

Hardware Implementations of Finite Automata and Regular Expressions

Extended Abstract

Bruce W. Watson^(✉)

FASTAR Group, Department of Information Science, Stellenbosch University,
Stellenbosch, South Africa
`bruce@fastar.org`

1 Introduction

This extended abstract sketches some of the most recent advances in hardware implementations (and surrounding issues) of finite automata and regular expressions. The traditional application areas for automata and regular expressions are compilers, text editors, text programming languages (for example Sed, AWK, but more recently Python, and Perl), and text processing in general purpose languages (such as Java, C++ and C#). In all these cases, while the regular expression implementation should be efficient, it rarely forms the performance bottleneck in resulting programs and applications. Even more exotic application areas such as computational biology are not particularly taxing on the regular expression implementation — provided some care is taken while crafting the regular expressions [5].

One application domain stands out in its requirement of very high performance — regular expression processing of network traffic. Such processing is required in a variety of contexts: network security (intrusion detection and prevention), protocol detection, policy enforcement, load balancing/traffic differentiation, and quality of service. Given that it usually involves regular expression pattern matching over the network packet ‘payload’, it is often known as *deep packet inspection* (DPI). Currently, all network equipment vendors (and several software vendors) provide DPI products using regular expressions. Despite its age, [11] still gives the best introduction to the algorithmic and implementation intricacies of networks.

Current network speeds at a typical switch are 40 Gbits/s. Full regular expression processing must therefore be done at 4 Gbytes/s after accounting for overheads — one byte per clock cycle on a fast 4 GHz processor. The latency requirements vary per application (e.g. telephony and banking require low latency, while video and music streaming can allow for higher latency provided the variability is low) — meaning that significantly delaying a packet for processing is typically unacceptable. Network packet sizes vary dramatically from hundreds of bytes to tens of kilobytes. Packets from various network flows (e.g. some from a web-browsing session, ftp, mail, and a web application) are interspersed and may arrive out of order, implying that any regular expression

processor must ‘context switch’ appropriate at the beginning and end of a new packet. Lastly, the number of regular expressions (relatively small Perl-like regular expressions) being matched is usually in the range from a few hundred to a few thousand, making it infeasible to deal with them individually. Occasionally, the regular expression set changes, giving the additional challenge of updating the processor, either in batch mode or incrementally when only few of the regular expressions have been edited.

Unfortunately, from a performance perspective, network speed and volume has been outpacing Moore’s law for computational performance.

2 Typical Solutions

As mentioned earlier, Varghese [11] remains an excellent introduction to the algorithmics and implementation aspects of high-performance networking, including regular expression processing. Recently, [14] gives an overview of the latest developments in DPI for networking in virtualized (and cloud-based) environments. Essentially, all solutions share a common set of abstractions grounded in formal languages, and then vary based on implementation.

2.1 Abstractions

While occasional attempts have been made to implement regular expressions directly in hardware¹, most require the ‘compilation’ of the regular expression(s) to some form of finite automaton, with the predictable tradeoffs:

- Nondeterministic automata — requiring space linear in the size of the regular expressions. DPI does not allow for backtracking simulation of the automaton, meaning that all paths are pursued in parallel (processing a byte can take up to time linear in the size of the automaton) and the ‘current state set’ is a significant data-structure overhead which must be stored/restored during context (also potentially taking time linear in the size of the automaton).
- Deterministic automata — requiring space potentially exponential in the size of the regular expression. Processing a byte of network traffic requires a small number of clock cycles (largely independent in the size of the automaton, though memory caching can affect this slightly), as does a context switch.

2.2 Implementations

The above-mentioned abstractions underlie most of the software implementations of DPI². While there is some variation in the CPU speed, cache memory,

¹ Most such attempts decompose the regular expressions in a set of much smaller ones which are then mapped to *content-addressable memory* (CAM) implementations. None of these implementations have yet proven competitive in practice.

² See [13] for one of many treatments of automata and regular expression implementations in software.

etc., eventually all such implementations are outpaced by the network traffic, leading DPI implementers to consider *acceleration* options.

The first option is to use the *graphics processing unit* (GPU) [8]. Numerous such DPI accelerations can be found in the literature (indeed, it appears to be a favourite student project), all showing impressive performance improvements in large packets arriving in-order. The architecture of the GPU (SIMD, meaning that numerous smaller processing elements execute the same instructions in lockstep) and the interface to the CPU (network traffic being transferred over this interface) impair the performance in realistic networks, which involve widely varying packet sizes and frequent context switches. This largely limits GPU accelerations to open-source and software only DPI.

Instead of a general purpose CPU, most network equipment vendors use domain-specific *network processing units* (NPUs)³. Most NPUs have been designed for the breadth of packet processing tasks (routing, packet verification, etc.), with relatively little memory and silicon real-estate devoted to DPI, and such DPI implementations tend to suffer from the same performance limitations as on CPUs⁴.

