Optimistic Regular Expression Matching on FPGAs for Near-Data Processing

Andreas Becher, Stefan Wildermann, Jürgen Teich
Chair of Hardware-Software-Co-Design, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
<andreas.becher,stefan.wildermann,juergen.teich>@fau.de

ABSTRACT

Regular expressions (regex) are the main means to search for specific patterns in the vast amount of stored textual information. As a consequence, different designs of hardware accelerators have been proposed that enable memory-bound regex processing. Here, the regular expression to be evaluated is translated to a non-deterministic (NFA) or deterministic finite automaton (DFA) which is then mapped onto the hardware design. The available hardware resources of the design imply the maximum size (in terms of amount of states and transitions) of the supported automata. However, regular expressions may be arbitrarily complex.

As a remedy, we propose optimistic regular expression evaluation which follows the idea of pruning the DFA of a regex such that it fits the available hardware resources. Consequently, we obtain a DFA with not only matching states but also an uncertain state. Texts marked as uncertain have to be re-evaluated by software. This is particularly tailored to near-data processing where the optimistic regex evaluation is performed near the data source thus reducing the overall amount of data to be transmitted to the requesting host.

A prototype is implemented within Google's RE2 meaning a complete coverage of RE2 supported regular expression for the proposed design. Regular expression evaluation of up to 2.66 GByte/s could be achieved on an FPGA-based Zynq SoC.

CCS CONCEPTS

• Computer systems organization → Embedded systems; **• Information systems → Query operators**; **• Applied computing → Document searching**; **• Hardware → External storage**;

KEYWORDS

FPGA,SoC,Regular Expression

ACM Reference Format:
Andreas Becher, Stefan Wildermann, Jürgen Teich. 2018. Optimistic Regular Expression Matching on FPGAs for Near-Data Processing. In *DaMoN'18: 14th International Workshop on Data Management on New Hardware, June 11, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3211922.3211926

1 INTRODUCTION

In modern data processing systems, regular expression matching is of great importance in order to leverage the amount of textual information generated everyday. Consequently, multiple approaches have been proposed to provide hardware accelerators for memory-bound text processing [5, 7, 9]. They all follow the idea of translating the regular expression to an NFA which is then mapped onto the accelerator. Due to hardware restrictions, only automata with a maximum amount of states and transitions fitting the available resources are supported. However, as generally arbitrarily complex queries can be expressed, hardware accelerators tend to be oversized or unable to process the query at hand.

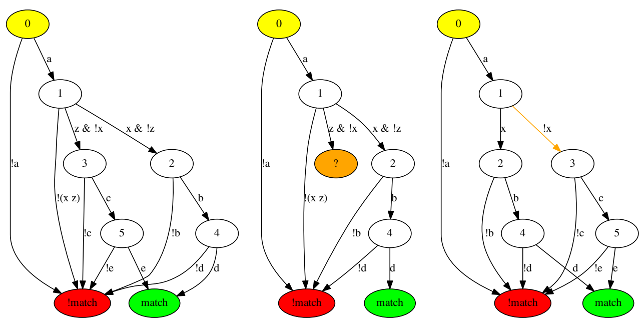
In this paper, we propose optimistic regular expression matching. It is tailored to *near-data processing* where parts of computations are moved towards the data sources for filtering the data before evaluation on the requesting host system [3]. The target is to pull the (sparsely stored) data of interest out of the sea of big data and only transmitting an information-rich subset to the host system. We propose to prune the DFA of an arbitrary regex such that it fits the available resources of the near-data hardware accelerator. The accelerator can thus filter out data which do not match the expression before transmission to the processor. Still, the data stream may contain data where it is uncertain whether it matches or not so that the host still has to evaluate the received data. Nonetheless, this approach has the potential of significantly reducing data transfers between memory and processor [1] as one dominant factor of power consumption. Furthermore, host systems can now be better utilized by avoiding or at least decreasing the processing of irrelevant data. As data processing in dedicated hardware is usually more energy-efficient than on general-purpose hardware, near-data processing has been proven to reduce the overall energy demands in comparable setups; e.g., in [2] where hash-value computation is performed near-data to significantly reduce the data for hash joins.

In the following, we introduce the concept of optimistic regular expression matching, present results obtained for an FPGA-based prototype integrated into Google's RE2, before discussing the future work.

