ORIGINAL ARTICLE

Parsing Millions of DNS Records per Second

Jeroen Koekkoek[1] | Daniel Lemire[2]

[1]NLnet Labs, Science Park 400, 1098 XH Amsterdam, The Netherlands

[2]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-2024-03787

The Domain Name System (DNS) plays a critical role in the functioning of the Internet. It provides a hierarchical name space for locating resources. Data is typically stored in plain text files, possibly spanning gigabytes. Frequent parsing of these files to refresh the data is computationally expensive: processing a zone file can take minutes.

We propose a novel approach called simdzone to enhance DNS parsing throughput. We use data parallelism, specifically the Single Instruction Multiple Data (SIMD) instructions available on commodity processors. We show that we can multiply the parsing speed compared to state-of-the-art parsers found in Knot DNS and the NLnet Labs Name Server Daemon (NSD). The resulting software library replaced the parser in NSD.

1 | INTRODUCTION

The Domain Name System (DNS) was proposed in 1983 by Mockapetris [1, 2]. It provides a distributed hierarchical name space. The name space is organized in a tree structure where each node and leaf have an associated label and resource set. The root domain (represented by a dot, '.') is at the top. Beneath the root, there are top-level generic domains (TLDs) like '.com', '.org', and country-code TLDs (e.g., '.ca' for Canada). The unique combination of labels from the leaf to the root forms a domain name (e.g., `www.example.com.`) that names resources.

Resource records (RRs), simply referred to as record hereafter, are the fundamental units of information in the DNS. A record contains the name, type, class, time to live (TTL), length of the resource data and the resource data (RDATA) itself.

The combination of type and class defines the meaning of the RDATA. Well-known types include the A and AAAA types, which are used to map domain names to one or more IP addresses. Records can be presented in text using the *presentation format* [3]. For example, a record for the domain '`www.example.com.`' might map it an IPv4 address:

```
www.example.com.  3600  IN  A  192.0.2.1
```

We interpret this line as follows:

- Name: `www.example.com.`,
- Type: A (Address),
- Class: IN (Internet),
- TTL: 3600 (1 hour),
- RDATA: 192.0.2.1.

In this entry, `www.example.com.` is the domain name, `3600` is the time to live (TTL) in seconds, `IN` stands for Internet, and `A` indicates that this is an address record containing an IPv4 address `192.0.2.1`. Not all records refer to an IP address as the following example illustrates:

```
example.com.  1800  IN  TXT  "v=spf1 ip4:192.0.2.1 -all"
```

We interpret this line as follows:

- Name: `example.com.`,
- Type: TXT (Descriptive text),
- Class: IN (Internet),
- TTL: 1800 (0.5 hour),
- RDATA: "v=spf1 ip4:192.0.2.1 -all".

The domain name space is partitioned and the responsibility for the content is delegated due to the size and frequency of updates. A subtree for which authority is delegated is called a *zone* (usually per organization, etc). E.g., '`example.com.`' might correspond to a zone managed by a private company. DNS messages are encoded using the binary format defined in RFC 1035 [4, section 4] (henceforth the wire format) when transmitted over the network. However, authoritative DNS data is typically stored on disk in plain text in master files, more commonly referred to as zone files. Zone files and the corresponding text format (henceforth the presentation format) were first introduced in 1987 by RFC 1034 [4, section 3.6.1] and RFC 1035 [5, section 5], but their specification has been extended over time.

Zone files can become large (e.g., tens of gigabytes), and parsing these files can become a performance bottleneck. To accelerate processing, we can use our processors more efficiently. Most commodity processors (Intel, AMD, ARM, POWER) support Single Instruction Multiple Data (SIMD) instructions: these instructions operate on several words at once, using wide registers. For example, most Intel and AMD processors have 256-bit vector registers and they can compare two strings of 32 characters in a single instruction. Recent work on JavaScript Object Notation (JSON) parsing [6, 7, 8], notably in the simdjson library, demonstrates that applying SIMD instructions for parsing structured text can significantly boost performance. JSON has a relatively simple grammar compared to zone files, but there is some conceptual overlap. Can the ideas that made JSON parsing faster be applied to parse zone files faster? We sought to find out by building simdzone, whose name is a play on simdjson, to achieve a similar performance boost for parsing zone data. We find that the in-memory parsing speed of simdzone is nearly a gigabyte per second—at least three times faster than a competitive solution (Knot DNS).

The SIMD instructions improve the speed at two levels in our parser: they allow us to quickly index the content, and they accelerate the parsing of specific components in the processing (e.g., time stamps and Pv4 addresses). To our knowledge, simdzone is the first parser to bring SIMD-based parsing techniques to zone files.

2 | RELATED WORK

To our knowledge, there is no related work on the production of high-performance zone parsers. Much of the academic research regarding zone files relates to security issues. For example, Chandramouli and Rose [9] proposed a validation scheme for zone files. Korczyński et al. [10] discuss insecure zone-file updates which they refer to as *zone poisoning*. Kakarla et al. [11] present SCALE, a method for finding standard compliance bugs in DNS nameservers. They developed a tool called FERRET, which uncovered 30 new unique bugs across various implementations. They examined eight implementations: BIND, NLnet Labs Name Server Daemon (NSD), PowerDNS, Knot DNS, COREDNS, YADIFA, MARADNS, TRUSTDNS. Among the faults they found were server crashes (BIND, COREDNS and TRUSTDNS).

Though some DNS system vendors conduct extensive benchmarks [12], there is relatively little work in the formal literature. One exception is Lencse [13] who presents a performance analysis of four authoritative DNS server implementations: BIND, NSD, Knot DNS, and YADIFA. The benchmark is focused on serving the data rather than parsing [14]. They find that NSD and Knot DNS can achieve an order of magnitude higher performance than BIND and YADIFA.

There is related work regarding the high-performance parsing of Internet formats. E.g., XML parsing has received much attention [15, 16, 17], URL parsing can be accelerated [18], and we can encode and decode base64 data at high speeds [19, 20].

