HARDWARE-ACCELERATED REGULAR EXPRESSION MATCHING FOR HIGH-THROUGHPUT TEXT ANALYTICS

Kubilay Atasu, Raphael Polig, Christoph Hagleitner

IBM Research - Zurich
email: {kat,pol,hle}@zurich.ibm.com

Frederick R. Reiss

IBM Research - Almaden
email: frreiss@us.ibm.com

ABSTRACT

Advanced text analytics systems combine regular expression (regex) matching, dictionary processing, and relational algebra for efficient information extraction from text documents. Such systems require support for advanced regex matching features, such as start offset reporting and capturing groups. However, existing regex matching architectures based on reconfigurable nondeterministic state machines and programmable deterministic state machines are not designed to support such features. We describe a novel architecture that supports such advanced features using a network of state machines. We also present a compiler that maps the regexs onto such networks that can be efficiently realized on reconfigurable logic. For each regex, our compiler produces a state machine description, statically computes the number of state machines needed, and produces an optimized interconnection network. Experiments on an Altera Stratix IV FPGA, using regexs from a real life text analytics benchmark, show that a throughput rate of 16 Gb/s can be reached.

1. INTRODUCTION

Regular expression (regex) matching is one of the most computationally intensive tasks in network intrusion detection systems, such as Snort [1], and in information extraction systems, such as IBM's SystemT text analytics software [2]. Regex matching and dictionary matching operations can consume up to 90 percent of the execution time in SystemT.

Fig. 1 illustrates a typical SystemT query, where the results of a regex operation and the results of a dictionary operation are joined based on a predicate. For instance, we might be interested in identifying the regions of text where a regex match follows a dictionary match and the regex match starts, at most, 30 characters after the dictionary match ends. Such a query would be executed on a stream of text documents. For each document, the query execution could produce a set of regions that are marked by the start offsets of the dictionary matches and the end offsets of the regex matches.

A regex can be transformed into a nondeterministic finite state automaton (NFA) or into a deterministic finite state automaton (DFA) using well-known techniques [3]. Efficient accelerator architectures for programmable DFA [4, 5, 6, 7] and reconfigurable NFA [8, 9, 10, 11], which can process the input stream at a deterministic rate, are readily available. Approaches that combine NFAs and DFAs to explore the trade-offs between computation and memory consumption have also been explored [12, 13, 14]. However, all of these approaches are severely limited in supporting requirements, such as start-offset reporting. On the other hand, support for start-offset reporting and other advanced regex features, such as capturing groups (see for instance, Perl Compatible Regular Expressions library[1]), is a must in text analytics systems, such as SystemT, which heavily rely on such features.

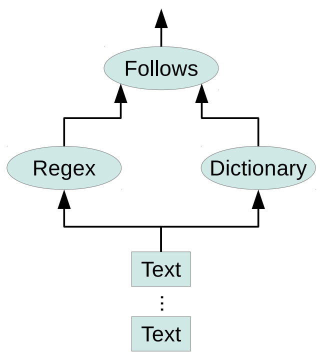

Fig. 1. A text analytics query to find the regions of text where a regex match follows a dictionary match, and there are a given number of characters between the end offset of the dictionary match and the start offset of the regex match.

In this work we describe optimized regex architectures for supporting start offset reporting and capturing groups. We demonstrate the scalability of our approach on Field Programmable Gate Arrays (FPGAs). Our contributions are:

1. An architecture that utilizes multiple state machines and an optimized network for NFA simulation, where pack and unpack operations are used to perform multiple nondeterministic state transitions in parallel;
2. Computation of the size of the network for each regex;

[1]www.pcre.org

978-1-4799-0004-6/13/$31.00 ©2013 IEEE

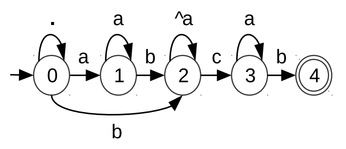

Fig. 2. NFA for the regex `.*a*b[^a]*ca*b`.

