fewer instructions than the other parsers. For example, on the top 100 dataset, it required 2200 instructions per URL (AMD) and 2400 instructions per URL (Apple) compared to 18 000 and 19 000 for curl: Ada required eight times fewer instructions.

6 | CONCLUSION

We developed and released a new URL parser that provides full compliance with the WHATWG URL specification. It replaced the legacy Node.js parser, multiplying the performance of URL parsing in Node.js. We believe that our good results can be explained in large part by the following strategies: (1) reduce the number of memory allocations to a minimum, using a single buffer if possible, (2) implement fast functions to check for common fast paths, (3) replace strings with simpler types such as integers whenever possible. Our work suggests that there is still much room for performance improvements in the software used to build web applications.

Future work could consider more advanced techniques. For example, we could design single-instruction-multiple-data (SIMD) algorithms able to benefit from the powerful new instruction sets (e.g., AVX-512, SVE2). We expect that

TABLE 3 Performance counters for Rome system

(a) wikipedia 100k

name	instr./URL	cycles/URL	instr./cycle
ada	2000	910	2.2
WHATWG URL	4800	1800	2.7
Boost.URL	6200	2600	2.4
rust-url	6600	2600	2.5
curl	14000	5500	2.5

(b) top 100

name	instr./URL	cycles/URL	instr./cycle
ada	2200	1000	2.2
WHATWG URL	6700	2500	2.7
Boost.URL	7200	3000	2.4
rust-url	9500	4000	2.4
curl	18000	7100	2.5

(c) Linux files

name	instr./URL	cycles/URL	instr./cycle
ada	2000	890	2.2
WHATWG URL	3700	1200	2.9
Boost.URL	7600	3100	2.4
rust-url	7400	3200	2.3
curl	11000	4100	2.6

(d) userbait

name	instr./URL	cycles/URL	instr./cycle
ada	2100	1100	2.0
WHATWG URL	5600	2500	2.3
Boost.URL	6500	3000	2.2
rust-url	9400	4300	2.2
curl	15000	6300	2.4

TABLE 4 Performance counters for Apple M2 system

(a) wikipedia 100k

name	instr./URL	cycles/URL	instr./cycle
ada	2000	440	4.6
WHATWG URL	5900	1100	5.3
Boost.URL	3600	740	4.9
rust-url	7200	1300	5.5
curl	15000	3200	4.5

(b) top 100

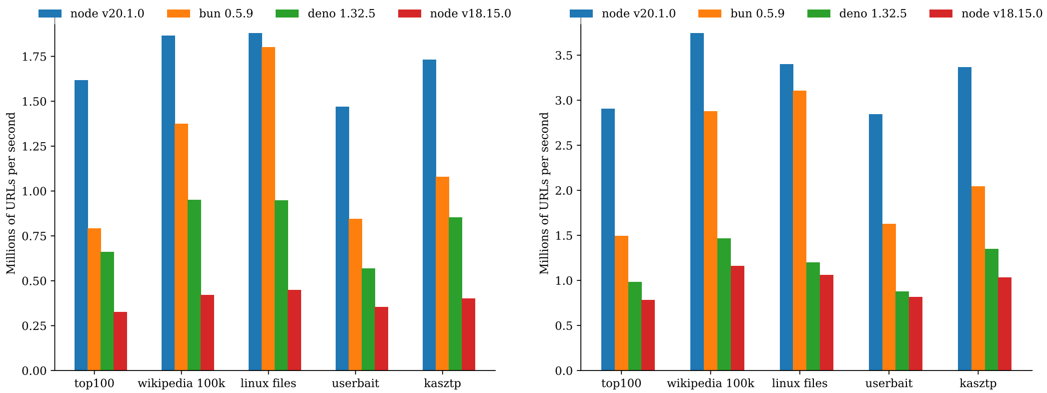
name	instr./URL	cycles/URL	instr./cycle
ada	2400	550	4.5
WHATWG URL	8200	1600	5.1
Boost.URL	4500	1000	4.5
rust-url	10000	2000	5.0
curl	19000	4200	4.5

(c) Linux files

name	instr./URL	cycles/URL	instr./cycle
ada	2400	570	4.2
WHATWG URL	5600	1100	5.1
Boost.URL	4100	920	4.4
rust-url	7900	1500	5.2
curl	12000	2500	4.7

(d) userbait

name	instr./URL	cycles/URL	instr./cycle
ada	2200	600	3.7
WHATWG URL	6700	1400	4.7
Boost.URL	4200	1000	4.1
rust-url	10000	2300	4.5
curl	16000	3900	4.1



(a) AMD Rome (x64)

(b) Apple M2 (ARM64)

FIGURE 9 Millions of URLs processed per second in JavaScript runtime environments

significant gains in URL parsing are still possible. There are other important components of modern web applications that could be optimized.