There is also work regarding JSON parsing.: i.e., Langdale and Lemire presented their simdjson library which can parse gigabytes of JSON per second using branchless routines and vectorization [8]. On commodity processors, the simdjson parser was the first standard-compliant JSON parser to process gigabytes of data per second on a single core. Our work builds on simdjson [8], in particular with respect to its indexing stage. The JSON syntax is practically a small subset of JavaScript which preserves the arrays, objects, numbers, strings, Booleans and null values: e.g., `{"name":"Jack","numbers":[11,22]}`). The simdjson parser identifies the starting location of all JSON nodes (e.g., numbers, strings, null, true, false, arrays, objects) and all JSON structural characters (‘`[`’, ‘`{`’, ‘`]`’, ‘`}`’, ‘`:`’, ‘`,`’). The parser stores these locations as an array of integer indexes. Given the JSON document `{"abc":2000}`, we might have the indexes 0, 1, 6, 7, 11. A difficulty is to distinguish the characters that are between quotes: it is solved without branching. The simdjson leverages SIMD instructions to process large blocks (64 bytes) of characters. These blocks are transformed in 64-bit words acting as bitsets from which indexes are extracted. Though the simdjson approach is designed for commodity processors, there are alternative methods on specialized processors [21, 22, 23, 24]. For simdzone, we focus on commodity processors.

In simdjson, the indexing phase covers an entire JSON document. One significant conceptual difference between simdjson and simdzone is that simdzone indexes smaller blocks—and not the whole zone file.

3 | ZONE FILES

Zones are typically stored in text files containing a sequence of records in the presentation format: a concise tabular serialization format with provisions for more convenient editing by hand. While name servers implement slightly different dialects due to the ambiguity of the specification, the presentation format provides sufficient interoperability in practice.

Zone files consist of a sequence of entries made up of a list of white space separated items. An entry is terminated by a newline, but parentheses can be used to continue a list of items across a line boundary. Any line can end with a comment. Comments start with a semicolon (‘`;`’). See Fig. 1. A detailed grammar for expressing items is provided in

Fig. 2.

The following entries are defined by RFC 1035 [5] and RFC 2308 [25]:

```
<blank>[<comment>]
$ORIGIN <domain-name> [<comment>]
$INCLUDE <file-name> [<domain-name>] [<comment>]
$TTL <TTL> [<comment>]
<domain-name><rr> [<comment>]
<blank><rr> [<comment>]
```

Blank lines, i.e., empty lines or lines consisting only of white space characters and/or a comment are ignored. Control entries (i.e., `$ORIGIN`, `$INCLUDE`, `$TTL`) are intended for more convenient editing by hand.

- `$ORIGIN` resets the current origin for relative domain names (domain names without a trailing dot). The directive allows for concise notation of domain names.

```
$ORIGIN example.com.
www A 192.0.2.1 ; becomes www.example.com. A 192.0.2.1
```

- `$INCLUDE` inserts the specified file in the current location. An optional domain name can be provided to define the origin for relative domain names in the file. The directive allows for templating. E.g. records common across zones can be defined in a single file using relative names and included in other files for consistency.
- `$TTL` resets the TTL for records that omit it.

Any entry starting with a dollar sign ('`$`') is best considered a control entry to allow for vendor specific control entries.

The remaining entry types are used to define records. The last provided owner (record domain name) is used if an entry omits it and starts with one or more white space characters.

The `<rr>` section takes one of the following forms:

```
 [<TTL>] [<class>] <type> <RDATA>
 [<class>] [<TTL>] <type> <RDATA>
```

TTL is a 32-bit decimal integer and mnemonics are used to express class and type. TTL and class default to the last provided values if omitted.

The `<RDATA>` section depends on the type and class. A complete presentation of the RDATA section for all record types is beyond our scope, but consider the following examples:

- The RDATA section for an A (Address) record consists of a single IPv4 address. Typically expressed as four decimal numbers separated by dots without interior spaces.

```
www.example.com. A 192.0.2.1
```

- The RDATA section for an NS (Name Server) record consists of a single domain name.

```
example.com. NS ns1.example.com.
```

- The RDATA section for a SOA (Start of Authority) record consists of the domain name for a name server that is the primary source of the zone, a domain name specifying a mailbox, a version number as a 32-bit decimal integer, three 32-bit time intervals in seconds, followed by the minimum TTL field for this zone.

```
example.com. SOA ns1.example.com. hm.example.com. (
  2024071301 ; YYYYMMDDNN
             ; Increased by 1 for changes
  3600       ; Refresh (how often to check for updates)
  600        ; Retry (how often to retry after a failure)
  86400      ; Expire (how long to use zone data)
  3600 )     ; Minimum TTL (how long to cache data)
```

Labels in domain names and text literals (e.g., TXT records) are expressed as character strings and allow for use of escape sequences to include characters that otherwise have structural significance. E.g., to include a dot (‘`.`’) in the label of a domain name it can be escaped (e.g., ‘`\.`’). If the octet is not a printable character a backslash followed by a decimal number describing the octet is allowed too (e.g., ‘`\008`’ for backspace). Values for other field types are expressed using the typical representation, no grammar is provided.

A sequence of bytes may have a different meaning based on location. E.g. if an entry starts with IN it is interpreted as a relative domain name whereas if it is preceded by one or more white space characters, it is interpreted as the mnemonic IN, which denotes the class. If the string `IN` is quoted, it is a character string and is never interpreted as a mnemonic.

