2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines

Runtime Parameterizable Regular Expression Operators for Databases

Zsolt István David Sidler Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zürich
{firstname.lastname}@inf.ethz.ch

***Abstract*—Relational databases execute user queries through operator trees, where each operator has a well defined interface and a specific task (e.g., arithmetic function, pattern matching, aggregation, etc.). Hardware acceleration of compute intensive operators is a promising prospect but it comes with challenges. Databases execute tens of thousands of different queries per second. Thus, if only one specific instantiation of an operator is supported by the accelerator, it will have little effect on the overall workload.**

In this paper we explore the tradeoff between resource efficiency and expression complexity for an FPGA accelerator targeting string-matching operators (LIKE and REGEXP_LIKE in SQL). This tradeoff is complex. For instance, the FPGA not always wins: simple queries that can be answered from indexes run faster on the CPU. On complex regular expressions, the FPGA is faster but needs to be parametrized at runtime to be able to support different queries. For very long patterns, the entire expression might not fit into the FPGA circuit and a combined mode CPU-FPGA must be chosen. We evaluate our design on a heterogeneous multi-core machine in which the FPGA has cache coherent access to the CPU memory. In addition to the string matching circuit, we also show how to implement database page parsing logic so as to be able to work directly on the same memory data structures as the database engine.

I. Introduction

Relational databases are fundamental to large applications and their design is an excellent example of balancing high performance and specialization. User queries are expressed in *Structured Query Language* (SQL), which provides the abstraction of operators (selection, aggregation, joins, etc.). The query optimizer turns queries into tree-like execution plans and chooses from different implementations of operators at runtime according to complex heuristics and cost functions. The performance of queries executed often with different input parameters can be increased with *prepared statements*, i.e. by preparing the operator tree beforehand and filling in parameters at execution time. This is how we approach FPGA accelerators in this paper: the type of operation to be performed is fixed but the actual parameters are only filled in at runtime.

One example of an operator that is very compute intensive in software and could benefit from acceleration is the selection of records based on a regular expression (using the LIKE and REGEX_LIKE operators in SQL). Software solutions on CPUs, or even novel solutions on GPUs [1], become quickly compute bound as the complexity of the expression increases. FPGAs have long been used to speed up regular expressions in networking, mostly for intrusion or event detection. Since in those cases the regular expressions do not change frequently, compiling them directly into non-deterministic finite automaton (NFAs) on the FPGA is a common approach. The seminal work of Sidhu and Prasanna [2] from 2001 has been the foundation for most solutions following afterwards. By contrast, databases do not operate on a fixed set of regular expressions. Instead, expressions change frequently, e.g., with each query. Another difference is that on a given data input, only a single or a few regular expressions are evaluated rather than hundreds. Finally, the notion of line-rate is different as the solution needs to match or come close to the memory speed instead of network rates.

While promising, the use of an FPGA accelerator might not pay off in every case. Our experiments show that if a query can be answered through an index lookup (in principle all "contains", "starts-with" and "ends-with" expressions) the software solution will be better than the FPGA which has to scan over the whole table. This drives the decision of focusing on complex regular expressions and, by the analogy of prepared statements, we want to use the FPGA not only for a single instance of a query but for a large number of queries. This means that the circuit needs to be runtime parameterizable while still allowing for arbitrary, complex, regular expressions.

One additional requirement is that the FPGA needs to be able to work directly on the same data structures as the database engine. The overhead of copying and partitioning data has been an impediment to the wider adoption of accelerators (regardless of their type) – even though their nominal performance benefits are impressive. With this in mind, our work targets hybrid multi-core machines where the FPGA is connected to the same memory as the CPU via a cache coherent interconnect (two examples being Intel and Altera's Heterogeneous Architecture Research Platform and IBM's CAPI-based FPGA extension cards). These architectures make it possible for the accelerator to work on the same memory pages and data structures as the database engine and, as a result, deliver better performance without having to reformat or partition the data.

This paper addresses the requirements outlined above, i.e., 1) runtime parametrization to be able to speed up a wide range of queries, 2) handling complex expressions to improve the cases where the database is very slow, and 3) compatibility with database data structures so that the data does not need to be reorganized. The paper makes the following contributions:

- We identify the types of regular expressions in databases that could be sped up by FPGAs, and also give contrasting examples of queries that can make use of indexes and therefore are faster in the database engine.
- We design a module that supports POSIX Extended Regular Expressions and is runtime parameterizable with little

978-1-5090-2356-1/16 $31.00 © 2016 IEEE
DOI 10.1109/FCCM.2016.61

overhead. For this we revisited ideas from the state-of-the-art and created a design well suited to the database-specific requirements outlined above.

- A spectrum of possible configurations is explored, including a hybrid hardware-software option to overcome the inherent space limitation on the FPGA. Specifically, we compare resource consumption as a function of maximum complexity and also target bandwidth. This information is relevant for future use-cases where the FPGA is shared between different operators and the query optimizer needs to divide the available resources, or when using the accelerator on a machine with different memory characteristics.

II. Background

A. Hybrid Multicore Architecture

The system used in this paper has an experimental Hybrid Multicore Architecture (HMA) and was made available through the *Intel-Altera Heterogeneous Architecture Research Platform* program [3]. As shown in Figure 1, the HMA is a dual socket machine[1], having a 10-core CPU (Intel Xeon E5-2680 v2) in one socket and an FPGA (Altera Stratix V 5SGXEA) in the other. The CPU socket has 96 GB of memory which can be accessed by the FPGA over QPI. On the FPGA socket no memory is installed. The FPGA has no other interfaces than QPI.