2 OPTIMISTIC REGULAR EXPRESSION MATCHING

Many software libraries utilize DFAs or NFAs for evaluating regular expressions, where regular expressions are expressed as a number of states and conditional transitions between them. These Finite State Machines (FSMs) can be implemented in hardware in numerous ways. As automata change with each regular expression, it is crucial that the hardware can be configured accordingly. For example, [4] presents the common implementation possibilities for ROM/RAM-based FSMs. While such implementations allow the configuration of the current automaton during run-time, the resources of the accelerator are fixed during synthesis time. This has impact on the number of transitions an automaton can have and/or the overall amount of states. Even when choosing synthesis parameters to utilize the maximum amount of available hardware resources, regular expressions may be arbitrarily complex and thus not be supported by the accelerator. On the other hand, the hardware may be underutilized for less complex expressions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DaMoN'18, June 11, 2018, Houston, TX, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5853-8/18/06…$15.00
https://doi.org/10.1145/3211922.3211926



(a) Initial **(b) Tranformation I state removal** **(c) Transformation II transition removal**

Figure 1: Tranformation of a DFA for the regex $a(xbd)|(zce)$.

Our idea of pruning the state machine is illustrated by a motivational example. Figure 1a depicts a state diagram for the regular expression $a(xbd)|(zce)$. In the following, we introduce two transformations for pruning a state machine to fit hardware restrictions.

The first transformation depicted in Figure 1b is to *reduce the amount of states*. In this example, we assume that our synthesized RAM-based FSM can only handle 7 states. The automaton can be fit to the hardware by, e.g., pruning states 3 and 4 and redirecting the transition $z\&!x$ from state 1 to a newly introduced state *uncertain* (?).

The second transformation illustrated in Figure 1c is to *reduce the number of outgoing transitions per state* to fit the hardware constraints. In this example, a single comparison is supported per state resulting in two outgoing transitions– one for `condition` and one for `!condition`. State 1 has three transitions $(x, z, !(xz))$. By merging conditions $!(xz)$ and $z\&!x$, the amount of transitions can be reduced. Furthermore, x is the only character which has to be matched and, consequently, a configuration of the available hardware is feasible. This merged transition is marked as an *uncertain* edge (orange in Figure 1c) and indicates that the result of this evaluation is uncertain and the whole string will be evaluated on the host again. However, if *!match* is reached, the expression is definitely not matched by the input string and thus does not have to be transmitted.

It is important to note that while these transformations limit the capability for the transformed state machine to match **every** input string correctly, some strings are still evaluated correctly or at least a *match* can be excluded. This allows to filter the input strings near the storage before they are sent to the host. Optimistic regular expression matching can therefore be used to filter strings according to regular expressions even with reduced resource requirements.

3 IMPLEMENTATION

We integrated our proposed pre-filtering with Google's RE2 [6] engine. RE2 is used to parse the regular expression and build the initial DFA. Our extension adds the *uncertain* state if too many states are present and applies the first transformation to reduce the amount of states for fitting the near-data accelerator. Likewise, the second transformation is applied and *uncertain* transitions are generated if the amount of transitions is too high.

Our hardware prototype is based on a regular RAM-based FSM implementation [4] and adopted to mark *uncertain* states and transitions. The accelerator produces two output bits for each processed input string: *MATCH* and ?. Strings resulting in the uncertain result

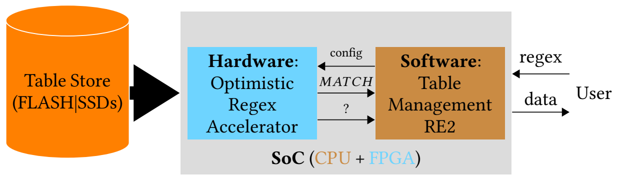


Table Store (FLASH|SSDs)
Hardware: Optimistic Regex Accelerator
config
MATCH
?
Software: Table Management RE2
regex
data
User
SoC (CPU + FPGA)

Figure 2: Proposed near data regex filtering system.

(?) have to be re-evaluated using the original DFA by our extended RE2 engine. Also if the regular expression is used to extract matches, strings evaluated with $MATCH = true$ will have to be reprocessed. However, the strings resulting in $MATCH = false$ can be safely ignored and are thus not handed over to the host. Figure 2 depicts a high level overview of such a near data regular expression filtering system.