RFC 9460 [26] introduces a key-value concept for the RDATA section of Service Binding (SVCB) record type and slightly updates item grammar. Parameters may be specified in any order as either `key=value` or `key="value"`. The key dictates the type of value along with any particular escaping rules. For example, the `port` parameter takes a port number while the `ipv4hint` parameter takes an IPv4 address. Neither `port` nor `ipv4hint` allow character escaping. In contrast, the `alpn` parameter takes a comma-separated list of character strings and hence allows for escaping. Consider the following example:

```
example.com.   SVCB   16 foo.example.org. (
                         alpn=h2,h3-19 mandatory=ipv4hint,alpn
                         ipv4hint=192.0.2.1
                         )
```

Name servers might lack support for some record types. To improve interoperability, RFC 3597 [27] introduces a generic notation. Instead of using a mnemonic, the type can be specified using the word `TYPE` directly followed by the type code. E.g. `TYPE1` and `A` are equivalent. Similarly, the class can be specified using the word `CLASS` directly followed by the class code. E.g. `CLASS1` and `IN` are equivalent as well. The RDATA section in generic notation is specified using the special token \# followed by a 16-bit decimal integer for the length of the RDATA and the RDATA in hexadecimal encoding. For example, to map `www.example.com.` to `192.0.2.1`:

```
www.example.com. 3600 CLASS1 TYPE1 \# 4 C0000201
```

4 | DATA PARALLISM

Commodity processors offer advanced Single Instruction Multiple Data (SIMD) instructions (e.g., SSE4.2, AVX2, etc.) to process multiple data elements simultaneously using a single instruction. While optimizing compilers can make use of them, a deliberate design of the algorithms by the programmer remains needed in most cases. In C/C++, SIMD instructions are available through special functions called intrinsics (see Table 1)

```
$ORIGIN example.com.  ; Defines the origin of the zone

; Free standing @ is replaced by origin
@ 3600 SOA ns1.example.com. hm.example.com. ( 2024071301 ; YYYYMMDDNN
    ; Increased by 1 for changes
    3600 ; Refresh (how often to check for updates)
    600 ; Retry (how often to retry after a failed refresh)
    86400 ; Expire (how long to use zone data)
    3600 ) ; Minimum TTL (how long to cache data)

; Name server records
ns1 86400 NS ns1.example.com.
ns2 86400 NS ns2.example.com.

; Mail server record
mx 10 mail.example.com.

; Web server record
www 3600 A 192.168.1.100
```

FIGURE 1 Example of a simple zone file

```
⟨non-special⟩ ::=  OCTET - NUL, WSP, LF, DQUOTE, ;, (, ), and \
⟨non-digit⟩ ::=  OCTET - NUL and DIGIT
⟨dec-octet⟩ ::=  (0 / 1) 2DIGIT / 2 ((0-4) DIGIT) / (5 (0-5))
⟨escaped⟩ ::=  \ (non-digit / dec-octet)
⟨contiguous⟩ ::=  1*(non-special / escaped)
⟨non-dquot⟩ ::=  OCTET - NUL, DQUOTE, and \
⟨quoted⟩ ::=  DQUOTE *(non-dquot / escaped) DQUOTE
⟨char-string⟩ ::=  contiguous / quoted
```

FIGURE 2 Augmented Backus–Naur form (ABNF) grammar for items in resource records

Most intrinsics are straightforward. For example, the `_mm256_cmpeq_epi8` intrinsic takes two 32-byte registers and outputs a new 32-byte register containing of the result of the comparisons of the inputs. A zero byte indicates that the bytes were different whereas a byte value of 255 indicates that the bytes were equal.

The `vpshufb` instruction provides a vectorized table lookup. It works independently on two lanes of 16 bytes. Given an input register v and a control mask m, it outputs new values $(v_{m_0}, v_{m_1}, v_{m_2}, v_{m_3}, \ldots, v_{m_{15}})$ with the additional convention that only the least significant 4 bits of $m_0, m_1, \ldots$ are considered and that if the most significant bit of m_i is set, then the result is zero (e.g., $v_{128} \equiv 0$). It has low latency (usually 1 CPU cycle) and a high throughput [28]. POWER and ARM processors have instructions similar to `vpshufb` with slightly different conventions.

TABLE 1 Intel AVX2 intrinsics and instructions

intrinsic	bits	instruction	description
`_mm256_or_si256`	256	`vpor`	bitwise OR
`_mm256_and_si256`	256	`vpand`	bitwise AND
`_mm256_cmpeq_epi8`	256	`vpcmpeqb`	compare 32 pairs of bytes, outputting 0xFF on equality and 0x00 otherwise
`_mm256_movemask_epi8`	128	`vpmovmskb`	construct a 32-bit integer from the most significant bits of 32 bytes
`_mm256_shuffle_epi8`	256	`vpshufb`	shuffle two lanes of 16 bytes
`_mm256_loadu_si256`	256	`vmovdqu`	load 32 bytes from memory into a vector register
`_mm256_storeu_si256`	256	`vmovdqu`	write a 32-byte vector register to memory

For parsing tasks, SIMD instructions can be leveraged to compare multiple bytes against a pattern in parallel rather than sequentially. Traditional parsing might involve many if-else statements or loops to check each byte. Using SIMD, we can reduce the number of branches, which are costly in terms of CPU pipeline flushes.

5 | ARCHITECTURE

A complete description of the simdzone parser would be overly technical. Instead, we describe the architecture, and refer the interested reader to our freely available source code.

Like simdjson [8], simdzone works in two stages: an indexing (stage 1) and a parsing (stage 2) stage. Stage one seeks to determine the location of structural characters (blank and special characters as presented in Fig. 5) and the start and end location of each *item* (character-strings and sequences of non-structural characters [5]). At a low level, indexing operates on 64-byte blocks and uses vectorized classification to identify structural characters. The relative indexes are stored in a bitmask and binary logic is used to identify items. The indexes of all structural characters, the first character of each item and the delimiting character for each item are retained. See Fig. 3.