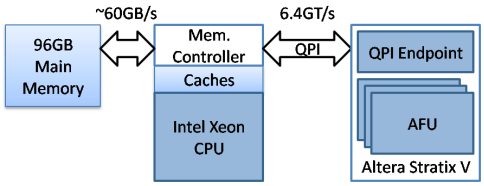


Fig. 1: Hybrid Multicore Architecture

In contrast to most other systems with accelerators, the FPGA has direct, cache coherent access to the main memory through the QPI bus and interacts with it at 512 b (cache line) granularity. We measured the memory bandwidth fetching pages of 4 MB over QPI to peak at 5.1 GB/s, lower than what the nominal transfer rate would allow for. Since the FPGA has to access memory located on a different socket, its bandwidth is naturally lower than that of the CPU which has memory on its own socket and can achieve up to 25 GB/s (note that this is also less than the theoretical maximum shown in Figure 1).

Intel provides an encrypted IP module which contains the QPI endpoint and a cache on the FPGA. This IP can be configured to expose either a physical or virtual memory address-based interface to the user logic. The former allows accessing multiple pages of up to 4 MB each, while the latter provides a single 2 GB memory region which can be shared between FPGA and CPU. For virtual addressing, the on-chip-cache is 64 KB direct-mapped and for physical addressing it is 128 KB two-way associative. We opted for the physical addressing option because our implementation is memory-bound and from the two options physical addressing gives higher bandwidth.

On the software side, Intel provides an *Accelerator Abstraction Layer* (AAL) as a library to interact with the FPGA. The operating system (Ubuntu 14.04) is extended with a special kernel module to communicate with the FPGA. First, when instantiating an application that uses the FPGA, a handshake between the hardware and software verifies that the correct *Accelerator Functional Unit* (AFU) is deployed in hardware. Then a so called Device Status Memory (DSM) page is allocated to share control and status information between the software and hardware. After the DSM has been set up they can communicate directly using shared memory data structures.

B. Database Operators

Fig. 2: A simplified example of how databases build operator trees based on user queries

Logically a database table looks like the example shown in Figure 2. Each entry is represented by a row with multiple attributes. Our example assumes that the shipping address is not broken up into multiple attributes (first name, last name, street, city, state), instead it is a single variable length string. In databases, queries are compiled by the query optimizer into a query plan that is, in essence, a tree of operators where each operator executes part of the query. Our simple example query in Figure 2 is converted into a tree with two operators: `Filter` and `Min`. At the leaf of the tree is the database containing the data which is stored across multiple pages. The `Filter` operator selects the rows in the table which satisfy the condition `LIKE 'SEATTLE'` (using a regular expression matcher logic) and forwards those to the `Min` operator which will find the row with the minimum value iteratively and extracts the `Value` attribute. This number is then the final output of the query.

Real world queries are often much more complex than our example and the task of the query optimizer is to use the available operators and combine them in an execution tree that optimizes the estimated execution time. To make the combination of operators feasible, all work on the same well-defined data layout and have similar interfaces. In this work we envision a database which has operators implemented in software as well as hardware and the query optimizer can pick any of them to build an optimal operator tree. Thereby the operators implemented in hardware have to adhere to the same data layout as their software counterparts.

There are multiple ways databases lay out their data, ranging from row-oriented to column-oriented structures. One common trait among database systems is that they perform their memory management independent from the OS and

[1]Following the Intel legal guidelines on publishing performance numbers we want to make the reader aware that **results in this publication were generated using pre-production hardware and software, and may not reflect the performance of production or future systems.**

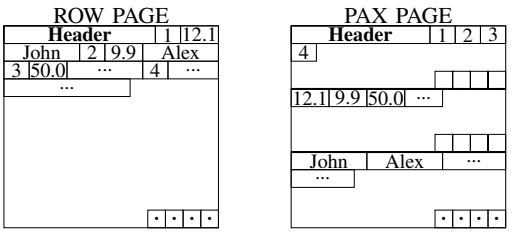


Fig. 3: Example data, in ROW page and PAX page

usually group records into pages. In this work we decided to use the PAX storage layout [4], which splits the rows in a page into its attributes and rearranges them in mini-pages inside the same logical container, as illustrated in Figure 3. This layout combines two characteristics: inter-record spatial locality and low record reconstruction cost. For our solution the first characteristic is important because, as we will later show, our regular expression circuit is memory bound, so reading only the attribute of interest makes sure that the bandwidth is optimally utilized.

III. Design and Implementation

Our FPGA solution relies on the technique of converting regular expressions into a non-deterministic finite state automaton (NFA). The benefit of using NFAs is that multiple states can be active at the same time, and input characters can trigger many different state transitions in constant time – a behavior that is not possible in software. The first difference that sets us apart from most related work is that in the database domain regular expressions are generally used for search on (natural) text, in contrast to pattern matching on packet content. Natural text contains often longer words or names. This might lead to high state counts even in NFAs, since every character translates to a state. Take for instance the following expression that looks for a person's first and last name separated by multiple spaces or tabs and then followed by the city name:

```
Samuel[ \t]+Smith.*London
```

It is clear that in these types of expressions, the number of total characters is much larger than the actual states required to express the matching logic. This particular expression can be represented with "tokens" as AB+C.*D. Therefore in our design we decouple character matching from the NFA state transition logic and allow a sequence of characters to be matched as a single token. Ranges (e.g., [0-9]) are just special characters, and can also be part of a token. This decoupling inside the regular expression matcher directly defines the two *synthesis-time parameters* of the circuit: a) the maximum number of characters it can identify and b) the maximum number of states that the NFA can contain. The actual number to use at runtime and the characters to associate with each token, as well as the state transitions are all *runtime parameters*.