The filter hardware has been implemented in a cost-optimized Xilinx Zynq XC7Z020 SoC with a maximum of 128 states. Moreover, the hardware is based on four parallel character matchers each testing one transition condition. Thus, 4 transitions per state are supported. The character matchers are able to match sequential characters (ranges) whereas the first and last matching character is state dependent. 32 individual values can be selected for each starting and ending character. By also utilizing the second port of dual port BRAMs available in the FPGA, two input streams can be processed concurrently without increasing the amount of total BRAM usage. 8 parallel, pipelined engines can therefore handle 16 input characters each clock cycle, leading to 2.66 $\frac{GBytes}{s}$ throughput at 166 MHz clock frequency. This throughput is sufficient to handle tables located at four parallel SATA-III 6G connected drives at line rate, thus imposing no performance penalties for the host while providing all near-data processing advantages of data reduction.

4 CONCLUSION AND FUTURE WORK

Near-data processing in hardware for regular expression matching has great potential to improve the overall power consumption and energy demands as well as processor utilization of big-data systems. This paper introduces the concept of *optimistic regular expression matching* to prune regular expressions to fit the available resources of the hardware. The accelerator can thus filter out irrelevant data before transmission to the host. Due to the pruning, some data is labeled as *uncertain* and still has to be re-evaluated on the SoCs processor or at the host. We have integrated the concept as an extension to Google's RE2 utilizing an FPGA-based prototype for near-data processing. Early results on evaluating common queries [1][8] show a great potential for the technique of optimistic regular expression matching if the transformation are applied at the right state/transition.

In future work, we will elaborate on further extensions and evaluations: (a) Hardware concepts that are tailored to optimistic regex, which may enable further hardware optimizations as long as approximated computations produce output that is labeled as *uncertain* and re-evaluated at the host. (b) Software concepts that perform state and transition pruning in an optimized fashion with the goal of increasing the coverage of correctly labeled data. (c) Thorough evaluation of the potential, benefits and also limitations of the approach based on benchmarks and real-world data.

[1] URI (protocol://server/path): $([a-zA-Z][a-zA-Z0-9]*) : //([\hat{}\ /]+)(/[\hat{}\ ]*)?$
Email (name@server): $([\hat{}\ @]+)@([\hat{}\ @]+)$
Date (month/day/year): $([0-9][0-9]?)([0-9][0-9]?)/([0-9][0-9]([0-9][0-9])?)$
URI|EMAIL: $([a-zA-Z][a-zA-Z0-9]*) : //([\ /]+)([\hat{}\ ]*)?|([\hat{}\ @]+)@([\hat{}\ @]+)$

REFERENCES

[1] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. 2014. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro* 34, 4 (July 2014), 36–42. https://doi.org/10.1109/MM.2014.55

[2] A. Becher, D. Ziener, K. Meyer-Wegener, and J. Teich. 2015. A co-design approach for accelerated SQL query processing via FPGA-based data filtering. In *2015 International Conference on Field Programmable Technology (FPT)*. 192–195. https://doi.org/10.1109/FPT.2015.7393148

[3] Edward Doller, Ameen Akel, Jeffrey Wang, Ken Curewitz, and Sean Eilert. 2014. DataCenter 2020: Near-memory acceleration for data-oriented applications. In *Symposium on VLSI Circuits, VLSIC 2014, Digest of Technical Papers, Honolulu, HI, USA, June 10-13, 2014*. 1–4. https://doi.org/10.1109/VLSIC.2014.6858357

[4] I. Garcia-Vargas, R. Senhadji-Navarro, G. Jimenez-Moreno, A. Civit-Balcells, and P. Guerra-Gutierrez. 2007. ROM-Based Finite State Machine Implementation in Low Cost FPGAs. In *2007 IEEE International Symposium on Industrial Electronics*. 2342–2347. https://doi.org/10.1109/ISIE.2007.4374972

[5] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783747

[6] Google Inc. 2018. RE2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in PCRE, Perl, and Python. It is a C++ library. (March 2018). Retrieved March 26, 2018 from https://github.com/google/re2

[7] Z. István, D. Sidler, and G. Alonso. 2016. Runtime Parameterizable Regular Expression Operators for Databases. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 204–211. https://doi.org/10.1109/FCCM.2016.61

[8] Heng Li. 2010. Benchmark of Regex Libraries. (March 2010). Retrieved March 26, 2018 from http://lh3lh3.users.sourceforge.net/reb.shtml

[9] R. Sidhu and V. K. Prasanna. 2001. Fast Regular Expression Matching Using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*. 227–238.