Parsing zone files really fast

Jeroen Koekkoek, FOSDEM 2023



Parsing zone files really fast Or, going from *50MB/s to 700MB/s (and beyond!)

Jeroen Koekkoek, FOSDEM 2023



Motivation

- Zone files can be huge and/or many
 - .com (160,691,699 domains, 23.9GB)
 - .nl (6.288.572 domains)
 - .se (1.464.131 domains, 1.2GB)
- Load times are an issue
 - Database performance plays a role too



Bit on parsing

- Tokenize (Lex) then parse according to grammar (Yacc)
- Useful for context-free languages
- Preprocessor output is compiled

```
#include <stdio.h>
// directives and comments are
// handled by preprocessor
```

```
// keywords, identifiers, string
// literals and integers are always
// parsed as such, regardless of
// scope or position
int main(int argc, char *argv[])
```

```
printf("Hello world!\n");
return 0;
```



Bit on zone parsing

- Not context-free
 - Location defines type
 - Only tokens are strings, newlines and parentheses

\$ORIGIN example.com.
\$TTL 3660

example.com. IN 3600 SOA ns.example.com. hostmaster.example.com. (2023020401 7200 3600 1209600 3600)

; reuse owner (start with blank), class and ttl A 192.0.2.1

; append origin (no trailing .) and reuse class www 3600 A 192.0.2.1

; syntax error (start with blank), but no class or ttl mail A 192.0.2.1

; syntax error (did not start with blank), type is owner A 192.0.2.1



Bit more on zone parsing

- Lex and Yacc make it harder
- Only more-or-less "standardized"



But, why is it slow?

- Lex prefers longest prefix
 - Match multiple expressions (optimized?)
- Copies and unescapes each token
- Splits and rejoins labels
 - (re)allocate, cat, repeat
- Joins encoded data first



But, why is it still slow?

- Dropped Lex and Yacc
 - Fields and order are (sort-of) fixed
- Cut (re)allocations
 - Maximum size 65535 bytes
- Yields around ~180 MB/s (with mmap)



Pipelining

 Fetch, decode, execute and writeback happen simultaneously in various stages



https://en.wikipedia.org/wiki/Pipeline_stall



Pipeline stalls

- Data dependencies introduce a delay in execution, aka stall
- Basically, given a + b = c and c + d = e, the latter cannot be decoded before result of the former is written back
- NOP (no operation) cycles are called "bubbles"



https://en.wikipedia.org/wiki/Pipeline_stall



Pipeline flushes

- Jump instructions, e.g. if statements, may require fetched instructions to be discarded
- Branch prediction is used to improve flow
 - Mispredicted branches require a flush



So, why is it not fast yet?

- State machine is sequential
- Hard to predict branches on user input



Single instruction, multiple data

- Instruction set(s)
 - Vector registers and instructions
- Interest sparked by simdjson
 - Expresses throughput in GB/s
 - Talk by Daniel Lemire https://www.youtube.com/watch?v=wlvKAT7SZIQ
 - Paper by Geoff Langdale and Daniel Lemire https://arxiv.org/abs/1902.08318



Classify blocks, not bytes

Quickly identify 16, 32 or 64 bytes (in a set)



- Repeat multiple times
 - backslash, quote, semicolon, newline, special and space
- Cut branches and dependencies



Vertical, not horizontal

- Single operation, no logic
- Create mask for logic operations





Classify escaped bits

- Find escaped characters
 - Follows odd sequence of backslashes
 - (" \\ \ \\\ ") 0110101110 > 0000010001





Classify quoted bits

- Mask quoted and comment
 - Semicolons in quoted, both in comments
 - Newlines only for comments
 - Hard to solve, branch for comments

("";";";"'n") 1110101111 > 1010100001 > 110011110

Classify contiguous bits

• Bits that remain are contiguous

(""x"yyy;z\n") 110000110 ... 101000000 > 000111000

- Identify transitions
 - Required for zone data, depends on format

(""x"yyy;z\n") 101100101



Transitioning from bits

- Write out transitions
 - Uses fast bit counts
 - Complete tokens only, avoid branches
 - Sliding window, not continuous
- Unlikely branch to defer line count



. . .

- Speedup text to wire conversion for names
 - Scan for non-escaped dots
 - Iterate over indexes
 - Fill in label lengths ((i + 1) i)



Sort-of perfect hash

- First char is primary key
 - Alphabetic, select 16 byte table
- Last char + length is secondary key
 - Alphanumeric (so far)
 - Multiply for good distribution (x + 1 = y)
 - Add length (no clashes so far)
 - Use SIMD compare-equal
- Simply alter "hash" if collisions occur



But wait, there's more

- Numbers and strings
 - Algorithms used in simdjson can likely be adopted
- Base64
 - Wojciech Muła and Daniel Lemire wrote a paper:
 "Faster Base64 Encoding and Decoding Using AVX2 Instructions"
- Hexadecimal
 - Geoff Langdale and Wojciech Muła wrote an article: "Parsing hex numbers with validation"
- IP address conversion
 - "Fastest way to get IPv4 address from string"



So nice, they compile it twice!

- More-or-less. Depends on architecture
 - SSE4.2, AVX2, AVX-512 for x86_64
- Use CPUID to select implementation
 - Do once at start, defeats purpose otherwise



Are we there yet?

- In progress, definitely on right track!
 - Needs polish, working towards a release
- 700MB/s, aiming for 1GB/s
 - Depends on input too
- https://github.com/NLnetLabs/simdzone
 - Standalone, modern C library (BSD-3-Clause)
 - Easy to integrate and contribute



Acknowledgments

Geoff Langdale, Daniel Lemire, simdjson authors and contributors





Questions?

jeroen@nlnetlabs.nl

If you find this interesting, extra pair(s) of eyes and/or hands are always welcome!