Overall we target *POSIX Extended Regular Expressions* (we will refer to these simply as regular expressions) and this work was inspired by related work such as [5] and [6] (for a detailed comparison to these and other related work, see Section V), with the difference that in our system the regular expressions can be quickly modified on the FPGA and that through compression the number of NFA states can be significantly reduced.

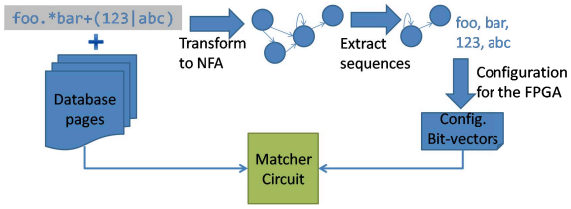


Fig. 4: The hardware circuit is parametrized at runtime and works on memory pointers received from software

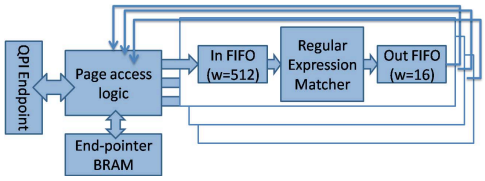


Fig. 5: Working on multiple strings in parallel to achieve higher throughput

The deployment of a particular instance of a regular expression to the hardware is shown in Figure 4. First, the regular expression is transformed into an NFA representation using a Perl library[2]. Then, using a C++ program, we extract from this NFA all character sequences to be represented by tokens and compact the corresponding states into single states. Based on this compressed form of the NFA and the character sequences, we derive the configuration for the FPGA (the exact format of this configuration is explained later). In our current implementation we used the header of the PAX page to specify the regular expression configuration for the data items contained in the page. Since each page is possibly of a different size and representing a different user query, runtime parametrization is a necessity. To start processing, the software passes the pointer to the PAX pages to the hardware, which will navigate the internal structures on its own.

We designed the regular expression matcher to process 1 byte per cycle. To achieve higher throughput we deploy multiple matchers in parallel and distribute the input data using 512 bit-wide FIFOs (Figure 5). The pointers in the PAX page can be used to determine the length of the strings (we use 16 bits to store the length in bytes) and this information is used to split the input into different strings. The outputs are collected in 16 bit-wide FIFOs, read out in a round robin manner, and written into a memory page. Each output value is a 16 bit unsigned integer, in case of a match it is the byte-position of the end of the match, otherwise it is zero.

The core of our implementation is the regular expression matcher module that works on the input one character at a time and is built from three parts: Input Manager, Tokenizer (containing the character matchers) and State Blocks (the NFA states). Figure 6 provides an overview of an example matcher with up to 4 tokens and 4 states (we chose these sizes for simplicity, the evaluation setups are more complex). The Input Manager unit is responsible for converting from 512 bit input (width of the QPI bus) to single characters, and feeding these to the Tokenizer. It also counts how many characters have been processed and, whenever a match happens, it will output

[2] search.cpan.org/~loic/Regexp-ERE-0.01/

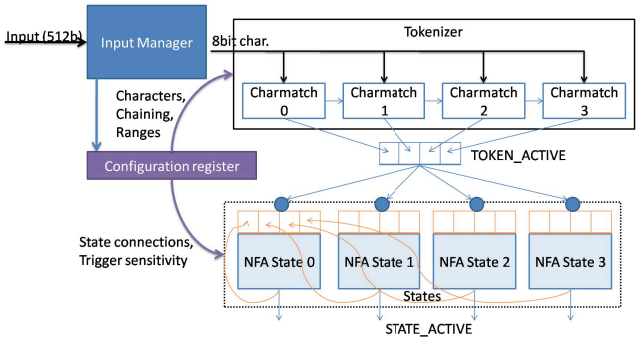


Fig. 6: Structure of a regular expression matcher engine

the character location of that match. Additionally, if a match happens, it will skip the remainder of the string and fast-forward the input to the beginning of the next string. As seen in Figure 6, the Input Manager is also responsible for managing runtime parametrization. It treats the first inputs after a soft reset as configuration words (our metadata is in the header of the PAX page, that is read first) and will load them into a wide register (if necessary by shifting multiple words into this register). Once the configuration register has been loaded, it will enable the configuration mode of all character matchers in the Tokenizer and will set up state transitions.

The Tokenizer unit is built using a series of Character Matchers. These have two implementations, shown in Figure 7 and alternate in sequence (in Figure 6, blocks 0 and 2 are type L, and 1 and 3 are H). This is so that they can be either used one by one as character matchers (simple equality) or as a pair to implement range comparisons. In the Tokenizer a sequence of character matchers is mapped to a Token by ignoring the $active$ output for all but the last character in the sequence – so in some sense the Tokens only exist in the way the circuit is parametrized at runtime. The Tokenizer block has a delay of one clock cycle and can consume an input character every cycle. The output is a bit-vector concatenated from each character matcher's output.