Specifically, stage one proceeds by successively loading 64-byte blocks. The data is loaded in a structure as in Fig. 4 where the input is made of two 32-byte wide registers for the AVX2 instruction set. The 64-bit words act as bitsets, where each bit corresponds to a byte in the input. The `newline`, `backslash`, `quoted` and `semicolon` words are initialized by comparing the 64-byte input with the corresponding ASCII characters (‘`\n`’, ‘\’, ‘`"`’ and ‘`;`’) using fast SIMD comparisons with `vpcmpeqb` followed by a conversion to a 64-bit register with `vpmovmskb`. That is, as little as two instructions are needed to identify the locations of the newlines, backslashes, etc.

These straightforward comparisons are used for characters we must uniquely identify for bit manipulation. Characters in the same class that are equally important are located using *vectorized classification*. We mark the ‘ ’, ‘`\t`’ and ‘`\r`’ with the `blank` word and the parentheses, the null character and newline with the `special` word. The ‘`\r`’ char-

```
example.com.  3600  IN  TXT  "v=spf1 ip4:192.0.2.1 -all"  ; SPF : input data
_______________________________________________________________1: newline
__________________________________________________________1_____: semicolon
__________________________________________________________11111_: in_comment
_____________________________1_________________________1________: quoted
_____________________________11111111111111111111111111_________: in_quoted
____________11____11__11___11_______1_____________1_____11_1____: blank
111111111111__1111__11__111_____________________________________: contiguous
1_____________1_____1___1____1_________________________________1: fields
____________1_____1___1____1___________________________1________: delimiters
```

FIGURE 3 Bit representation of a 64-byte zone file input. After computing the 'fields' value, the indexer would write 0, 14, 20, 24, 29, 63 to the first index array—values corresponding to the location of the 1s in 'fields' and 12, 18, 22, 27, 55 to the second index array—values corresponding to the location of the 1s in 'delimiters'.

```c
struct block {
  simd_8x64_t input;
  uint64_t newline;
  uint64_t backslash;
  uint64_t escaped;
  uint64_t comment;
  uint64_t quoted;
  uint64_t semicolon;
  uint64_t in_quoted;
  uint64_t in_comment;
  uint64_t contiguous;
  uint64_t follows_contiguous;
  uint64_t blank;
  uint64_t special;
};
```

FIGURE 4 Structure in C containing a 64-byte block

acter is not technically part of the class `blank`, we treat it as such for correct length calculation on machines where a newline consists of `\r\n` rather than `\n`.

Specifically, we use the least significant 4 bits of each byte to lookup a byte value in a table and compare the result with the original byte value: if they match then we know that we have marked the characters. We present the tables in Fig. 5. Our generic function is presented in Fig. 6 for the AVX2 case: we identify blank characters by calling `simd_find_any_8x64(input, blank)` and special characters by calling `simd_find_any_8x64(input, special)`. These functions process 64 bytes of input and they return a 64-bit word where the bits are set to identifying the target characters (blank or special).

The rest of the processing follows by doing scalar operations over the words. E.g., we need to compute the bitwise AND NOT with the word indicating the presence of quoted or commented characters. Determining the location of quoted content is branch-free, but comments complicate indexing. If a semicolon appears, a scalar loop is used to correctly discard semicolons within quoted sections and quotes within comments.

When loading blocks of data (i.e., 64 bytes), it is convenient not to have to check for end-of-stream loads—where fewer bytes may be available. Thus, simdzone requires input buffers to be sufficiently large to ensure optimized operations can safely load blocks of input data without reading past the buffer limit. This requirement is of no concern to the user of simdzone when parsing files as simdzone manages input buffers.

The second stage (parsing) processes all items and structural characters. The parsing stage follows the indexing

```
simd_table_t blank = SIMD_TABLE(
  0x20, // 0x00 : " "
  0x00, // 0x01
  0x00, // 0x02
  0x00, // 0x03
  0x00, // 0x04
  0x00, // 0x05
  0x00, // 0x06
  0x00, // 0x07
  0x00, // 0x08
  0x09, // 0x09 : "\t"
  0x00, // 0x0a
  0x00, // 0x0b
  0x00, // 0x0c
  0x0d, // 0x0d : "\r"
  0x00, // 0x0e
  0x00 // 0x0f
);
```

```
simd_table_t special = SIMD_TABLE(
  0x00, // 0x00 : "\0"
  0x00, // 0x01
  0x00, // 0x02
  0x00, // 0x03
  0x00, // 0x04
  0x00, // 0x05
  0x00, // 0x06
  0x00, // 0x07
  0x28, // 0x08 : "("
  0x29, // 0x09 : ")"
  0x0a, // 0x0a : "\n"
  0x00, // 0x0b
  0x00, // 0x0c
  0x00, // 0x0d
  0x00, // 0x0e
  0x00 // 0x0f
);
```

FIGURE 5 Tables used for vectorized classification of the blank and special characters

```
typedef struct { __m256i chunks[2]; } simd_8x64_t;

typedef uint8_t simd_table_t[32];

uint64_t simd_find_any_8x64(
  simd_8x64_t *simd, simd_table_t table)
{
  __m256i t = _mm256_loadu_si256((const __m256i *)table);

  __m256i r0 = _mm256_cmpeq_epi8(
    _mm256_shuffle_epi8(t, simd->chunks[0]), simd->chunks[0]);
  __m256i r1 = _mm256_cmpeq_epi8(
    _mm256_shuffle_epi8(t, simd->chunks[1]), simd->chunks[1]);

  uint64_t m0 = (uint32_t)_mm256_movemask_epi8(r0);
  uint64_t m1 = (uint32_t)_mm256_movemask_epi8(r1);

  return m0 | (m1 << 32);
}
```

FIGURE 6 Generic C function to identify target characters in a block of 64 characters using vectorized classification with AVX2 intrinsics

stage, but the two stages are interleaved. Interleaving is required because zone files can be large. However, records are relatively small and are parsed independently. Small windows that often fit in CPU cache are indexed and parsed successively. As a consequence, we must handle with care partially indexed tokens: we use an adaptive window that grows as needed up to a pre-determined maximum (≈1 MB).