Author Contributions

Yagiz Nizipli: conceptualization; investigation; software; Node.js integration; experimentation; writing-review and editing. Daniel Lemire: conceptualization; software; validation; experimentation; data analysis; writing-original draft; writing-review and editing.

Acknowledgements

We thank M. Teixeira, R. Nagy, N. Nuon, S. Vohr, M. Atlow, and D. Chatterjee for their software contributions. We thank A. van Kesteren for answering our technical questions on the URL standard. We thank that Node.js team for their feedback.

Data Availability Statement

All our data and software is freely available online. The C++ benchmarking software is available online at `https://github.com/ada-url/ada`. We make the URL datasets available online at `https://github.com/ada-url/url-various-datasets`. We wrote and published a JavaScript benchmark which is available online at `https://github.com/ada-url/js_url_benchmark`. We collected performance data and processed it using Python scripts: both the raw data and the scripts are available online at `https://github.com/ada-url/ada_analysis`.

references

[1] Berners-Lee T, Fielding R, Masinter L, Uniform resource identifier (URI): Generic syntax; 2005. Internet Engineering Task Force, Request for Comments: 3986. `https://tools.ietf.org/html/rfc3986` [last checked May 2023].

[2] Ajmani DK, Koishybayev I, Kapravelos A. yoU aRe a Liar://A Unified Framework for Cross-Testing URL Parsers. In: 2022 IEEE Security and Privacy Workshops (SPW) IEEE; 2022. p. 51–58.

[3] Reynolds J, Bates A, Bailey M. Equivocal URLs: Understanding the Fragmented Space of URL Parser Implementations. In: European Symposium on Research in Computer Security Springer; 2022. p. 166–185.

[4] Lemire D. Number parsing at a gigabyte per second. Software: Practice and Experience 2021;51(8):1700–1727. https://doi.org/10.1002/spe.2984.

[5] Park H, Cha M, Moon SM. Concurrent JavaScript parsing for faster loading of Web apps. ACM Transactions on Architecture and Code Optimization (TACO) 2016;13(4):1–24.

[6] Van Engelen RA. Constructing finite state automata for high performance web services. In: IEEE International Conference on Web Services Citeseer; 2004. .

[7] Kostoulas MG, Matsa M, Mendelsohn N, Perkins E, Heifets A, Mercaldi M. XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization. In: Proceedings of the 15th International Conference on World Wide Web WWW '06, New York, NY, USA: ACM; 2006. p. 93–102.

[8] Cameron RD, Herdy KS, Lin D. High Performance XML Parsing Using Parallel Bit Stream Technology. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds CASCON '08, New York, NY, USA: ACM; 2008. p. 17:222–17:235.

[9] Langdale G, Lemire D. Parsing gigabytes of JSON per second. The VLDB Journal 2019;28(6):941–960.

[10] Muła W, Lemire D. Faster Base64 encoding and decoding using AVX2 instructions. ACM Transactions on the Web (TWEB) 2018;12(3):1–26.

[11] Muła W, Lemire D. Base64 encoding and decoding at almost the speed of a memory copy. Software: Practice and Experience 2020;50(2):89–97.

[12] Faltstrom P, Hoffman P, Costello AM, Internationalizing Domain Names in Applications (IDNA); 2005. Internet Engineering Task Force, Request for Comments: 3490. https://tools.ietf.org/html/rfc3490 [last checked May 2023].

[13] Costello AM, Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA); 2003. Internet Engineering Task Force, Request for Comments: 3492. https://tools.ietf.org/html/rfc3492 [last checked May 2023].

[14] Schmidt DC. In: GPERF: A Perfect Hash Function Generator USA: Cambridge University Press; 2000. p. 461–491.

[15] Lemire D, Muła W. Transcoding billions of Unicode characters per second with SIMD instructions. Software: Practice and Experience 2022;52(2):555–575.

[16] Keiser J, Lemire D. Validating UTF-8 in less than one instruction per byte. Software: Practice and Experience 2021;51(5):950–964.