The Tokenizer is parametrized using the following data (C denotes the number of character matchers and S denotes the number of states in the expression matcher):

- **CHAR_CONTENT**: $C * 8$ bits, defining the character to match against for each character matcher block.
- **IS_CHAINED**: C bits, indicating if the character is part of a sequence, therefore only matches if the preceding character also matched in the previous cycle. The first character in a sequence will have this not set.
- **IS_RANGE**: $C/2$ bits, indicating if a pair of matchers is used for comparing the input to a range.

Each character matcher runs the following logic:

```
char_match[t] := (CHAR_CONTENT[t]==in_char || (IS_H_TYPE[t]
    && IS_RANGE[t/2] && char_smaller[t-1]==1 &&
    CHAR_CONTENT[t]>in_char))
char_smaller[t] := (CHAR_CONTENT[t]<=in_char)
token_active[t] := (char_match[t] && (!IS_CHAINED[t] ||
    was_active[t-1])
```

The states are represented as very simple logic blocks that in essence have a single bit of state (whether they are active or not). The output of the Tokenizer is received every cycle as a bit-vector (TOKEN_ACTIVE) and the states are updated all at once, synchronously. To perform this update the following parametrization data is used:

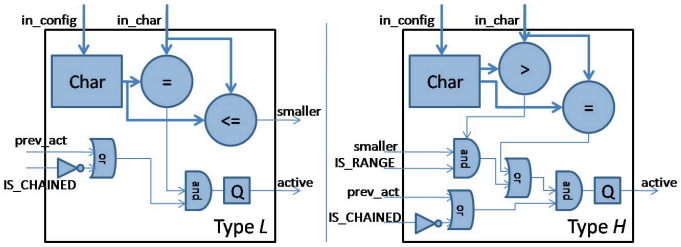


Fig. 7: Detailed view of the character matcher logic (type L on left, type H on the right)

- **STATE_TRIGS**: $S * C$ bits, a vector for each state that encodes which tokens (i.e., character matcher outputs) the state is sensitive to.
- **STATE_DELAY**: $S * \log_2(C)$ bits, the length of the token the state is sensitive to. If there are multiple tokens of different length that should trigger the same state, the software will create additional states.
- **STATE_INEDGE**: $S*S$ bits, a vector encoding which states can transition to state_i (incoming edge-based encoding).
- **STATE_STICKY**: S bits, if a state is sticky then it has a .* edge onto itself. This is an optimization to save states for expressions with wildcards between sequences.

These vectors are used in the following way to determine at the next cycle what states will be active:

```
may_activate[s] := (TOKEN_ACTIVE & STATE_TRIGS[s] != 0) ||
    STATE_STICKY==1 || STATE_TRIGS[s]==0
has_active_inedge[s] := (STATE_ACTIVE & STATE_INEDGE[s] !=
    0) || STATE_INEDGE[s]==0
will_activate[s] := may_activate[s] && has_active_indedge[s]
```

The states also have a small shift register inside them which delays the incoming activation signals using a shift register as many cycles as the sequence takes to be detected for the given state. This is configured using the STATE_DELAY vector.

The regular expression is considered to be a match if the state with the highest index becomes active. The configuration software always maps the accepting state to the state with the highest index. If there are multiple accepting states, these will be either merged together (if their input transitions are all conditioned by the same token) or an additional state will be introduced with unconditioned transitions to the new acceptance state.

It is clear that, in the current setup, one will always be able to create a regular expression that does not fit in the available space. It would be possible to extend this architecture with a configuration register that can hold multiple configuration words and on partial match of the regular expression the circuit can re-parameterize itself with the next word. This way, long regular expressions can be tiled into multiple shorter ones and still be executed in hardware. Since the cost of reloading the configuration is only 2 clock cycles, the processing could be resumed quickly. It is clear, however, that in this case the design will potentially waste logic in the average case in the expectation of a very complex regular expression.

TABLE I: Patterns used for evaluation

	Pattern	Complexity	Use case
P_1	'P\.O\. Box'	low	DB
P_2	'Next.*Day.*Shipping'	medium	DB
P_3	'a(REQIMG|RVWCFG)b'	medium	Snort
P_4	'Max-dotdot[\n]*[0-9]{3,}'	medium	Snort
P_5	'(P\.O\. Box|PB).*(87[0-9]{4})'	high	DB
P_6	'SITE[\t\r\n\v\f]+NEWER'	high	Snort

As the alternative, and more resource sparing solution, we handle the occasional regular expression that does not fit on the FPGA by dividing it between hardware and software. Since the software gets for each string a pointer to the end-position where the regular expression matched on the FPGA, it can simply resume the search at that location for its own part of the expression. While there is some limitation on what regular expressions can be split this way between software and hardware, cutting the regular expression at a ".*" symbol will always lead to correct results.

IV. Evaluation

The regular expression matcher described in the previous section was implemented on the HMA machine introduced in Section II-A. The QPI endpoint provides a 200 and a 400 MHz clock. Since the QPI interface exposed to user logic runs at the former clock speed, our default clock rate for the regular expression matcher is also 200 MHz, and we deploy 32 parallel matchers to provide a 6.4 GB/s throughput for the circuit. As will be shown, this is more than 25% higher than the QPI bandwidth in this machine.