A zone file may contain the $INCLUDE control directive. If such a control directive is encountered, the parser is expected to parse the data in the specified file before parsing the remaining contents in the current file. Indexer state is therefore managed on a per file basis.

Indexes are written to a *tape*, a fixed size, pre-allocated region to avoid allocating memory for each item. While a single tape suffices to implement the logic, delimiting indexes are written to a secondary tape. Deciding if a delimiting index must be discarded introduces extra branches and data dependencies in the hot path. E.g., indexes for delimiting white space characters must be discarded whereas indexes for newline characters have structural significance and must be retained. Writing delimiting indexes to a separate tape allows for discarding the delimiting index unconditionally. During the second stage the item length in bytes is determined by subracting the starting index from the delimiting index. Knowing the length beforehand simplifies parsing the many different data types tha can appear in zone files. E.g., it is not required to account for delimiters in the input when parsing either quoted or contiguous character strings.

⟨*domain*⟩ ::= ⟨*subdomain*⟩ |
⟨*subdomain*⟩ ::= ⟨*label*⟩ | ⟨*subdomain*⟩ . ⟨*label*⟩
⟨*label*⟩ ::= ⟨*letter*⟩ [[⟨*ldh-str*⟩] ⟨*let-dig*⟩]
⟨*ldh-str*⟩ ::= ⟨*let-dig-hyp*⟩ | ⟨*let-dig-hyp*⟩ ⟨*ldh-str*⟩
⟨*let-dig-hyp*⟩ ::= ⟨*let-dig*⟩ | -
⟨*let-dig*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩
⟨*letter*⟩ ::= any one of the 52 alphabetic characters A through Z in upper case and a through z in lower case
⟨*digit*⟩ ::= any one of the ten digits 0 through 9

FIGURE 7 Preferred name syntax from RFC 1035 section 2.3.1

Given the record type, the data has a predefined layout, though records may contain any type of data as long as it does not exceed 65535 bytes in size. This maximum is imposed by the wire format where a 16-bit unsigned integer is used to specify the data length. We use this hard limit to preallocate the output buffer and avoid memory allocations and simplify buffer management.

5.1 | Domain Names

There are many data types. We optimized the processing of the most common ones. In particular, domain names account for much of the data, especially since frequently used records reference other domain names. E.g. RDATA for NS, SOA, MX, CNAME and SVCB records all contain one or more domain names. Much like the indexing stage, parsing of domain names is typically implemented as a scalar loop operating on individual bytes, handling escape sequences, label separators (dots) and label requirements. We must parse the text representation into the wire format [5]: *each label is represented as a one octet length field followed by that number of octets*. If we compare the wire format of domain names to the presentation format, the main difference between the two is that the dots in the presentation format are replaced by the length of the label. E.g. `www.example.com.` is encoded in wire format as `3www7example3com0` (where the digits represent the actual number, not the ASCII equivalent). Most of the data can therefore be copied without modification. While escape sequences may occur, domain names rarely include characters not part of the preferred name syntax. See Fig. 7. Registrants typically select domain names that are easy to remember. Consequently, domain names often exceed 16 bytes, but not 32 bytes. We apply the logic used to optimize the indexing stage to the processing of domain names. We load and store 32 bytes of data unconditionally. We use fast SIMD comparisons to check for '`.`' and '\' in the input and the results are converted to bitmasks. If escape sequences are detected, i.e., the bitmask for '\' is non-zero, the parser unescapes one sequence and starts just after the escape sequence the next iteration. The parser then proceeds to copy and shift the dots bitmask by one to detect null-labels followed by a scalar loop over the indexes to replace the dots in the output by the length of the label.

5.2 | Record Types

Records must contain an identifiable TYPEs (RTYPE) followed by a data payload (RDATA). Typically, given the type, the RDATA layout is predictable. This knowledge can be leveraged to expect the right token and fallback to a slower path in uncommon cases. The optimized (common) path can be inlined while the slower path resides in a function. Binary

```
uint8_t hash(uint64_t prefix) {
  uint32_t value = (uint32_t)((prefix >> 32) ^ prefix);
  return (uint8_t)((value * 3523264710ull) >> 32);
}
```

FIGURE 8 Function used as part of our perfect hash function to recognize resource types

size is hereby reduced as much as possible while token extraction is as fast as can be. This knowledge also allows for calling the correct parser functions in order, eliminating the need for calling more generalized parser functions based on a descriptor table. There is a finite number of RTYPE values: e.g., “A”, “AAAA”, “AFSDB”, “APL”, “CAA”, “CDS”, “CDNSKEY”, “CERT”, “CH”, “CNAME”, “CS”, “CSYNC”, “DHC”, etc. We need to support 67 types. Identifying the type quickly, with minimal branching is important. For this purpose, we designed a perfect hash function [29, 30]. We use a 256-byte table containing either an empty value. Given the input string, we load it into two 8-byte registers—including some possible trailing content. We convert it to ASCII upper case with a binary operation. We set to zero all bytes after the length of the string, by loading a mask from a precomputed table and computing a bitwise AND. We then apply a hash function on the first eight bytes (see Fig. 8) and take the 8-bit value as an index in the 256-element table. The hash function uniquely distinguishes between the existing types: no two type has the same 8-bit value. If the recovered element is empty, we know that the type is unrecognized. Otherwise, we check that we have the correct type by an efficient comparison and we use the resulting code to branch into the corresponding processing. Though our approach assumes that the types are fixed, we can update the code when new types are added with scripts.

5.3 | Binary Data

Binary data is presented in text using some of the conventional binary encoding schemes [31] such base16 or base64. For example, RDATA in generic notation is presented using base16 (hexadecimal) encoding and the signature in RRSIG records is presented using base64 encoding. Encoding and decoding of base64 data can be done at high speeds [19, 20] and prior work on base16 decoding demonstrates similar improvements can be achieved there. To speedup loading of DNSSEC signed zones, we integrated an optimized base64 decoder [32], though only the scalar version. The base64 decoder served as a template for integrating an optimized base16 decoder [33]. We implemented an optimized base32hex decoder for the SSE4.2 and AVX2 instruction sets to improve parsing of NSEC3 records.