Any remaining acceleration is only achievable with custom hardware, which broadly falls into two categories: *reconfigurable hardware*⁵ and *application specific integrated circuits* (ASICs). Several vendors provide for FPGA solutions, and the relatively low cost of implementation makes it also an attractive student project [7, Chapter 34]. The regular expression set is usually compiled on a CPU (see [12] for a variety of such compilation algorithms) to an automaton or to circuit structures encoding the automaton, which are then downloaded to the FPGA. The chosen circuit structures are usually optimized for high-speed processing (fewest clock cycles per byte of network traffic), or least silicon real-estate — though the cost of updating the regular expressions is usually high due to the compilation on the CPU and the CPU-FPGA bandwidth for reconfiguring the FPGA.

ASIC solutions typically use a circuit structure resembling a generic automaton (with additional circuitry to simulate it), allowing for rapid updating of the automaton as the regular expressions are changed. As such, the ASIC solution has only a few advantages over FPGAs: higher density and performance, lower volume costs and lower power consumption, but much higher development costs.

2.3 Gaps in Current Solutions

Clearly, all current solutions involve trading off byte-processing time against silicon real-estate, and the ease of updating the regular expression set.

³ See the websites of prominent vendors such as Cisco, Netronome (which took over Intel’s NPU product line) and IBM.

⁴ A notable exception is Netronome’s NPU which includes SIMD processing — in turn having the same performance characteristics as DPI on GPUs.

⁵ In the form of *field programmable gate arrays* (FPGAs).

3 New Implementations

Homogeneous automata⁶ are (not necessarily deterministic) ones in which any given state has in-transitions on the same alphabet symbol (byte). This allows for an efficient encoding of the transition relation — without node labels, as an adjacency matrix — and with a mapping from each state to ‘its symbol’. The bit-matrix and -vector operations (see [13] for implementation details, then in software) map extremely efficiently to digital circuits and will be discussed in detail in this talk. In particular, the bit vectors are linear in the total regular expression size and allow for single clock cycle bit-vector operations to pursue all nondeterministic automaton paths simultaneously. Furthermore, context switching can be done rapidly using burst transfers of the bit-vector to/from memory.

Interestingly, *dual homogeneous* automata⁷ enjoy a similarly compact encoding. The resulting mapping is subtly different from that of homogeneous automata, with occasional circuit real-estate and power savings.

The compilation algorithm mapping a regular expression to a homogeneous automaton is virtually identical to that mapping to a dual homogeneous one. Our most recent work (included in this talk) encodes the compilation algorithm in the circuitry with a minimal overhead. For the first time, this enables an embedded DPI device to be fed regular expressions for compilation directly in silicon — a significant win over first compiling on a CPU and then downloading the automaton (which is typically much larger than the regular expression).

4 Ongoing and Future Work

Brzozowski’s algorithm for constructing a deterministic automaton are both elegant and efficient in practice [3]. Recent work led by Strauss and Kourie [10] has given a parallel version of Brzozowski’s algorithm as *communicating sequential processes* (CSP). Coincidentally, Brzozowski’s career has included lines of research into mapping CSP-like programs to delay-insensitive (unclocked) circuits — see [4], though numerous others have also worked on such mappings and circuitry. This talk also covers the use of such mappings to directly compile Brzozowski’s construction algorithm to a delay-insensitive circuit.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading (1988)
2. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theoretical Comput. Sci. **48**, 117–126 (1986)

⁶ These automata (and variants thereof) were discovered by [1, 2, 6, 9] and are detailed in most treatments of automata construction algorithms.

⁷ Where any given state has *out-transitions* on the same alphabet symbol, see [12] where they are referred to as *reduced finite automata*.

3. Brzozowski, J.A.: Regular expression techniques for sequential circuits. Ph.D. thesis, Princeton University, Princeton, New Jersey, June 1962
4. Brzozowski, J.A., Seger, C.J.: *Asynchronous Circuits*. Springer (1995)
5. Friedl, J.: *Mastering Regular Expressions*, 3rd edn. O'Reilly Media Inc., Sebastopol (2006)
6. Glushkov, V.: The abstract theory of automata. *Russ. Math. Surveys* **16**, 1–53 (1961)
7. Hauck, S., DeHon, A. (eds.): *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, San Francisco (2007)
8. Kirk, D.B., Hwu, W.M.W.: *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufmann, San Francisco (2010)
9. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. *IEEE Trans. Electron. Comput.* **9**(1), 39–47 (1960)
10. Strauss, T., Kourie, D.G., Watson, B.W.: A concurrent specification of Brzozowski's DFA construction algorithm. *Int. J. Found. Comput. Sci.* **19**(1), 125–135 (2008)
11. Varghese, G.: *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, San Francisco (2004)
12. Watson, B.W.: A taxonomy of finite automata construction algorithms. Technical Report 43, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands (1993)
13. Watson, B.W.: The design of the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions. Technical Report 22, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands (1994)
14. Watson, B.W.: Elastic deep packet inspection. In: Brangetto, P., Maybaum, M., Stinissen, J. (eds.) *6th International Conference on Cyber Conflict*, pp. 241–253. IEEE, Tallinn (2014)

Implementation and Application of Automata
20th International Conference, CIAA 2015, Umeå,
Sweden, August 18-21, 2015, Proceedings
Drewes, F. (Ed.)
2015, XXIII, 317 p. 60 illus., Softcover
ISBN: 978-3-319-22359-9