As a baseline comparison for string matching, we chose two high performance software regular expression engines: Intel Hyperscan and Google RE2, and a widely adopted commercial database (for legal reasons we will call this system DBx). Each experiment was repeated 10 times for software and 100 times for hardware and the average value is reported. We also measured the standard deviation and found it to be very small and consistent throughout the experiments. The software and hardware regular expression engines are both operating on the same PAX page layout explained earlier in Figure 3. For most experiments we used a size of 4 MB for this page. DBx uses a similar, but proprietary, format to store data in pages. To minimize the overhead of record parsing or reconstruction, DBx operates on a relation with a single column of type VARCHAR which contains fixed-length strings.

For our evaluation we used natural text in the form of address strings. Each string contains names, street, city, and area code, generated at random and concatenated into a single string. The length of the strings was chosen explicitly for different experiments, and they were either shortened or concatenated together to have a predefined fixed length (unless explicitly stated, it is 64 B). For each regular expression we used in our Evaluation we inserted a fixed amount of "hits" into the strings uniformly at random (20% unless otherwise stated). The actual contents of the strings are only relevant to ensure that character frequencies follow a distribution similar to real-life data.

Table I shows the regular expressions used in the experiments. We use three patterns that resemble queries in the TPC-W [3] benchmark. Additionally, as a reference to related work, three representative patterns from the Snort [4] community rule set were also added. As seen in the table, we classify the expressions into three complexity classes, based on how many regular expression features they use. P_1 and P_2 can be expressed in databases using LIKE, but the others require the more general REGEXP_LIKE. Only P_1 can be answered with an index lookup.

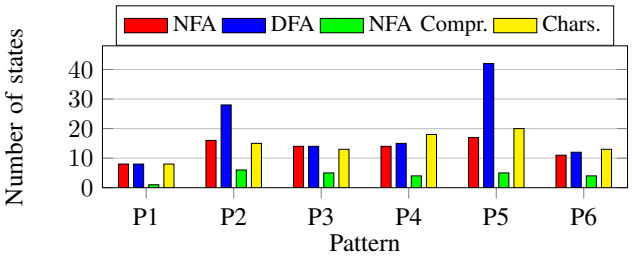


Fig. 8: Number of NFA vs. DFA states for each pattern and the benefit of compressing NFAs through character extraction

A. Compressed NFAs

As a first experiment, we looked at the proposed set of regular expressions and aimed to answer two questions: what is the benefit of sequence extraction on the final NFA complexity, and what is a good common configuration of the regular expression matcher to conduct the rest of the evaluation. Figure 8 shows how many NFA states are required to implement each pattern, if one character corresponds to one state. It also shows the equivalent DFA, and then the compressed NFA states, and the number of characters needed in total to evaluate the pattern (the number of tokens is smaller than the characters). Interestingly, the DFAs only suffer from "state explosion" for certain patterns. Namely P_2 and P_5, both contain an unlimited wildcard repetition '.*'. Further, we conclude from this experiment that the extraction of sequences is a useful optimization for these kinds of expressions as it reduces the number of states significantly (NFA Compr.). As a result of this experiment, our default configuration of the hardware circuit handles 8 states and up to 20 characters.

We also measured the time it took to derive the bit-vectors for runtime parametrization for P_1 through P_6. Depending on the complexity of the pattern, the Perl script produced the NFA in 40-240μs, and the C++ code ran typically in the 40-50μs range. These numbers are relatively low. Compared, for instance, to scanning 4 MBs of main memory over QPI that takes around 800μs, they are adequate as proof of concept (real database tables are typically larger than that). But in our prototype some of the work is duplicated between the Perl script and the C++ code, and as a result the overhead could be reduced. Since for Hyper and RE2 the time to parse the regular expression and create an internal representation is also high we report both their and our runtime without this overhead.

B. Performance

The first performance experiment we conducted answers two questions: how does the software performance compare

[3] www.tpc.org/tpcw/
[4] www.snort.org/downloads/

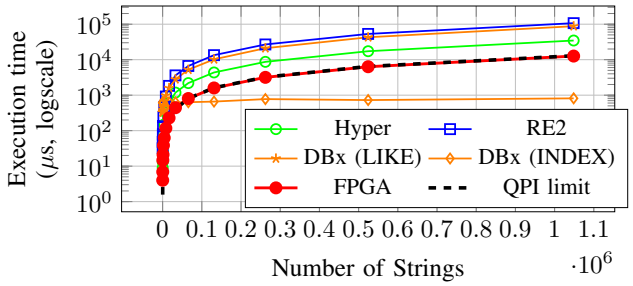


Fig. 9: Time to process a fixed number of strings using scanning and index techniques (P_1)

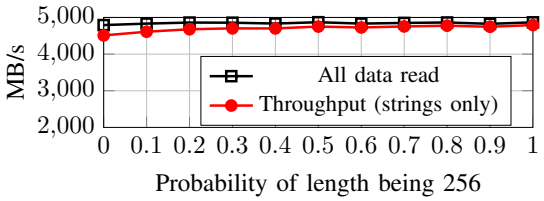


Fig. 10: FPGA reads strings with length 64 and 256 mixed randomly (P_5)

against the hardware for the simplest pattern (i.e., where the software is the fastest), and what characteristics can we expect from the QPI link in our evaluation machine. We evaluated P_1 on database tables containing 16 to 1 Mio. strings. We chose the simplest pattern because the database can build an index for it and the software shows the best performance. Figure 9 shows that an index clearly outperforms any approach that has to scan over the whole data set, be it hardware or software. On the other hand, if there is no index available, the hardware implementation will always be faster than software.