6 | EXPERIMENTS

We seek to benchmark the in-memory parsing speed of zone files. To our knowledge, there has been no benchmark of zone file parsing previously. Previous work [13] has focused on data queries—based on an already loaded database. Unfortunately, some popular DNS systems such as BIND and PowerDNS, do not offer a standalone parser: rather they integrate zone file parsing with the construction of the database. Thankfully, two of the fastest and most popular DNS systems (Knot DNS and NSD) have functions that merely parse the data from a file. Like simdzone, they allow us to pass a function as a parameter which receives the parsed records. By passing a trivial function (e.g., one that merely counts records), we can measure the parsing speed itself.

6.1 | Software

For benchmarking, we consider three software packages.

1. Our own simdzone parser[1]. We use the code version identified by the commit tag `aab6386`. It is written in C: it consists in over 35 000 lines of code and about 1000 distinct functions. Of this total, our tests alone consists in nearly 8000 lines of code. The software follows the high-level architecture presented in § 5. Additionally, some processor-dependent code is organized in three different kernel of functions: fallback, westmere and haswell. These kernels are specific to processors having no supported SIMD instructions, to x64 processors equivalent to Intel Westmere processors (16-byte SIMD) and to x64 equivalent or superior to Intel Haswell processors (32-byte SIMD). At runtime, we select the best kernel for the current processor. Further, we have a large set of generic functions which are written in an processor-independent manner, but can be compiled several times, to match different processors.
2. We use Knot DNS version 3.3.4 part of the Ubuntu Linux distribution (version 24.04).
3. We compare against the parser used by NSD up to version 4.9.[2]—prior to its adoption of simdzone. To help reproducibility, we extracted the parser to its own library and we make it freely available.[3] It relies on Flex and Bison. We use the versions of Flex and Bison available as part of Ubuntu 24.04 for benchmarking purposes.

Though details differ, all three software packages are used in a similar manner:

- A parser structure is constructed and initialized. For example, a default TTL is specified (we use 36000).
- A *callback* function is assigned. During the main processing, this function is called each time a record is parsed. This function accepts several values: the name, the type, the class, the TTL and the RDATA of the record.
- We must also assign a filename to the parser. The parser handles file inputs.
- We then call a parsing function which will process the records in sequence and call the assigned *callback* function repeatedly—until the zone file has been completly parsed, or a fatal error is encountered.
- When the parsing is completed, we call a closing function which closes the files.

Our complete benchmarking framework is also freely available.[4] To help reproducibility, we publish as part of benchmark framework a docker file based on the `ubuntu:24.04` image, and we run our benchmarks inside a docker container: docker containers running under a Linux system have practically no overhead except for input/output operations [34, 35]. Our C software is built using the CMake build system. We build the software in release mode using CMake's defaults (`-O3 -DNDEBUG`). The compiler is GCC 13.2—the default compiler under Ubuntu 24.04.

Our software first runs each function once (without timing) and then we run it five times, measuring the average and the minimum time. We include a pure file read function along with the three parsing functions. We verify that the difference between the average and the minimum time is less than 1% in all parsing functions. The non-parsing file reading function has more variation (about 20%), but it is also significantly faster.

6.2 | Benchmarking Method

Zone files are used to persist zone data to disk. While zone files can be copied by means of a file transfer protocol, the presentation format is never used over the network. Thus we seek to measure the speed with which we can deserialize zone files.

A methodological issue is that the zone file parsing functions from Knot DNS, NSD, and our own simdzone, assume that the data resides on disk. Disk performance varies greatly, from a few megabytes per second to gigabytes per second. Some recent disks support speeds above ten gigabytes per second. Network bandwidth varies in a similar

[1] `https://github.com/NLnetLabs/simdzone`
[2] `https://github.com/NLnetLabs/nsd`
[3] `https://github.com/k0ekk0ek/zonec`
[4] `https://github.com/lemire/zone_benchmarks`

TABLE 2 Datasets

zone	bytes	records	comments	date
`.com`	24.7 GB	412 174 597	0	May 3 2024
`.se`	1.4 GB	8 509 650	8	May 6 2024

manner. To avoid benchmarking the specific disk from our benchmarking system, we rely on the fact that our operating system buffers the zone files to memory when reading it more than times. We can verify that disk access is no longer a bottleneck by including a disk-read benchmark as part of our software. Specifically, we verify that we can read the zone files at speeds greater than 5 GB/s which is several times greater than our parsing speed. Alternatively, we could have used a RAM disk.

We deliberately make our results independent from disk performance. Our benchmark is computational: it depends on the performance our the processor. As a side-effect, it makes our measurements more precise: there is little source of variance. In real-world systems, there are other bottlenecks than pure parsing speed—but these considerations are outside of our scope.

6.3 | Data

We picked two important Top-Level Domain (TLD) zone files: `.com` and `.se`. Both are freely available: from the Centralized Zone Data Service (CZDS) and the Swedish Internet Foundation respectively. Table 2 presents the general characteristics of these zone files.

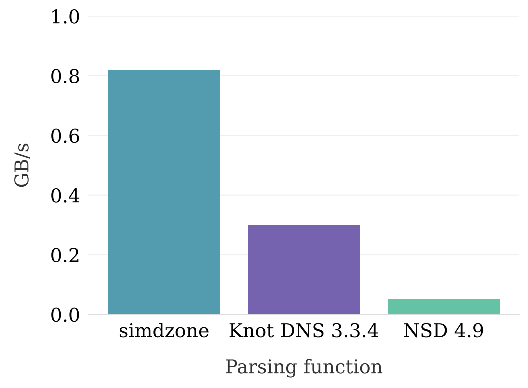
6.4 | Hardware

We conduct our tests on a Linux server with two Intel Xeon Gold 6338 (x64 Ice Lake microarchitecture, 2019) processors with 376 GB of DDR4 (3200 MT/s) memory. These processors have a maximal frequency of 3.49 GHz with a base frequency of 2 GHz. They have 48 MB of last-level cache. Though such a system is capable of running multiple threads at once, all of our code is deliberately single threaded. However, the relatively large amount of memory makes it possible to keep the zone files in cache—practically avoiding disk access. Our processor supports advanced SIMD instructions (AVX-512) but we limit our implementation to AVX2 instructions: AVX2 is available on almost all currently sold x64 processors unlike AVX-512.