When comparing to recent work on GPUs [1], the FPGA is slower for simplistic patterns, but it is using its memory bandwidth more efficiently. We measured 0.4ms on the FPGA vs. the 0.1ms reported on the GPU for 32k strings 64 B each. It is important to highlight that the memory bandwidth available for the GPU is more than ten times higher than our system, yet it is not proportionally faster. Also, the solution does not extend to general regular expression patterns, only simple substring lookups (i.e., only P_1 and P_2 in our examples).

This experiment also shows that the latency of accessing the hardware from software is in the range of 1-2 μs (we measured the internal processing time on the FPGA, and subtracted it from the execution time measured in software). The FPGA's throughput comes very close to the peak bandwidth of the QPI bus. The overhead comes from transferring metadata, such as page headers and pointers, and the write-back of the results to memory. These are not included in our reported throughput. Since our circuit has been provisioned for 6.4 GB/s it is clear that the execution is bound by the QPI bus.

To illustrate the bottleneck on the QPI bus, and also to show that our circuit is flexible in terms of string length and mixed workloads, we ran P_3 on the FPGA with different mix of string sizes. On one end all strings were 64 B long, while on the other end all were 256 B. In the middle they were randomly mixed. PAX pages contain both pointer and string data and in

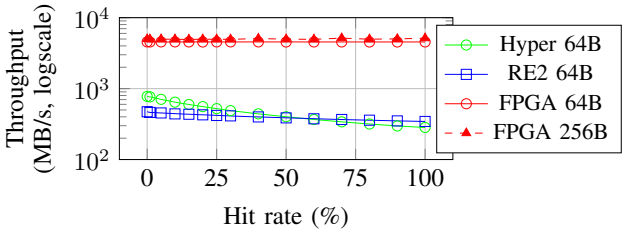


Fig. 11: Impact of hit percentage with P_3

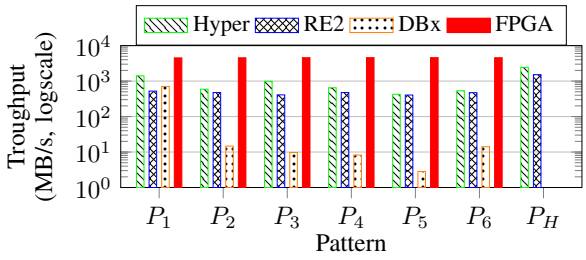


Fig. 12: Throughput as function of pattern complexity

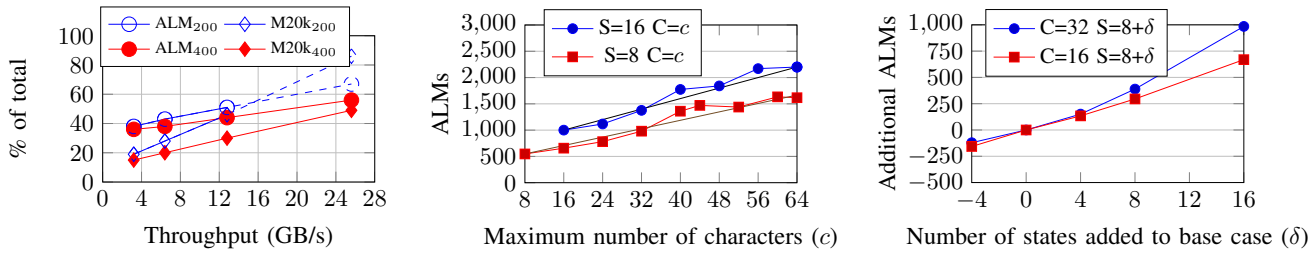
Figure 10 we show processing throughput both in terms of the total amount of bytes moved through QPI for processing the 4 MB page, and also in terms of useful throughput (i.e., strings only). As can be seen, the circuit has a constant speed, bound by the QPI bus, regardless of the string size mix.

The regular expression matcher units inside the FPGA are designed to consume input at a constant rate regardless of string length or regular expression complexity. Figure 11 shows that throughput stays constant with increasing hit rate both for 64 and 256 B strings. On the other hand, software slows down in case there are more matching strings in the input because it becomes increasingly more compute bound. This effect intensifies with more complex expressions.

As a final comparison to software (Figure 12), we illustrate how the increasing complexity of the regular expressions impacts the software solutions (DBx uses LIKE for P_1 and P_2, and REGEXP_LIKE for the rest). The FPGA can consume input at a constant rate, regardless the regular expression complexity, and stays at the QPI bandwidth bottleneck explained before. The software libraries are single threaded and run on a single core, but even with perfect linear scaling for 10 cores they will be slower than the FPGA for all but the simplest pattern. In Figure 12 there is an additional pattern, P_H. This is a concatenation of P_5 and P_1, where the first part is ran in hardware, and then the software continues matching P_1 on strings that are a partial match. We measured the performance of this hybrid setup both when using Hyper and RE2 for post-processing, and plot the results. The hybrid performance is very competitive even when compared to the simplest pattern in software (one reason might be that the software can skip ahead in the strings to the location of the match directly instead of re-scanning everything).