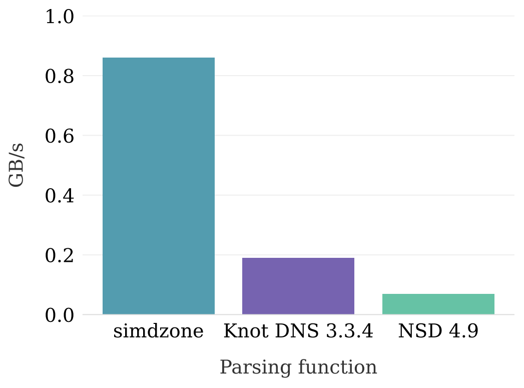
6.5 | Results

Fig. 9 presents the parsing speed for the two zone files. In both instances, the zone parser runs at over 0.8 GB/s. The second fastest parser is Knot DNS with a performance of 0.19 GB/s to 0.32 GB/s: it makes simdzone three to four times faster. The NSD parser has much lower performance, being about ten times slower than simdzone: 0.05 GB/s to 0.07 GB/s.

To help understand our performance, we use performance counters to record the number of instructions retired by byte as well the number of instructions per cycle and the effective frequency. See Table 3. We find that simdzone uses less than half the number of instructions than Knot DNS, and less than one tenth the number of instructions than NSD 4.9. The simdzone parsers also retires more instructions per cycles than Knot DNS, with a slightly lesser



(a) `.com` zone file



(b) `.se` zone file

FIGURE 9 Parsing speed in gigabytes per second

TABLE 3 Performance counters: retired instructions per input byte, instructions retired per cycle, and effective CPU frequency

(a) `.com`

name	instr./byte	instr./cycle	freq. (GHz)
simdzone	12	3.5	2.8
Knot DNS	27	2.6	3.1
NSD 4.9	204	3.3	3.49

(b) `.se`

name	instr./byte	instr./cycle	freq. (GHz)
simdzone	11	3.4	2.8
Knot DNS	23	1.4	3.2
NSD 4.9	127	2.9	3.49

effective frequency. Thus, overall, the simdzone parser requires fewer instructions than a competing parser such as Knot DNS while being able to retire more instructions per cycle—at least in these tests. It matches our expectation: we require fewer instructions to process the same data in part because our software architecture is designed for SIMD instructions. In turn, a single SIMD instruction might carry the work that would require several conventional instructions. Importantly, we reduce significantly the number of needed instructions while maintaining a relatively high rate of instruction execution (instructions per cycle).

Using profiling, we find that the speed of simdzone is limited in part by the `parse_rrsig_rdata` function when parsing the `.se` zone. It is our main function which processes RRSIG records, testing for the presence of various tokens (type, algorithm, TTL, time, base64 sequences, etc.) and parsing them as they are encountered. We find that 44% of all records in the `.se` zone are of type RRSIG. The `.com` zone only contains 5% of records of type RRSIG. In contrast, about half of the records in the `.com` zone are of type NS and the corresponding function (`parse_ns_rdata`) takes a significant fraction of the running time ($\approx$ 15%). Irrespective of the chosen zone file, we find that the function `maybe_take` is a significant burden corresponding to nearly a third of the running time according to our profiling data. This function advances to the next record, while handling open and closing parentheses and taking care of buffer management.

7 | CONCLUSION

Our initial hypothesis is that the good results obtained by the simdjson library for parsing JSON can be generalized to the parsing of zone files. Our results support our hypothesis: we reach parsing speeds that are close to a gigabyte per second. The improved performance comes in part from a reduction in the number of instructions. However, we do not expect that our results are optimal. Further engineering efforts might increase further our performance.

We have adopted a fast base64 decoder. Yet our decoder does not use SIMD instructions. Future work should examine the benefit of SIMD-optimized base64 decoding [19, 20] for zone files.

There are also other directions to explore. ARM-based processors have become more popular on servers: Amazon's Graviton processors are one example. Yet currently simdzone only provides acceleration for x64 processors. Future work should consider the advanced instructions available on ARM processors such as SVE and NEON. SIMD code written for x64 processors does not directly translate into optimally efficient ARM NEON or SVE code due to architectural differences, although some software libraries such as SIMDe may provide a fast conversion. For example, there is no direct equivalent to the `vpmovmskb` instruction.

Furthermore, we should expect that gains are possible if we leverage the AVX-512 instructions available on many server-class x64 processors. The AVX-512 instruction sets not only support wider registers (64 bytes), but they also introduce many new powerful instructions. For example, many of its instructions support masking so that we can load, store or operate on only part of a SIMD register. The new compression/decompression instructions can dynamically prune or insert values (bytes or words) into existing registers. Similarly, though simdzone relies on a single core, future work should consider multicore parallelism [36, 37].

Our work focused on the efficient in-memory parsing of zone files. Since deploying simdzone in NSD, we received comments from some independent users testifying that loading their zone files was much faster.[5] Yet DNS servers have many functions and parsing is just one component. Future work should quantify the benefits of such parsing in the operation of a DNS server. Further, we expect other tools could benefit from adopting simdzone in the future (e.g., `ldns-signzone`, `Unbound`). Future work should quantify the benefits of simdzone within these tools.

references

[1] Mockapetris P, Domain names — concepts and facilities; 1983. Internet Engineering Task Force, Request for Comments: 882. https://datatracker.ietf.org/doc/html/rfc882 [last checked September 2024].

[2] Mockapetris P, Dunlap KJ. Development of the domain name system. In: Symposium proceedings on Communications architectures and protocols; 1988. p. 123–133.

[3] Hoffman P, Fujiwara K, DNS Terminology; 2024. Internet Engineering Task Force, Request for Comments: 9499. https://www.rfc-editor.org/rfc/rfc9499 [last checked September 2024].

[5] https://lists.nlnetlabs.nl/pipermail/nsd-users/2024-April/003304.html

[4] Mockapetris P, Domain names — concepts and facilities; 1987. Internet Engineering Task Force, Request for Comments: 1034. https://datatracker.ietf.org/doc/html/rfc1034 [last checked September 2024].

[5] Mockapetris P, Domain names — implementation and specification; 1987. Internet Engineering Task Force, Request for Comments: 1035. https://datatracker.ietf.org/doc/html/rfc1035 [last checked September 2024].

[6] Bray T, The JavaScript Object Notation (JSON) Data Interchange Format; 2017. Internet Engineering Task Force, Request for Comments: 8259. https://tools.ietf.org/html/rfc8259 [last checked September 2024].

[7] Keiser J, Lemire D. On-demand JSON: A better way to parse documents? Software: Practice and Experience 2024;54(6).

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

[9] Chandramouli R, Rose S. An integrity verification scheme for DNS zone file based on security impact analysis. In: 21st Annual Computer Security Applications Conference (ACSAC'05) IEEE; 2005. p. 10–pp.

[10] Korczyński M, Król M, Van Eeten M. Zone poisoning: The how and where of non-secure DNS dynamic updates. In: Proceedings of the 2016 Internet Measurement Conference; 2016. p. 271–278.

[11] Kakarla SKR, Beckett R, Millstein T, Varghese G. SCALE: Automatically finding RFC compliance bugs in DNS nameservers. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22); 2022. p. 307–323.

[12] DNS Benchmark; 2024. https://www.knot-dns.cz/benchmark/ [last checked September 2024].

[13] Lencse G. Benchmarking authoritative DNS servers. IEEE Access 2020;8:130224–130238.

[14] Georgescu M, Pislaru L, Benchmarking Methodology for IPv6 Transition Technologies; 2017. Internet Engineering Task Force, Request for Comments: 8219. https://tools.ietf.org/html/rfc8219 [last checked September 2024].

[15] van Engelen R. Constructing Finite State Automata for High-Performance XML Web Services. In: Proceedings of the International Conference on Internet Computing CSREA Press; 2004. p. 975–981.

[16] 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.

[17] 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.

[18] Nizipli Y, Lemire D. Parsing millions of URLs per second. Software: Practice and Experience 2024;54(5).

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

[20] 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.

[21] Dann J, Wagner R, Ritter D, Faerber C, Froening H. PipeJSON: Parsing JSON at Line Speed on FPGAs. DaMoN'22, New York, NY, USA: Association for Computing Machinery; 2022. p. 1–7.

[22] Hahn T, Becher A, Wildermann S, Teich J. Raw filtering of JSON data on FPGAs. In: 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE) IEEE; 2022. p. 250–255.

[23] Peltenburg J, Hadnagy Á, Brobbel M, Morrow R, Al-Ars Z. Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. In: 2021 International Conference on Field-Programmable Technology (ICFPT) IEEE; 2021. p. 1–9.

[24] Stehle E, Jacobsen HA. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. Proc VLDB Endow 2020 jan;13(5):616–628. https://doi.org/10.14778/3377369.3377372.

[25] Andrews M, Negative Caching of DNS Queries (DNS NCACHE); 1998. https://datatracker.ietf.org/doc/html/rfc2308 [last checked September 2024].

[26] Schwartz B, Bishop M, Nygren E, Service Binding and Parameter Specification via the DNS (SVCB and HTTPS Resource Records); 2023. https://datatracker.ietf.org/doc/html/rfc9460 [last checked September 2024].

[27] Gustafsson A, Handling of Unknown DNS Resource Record (RR) Types; 2003. https://datatracker.ietf.org/doc/html/rfc3597 [last checked September 2024].

[28] Fog A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Copenhagen, Denmark: Copenhagen University College of Engineering; 2022.

[29] Schmidt DC. GPERF: A Perfect Hash Function Generator. In: More C++ Gems USA: Cambridge University Press; 2000. p. 461–491.

[30] Czech ZJ, Havas G, Majewski BS. Perfect hashing. Theoretical Computer Science 1997;182(1-2):1–143.

[31] Simon Josefsson, The Base16, Base32, and Base64 Data Encodings; 2006. https://datatracker.ietf.org/doc/html/rfc4648 [last checked September 2024].

[32] Klomp A, Fast Base64 stream encoder/decoder; 2024. https://github.com/aklomp/base64 [last checked September 2024].

[33] Galbreath N, Fast c-string transformations; 2024. https://github.com/client9/stringencoders [last checked September 2024].

[34] Fava FB, Laviola Leite LF, Da Silva LFA, Da Silva Amalfi Costa PR, Diniz Nogueira AG, Gobus Lopes AF, et al. Assessing the Performance of Docker in Docker Containers for Microservice-Based Architectures. In: 2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2024. p. 137–142.

[35] Keller Tesser R, Borin E. Containers in HPC: a survey. The Journal of Supercomputing 2023;79(5):5759–5827.

[36] Jiang L, Qiu J, Zhao Z. Scalable Structural Index Construction for JSON Analytics. Proc VLDB Endow 2020 dec;14(4):694–707. https://doi.org/10.14778/3436905.3436926.

[37] Pavlopoulou C, Carman Jr EP, Westmann T, Carey MJ, Tsotras VJ. A Parallel and Scalable Processor for JSON Data. In: EDBT'18; 2018. .