C. Resource Usage

In Table II we show how many resources each individual building block needs, and also the space required by our default



(a) The full regular expression operator (C=20, S=8) with the QPI IP for different throughput targets at 200 and 400 MHz

(b) ALMs used for a single regular expression matcher, with varying character matchers in the Tokenizer

(c) The additional cost in ALMs of adding states to a C=16,S=8 and a C=32,S=8 regular expression matcher module

Fig. 13: Resource consumption

TABLE II: Resource consumption of basic building blocks

Component	Comb. ALUTs	Logic Reg.
Character matcher (L)	17	13
Character matcher (H)	13	11
Single state	10	23
Reg.Ex. S=8, C=20	*908*	*1028*
Page access logic	*3529*	*2111*

configuration with 8 states and up to 20 characters. While these numbers could be reduced by hard-coding the patterns in that case the accelerator would no longer be as well-adapted for databases which demand fast runtime parametrization.

For future systems we envision two trends: 1) The memory bandwidth of the FPGA will be increased, 2) dynamic scheduling of operators on the FPGA and possibly sharing it between multiple operators. Therefore, it is important to explore how different target throughput levels translate to resource consumption numbers, and the other way around. In Figure 13a we show how much space the operator uses when configured to match different throughput levels. This is reported together with the QPI endpoint, that needs a fixed 31% of the ALMs and 7% of M20k memories. Adjusting throughput is done by changing the number of parallel matchers, or by clocking the circuit up. It has to be mentioned that the interface to the QPI logic is clocked at 200 MHz and cannot be changed. This means that for the 25.6 GB/s case where our circuit is clocked at 400 MHz the QPI still runs at 200 MHz, so in principle this setup would be bottlenecked by the input (512 b/cycle). Nonetheless, it shows that our logic is quite scalable and could immediately benefit from a higher memory bandwidth.

Finally, in Figures 13b and 13c we illustrate the effect of adding more characters or states to the regular expression matchers. Here we report the numbers as the required ALMs reported by the tool for an individual regular expression matcher unit (excluding the FIFOs, but including the parametrization logic), They do not consume any M20K blocks, so we omitted these from the graph. It is visible in the figures that adding characters (Figure 13b) leads to a linear increase in resources, whereas states (Fig. 13c) have an exponential effect. In Figure 13b the slope increases slightly when there are more states in the regular expression matcher. This is not very surprising as the size of the bit-vectors used to transfer "active" states between character matchers and states is dependent on both sides. As explained earlier, thanks to our tokenization of character sequences, the compacted NFAs can easily fit in 8 or 16 states for most real world DB workloads. One take-away is that if there is free space on the chip, adding characters might be a more useful way of extending the circuit.

V. Related Work

A. Regular Expressions and Network Intrusion Detection

A large body of related work uses regular expression matching on FPGAs for network intrusion or event detection, e.g., [2], [5], [7], [8]. The use of these solutions in databases could be advantageous, however as we explained in the introduction, the assumptions and requirements made in existing work focused on networking. Network approaches are mostly designed for a pre-defined line-rate of 10 to 40 Gbps and their goal is to compare many concurrent patterns to the input stream. In databases, the goal is to optimally utilize the available memory-bandwidth, which generally is in the range of 30 to 60 GB/s. Although in our setup the FPGA is limited by the QPI bandwidth, we showed in the previous section that the regex matchers could process data at up to 25.6 GB/s (204.8 Gbps). Another difference to network solutions is that only a single or a few regular expressions are evaluated rather than hundreds, defined at run-time by the users.

Considerable work has been devoted to increasing the throughput of regular expression matchers on FPGAs, and one way of doing this is by accepting more than one character per cycle. Examples of such work are: [5], [9], [10], [11]. One approach for multi-character processing is building an NFA with multi-character transitions. This leads to an exponential increase in the number of states in the NFA. For that reason further techniques have to be applied to compact the state space and keep the resource consumption at a feasible level. In our work we decided to achieve higher throughput through data parallelism by deploying multiple regex matchers.

There is also work that uses more special purpose hardware, such as TCAMs [12] and specialized microprocessors [13], [14]. While these have promising performance, in our work we require more flexibility because we envision the FPGA as an accelerator on which many different database operators can be deployed dynamically, the regular expression being only one.

B. Parameterizable/Flexible Designs

There are different ways that other projects implemented runtime parametrization or re-loading of regular expressions.

For instance, Eguro et. al. [15] propose dynamic reconfiguration to trade off performance for capacity. Thereby virtualizing the fabric and time-multiplexing the computation of multiple complex patterns on the same chip. Other works employ very similar methods to our solution in terms of runtime parameterization by loading registers (or BRAMs) with character and/or state transition data at runtime. In [6] the authors use a Tokenizer that is similar to what we have used, but in their domain the NFA is fixed and only the transition parameters are configurable. In the work of Kaneta et. al. [16] the states transitions can be changed at will, but the characters are processed separately (no compaction of sequences) and the resulting NFAs can not capture some of the features of regular expressions (for instance the | operator for two alternate possibilities, e.g. `(a|b)`). The authors of [17] implement a similarly flexible system, but they use a DFA-based representation, which only allows a state transition every 2 cycles while our NFA can consume a character every cycle.

C. Database Acceleration

Related work in the field of database acceleration with FPGAs is often limited by the requirement of explicit data movement. While the need for true shared memory between accelerator and CPU is not a new one, it is only recently that commercial systems with such capabilities appeared (e.g., our evaluation system, IBM's CAPI accelerators or the Convey HC-2). An example of DMA-based solutions is the work of Ueda et al. [18] that pushes join algorithms to the FPGA that can access the host memory using DMA over PCIe. In addition to implementing two join algorithms, they also dynamically reconfigure the FPGA with the algorithm more suited to the current input data. Recent related work [19] explored the use of selection, merge join and sorting operators on FPGAs. Their system only accesses the local DDR memory on the FPGA boards. While it benefits from the high local memory bandwidth, it suffers however from data partitioning and in contrast to our system data is not directly accessible by the host for consistent modification. Further work [20] exploring hash joins on FPGAs uses simulation to evaluate their implementation, neglecting data movement questions.

Other related work uses the FPGA as a "bump in the wire" to implement database operators [21], [22], [23]. While our work operates on data in main memory which is common for databases, these works place the FPGA into the data path between CPU and the durable storage for tables. As a result they remove the problem of data movement and benefit form line-rate processing capabilities of the FPGA, but at the same time limit the types of operations they can accelerate. In hybrid architectures where the CPU and FPGA have similar memory access there are more offloading opportunities.

VI. Conclusion

In this work we made the case for runtime parameterizable designs for FPGAs in the context of databases. We illustrate the benefit of acceleration for regular expression matching operators, and also explain why the access overhead and configuration overhead of the FPGA had to be minimized to make a difference. Additionally, we show that the target for hardware acceleration should be moderate to complex regular expressions, because in the simple cases the database will always win using an index. We see this work fitting well with other efforts of database acceleration, and believe that the steps taken in identifying the benefits and challenges, and then re-imagining the state of the art for regular expression matching provides lessons that can be used for implementing other operators as well.

Acknowledgment

We would like to thank Intel for the generous donation of the hardware platform as part of the Hardware Accelerator Research Program. Part of this work has been funded by a grant from the MSR-ETHZ-EPFL Joint Research Center.

References

[1] E. Sitaridi and K. Ross, "Gpu-accelerated string matching for database applications," *PVLDB*, Nov. 2015.

[2] R. Sidhu and V. Prasanna, "Fast regular expression matching using FPGAs," in *FCCM'01*.

[3] N. Oliver, R. Sharma, S. Chang *et al.*, "A reconfigurable computing system based on a cache-coherent fabric," in *ReConFig'11*.

[4] A. Ailamaki *et al.*, "Data page layouts for relational databases on deep memory hierarchies," *PVLDB*, vol. 11, no. 3, Nov. 2002.

[5] Y. Yang, W. Jiang, and V. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in *ANCS'08*.

[6] J. Teubner, L. Woods, and C. Nie, "Skeleton automata for FPGAs: reconfiguring without reconstructing," in *SIGMOD'12*.

[7] J. Bispo, I. Sourdis, J. M. Cardoso *et al.*, "Regular expression matching for reconfigurable packet inspection," in *FPT'06*.

[8] L. Woods, J. Teubner, and G. Alonso, "Complex event detection at wire speed with FPGAs," *PVLDB*, vol. 3, no. 1-2, Sep. 2010.

[9] B. C. Brodie *et al.*, "A scalable architecture for high-throughput regular-expression pattern matching," in *ISCA '06*.

[10] C. Clark and D. Schimmel, "Scalable pattern matching for high speed networks," in *FCCM '04*.

[11] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," in *FPL'08*.

[12] C. R. Meiners, J. Patel, E. Norige, A. X. Liu, and E. Torng, "Fast regular expression matching using small tcam," *IEEE/ACM TON*, vol. 22, no. 1, Feb 2014.

[13] J. Van Lunteren *et al.*, "Hardware-accelerated regular expression matching at multiple tens of Gb/s," in *INFOCOM'12*.

[14] C. Sabotta, "Advantages and challenges of programming the micron automata processor," *Iowa State University Master Thesis*, 2013.

[15] K. Eguro, "Automated dynamic reconfiguration for high-performance regular expression searching," in *FPT'09*.

[16] Y. Kaneta, Yoshizawa, and other, "Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching," in *FPT'10*.

[17] Q. Tang, L. Jiang, X.-x. Liu, and Q. Dai, "A real-time updatable FPGA-based architecture for fast regular expression matching," *Procedia Computer Science*, vol. 31, 2014.

[18] T. Ueda, M. Ito, and M. Ohara, "A dynamically reconfigurable equi-joiner on FPGA," *IBM Tehnical Report RT0969*, 2015.

[19] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *FPGA'14*.

[20] R. J. Halstead, Sukhwani *et al.*, "Accelerating join operation for relational databases with FPGAs," in *FCCM'13*.

[21] P. Francisco *et al.*, "The Netezza data appliance architecture: A platform for high performance data warehousing and analytics." *IBM Redbooks*, 2011.

[22] C. Dennl, D. Ziener, and J. Teich, "On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library," in *FCCM'12*.

[23] L. Woods, Z. István, and G. Alonso, "Ibex - an intelligent storage engine with support for advanced SQL off-loading," *PVLDB*, vol. 7, no. 11, Jul. 2014.