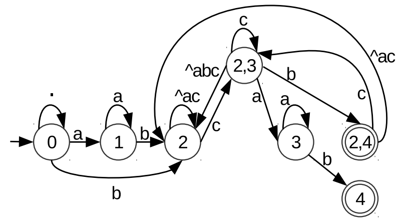

Fig. 3. NFA with a single nondeterministic state.

3. Support for start offset reporting, including possible simplifications in the interconnection network;
4. Support for capturing groups, including transformations on the NFAs to guarantee a maximum number of two next states for each input/state combination.

2. BACKGROUND

State-of-the-art regex architectures based on programmable DFAs and hardwired NFAs simply raise a flag to signal a regex match, which only reveals the end offset of the match in the input stream. The start offsets can be calculated based on the end offsets if the regex has a fixed length, but this is rarely the case. For instance, in the regex `a*b[^a]*ca*b`, any number of repetitions of `a` in `a*b` and in `ca*b`, and any number of repetitions of `^a` between these two are allowed, making the difference between start and end offsets variable.

A naive approach for start offset reporting involves recording the start offset each time the first character (or a fixed length prefix) of a regex is encountered in the input stream, e.g., the character `a` in the example regex above. The problem with such an approach is that `a` occurs multiple times in the regex `a*b[^a]*ca*b`. Assume, for instance that we are given the input string `abcabcab`. A regex matcher should normally report four matches in this string. Assuming that the first character of the input string is at offset 0, the first regex match starts at offset 0 and ends at offset 4, the second one starts at offset 1 and ends at offset 4, the third match starts at offset 3 and ends at offset 7, and the fourth regex match starts at offset 4 and ends at offset 7. Furthermore, if the leftmost match semantics are used, the second and the fourth matches should not be reported as they are contained by the first and the third matches, respectively. Note also that the first and the third matches overlap with each other. This simple example demonstrates that multiple start offset values must be remembered at any time by a

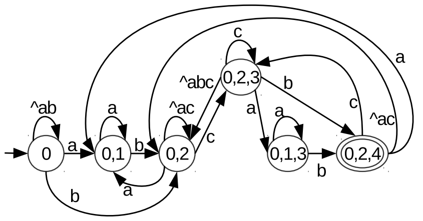

Fig. 4. DFA for the regex `.*a*b[^a]*ca*b`.

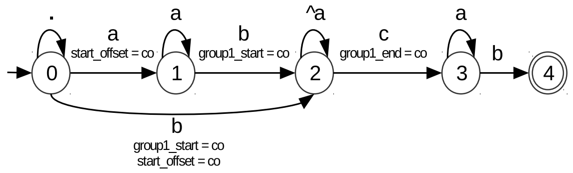

Fig. 5. NFA for the regex `.*a*(?b[^a]*c)a*b`. The start and end offsets of the captured group `b[^a]*c` are updated based on the current offset (co) in the input stream.

regex matcher and these values must eventually be associated with different end offsets. The existing regex accelerator architectures based on programmable DFAs and reconfigurable NFAs do not support such a functionality.

The NFA of the regex `.*a*b[^a]*ca*b` is depicted in Figure 2. To support non-anchored matching semantics, which enable the regex matching to start at any offset location, a nondeterministic start state (i.e., state 0) is included in the state diagram. In this example, the state 2 is also nondeterministic. In general, there can be multiple nondeterministic states in the NFA of a regex. The nondeterministic states can be eliminated by applying the powerset algorithm [3], which can result in an exponential increase in the size of the state diagram. Figure 3 shows the state transition diagram after removing the nondeterminism at state 2 using a partial application of the powerset algorithm, where the only remaining nondeterministic state is state 0. The powerset algorithm creates new states and state transitions, which correspond to a set of states of the original NFA. As an example, if the two NFAs were simulated on two non-deterministic machines, while the NFA of Fig. 3 is in state $\{2,3\}$ the NFA of Fig. 2 would be in state 2 and in state 3. Finally, Figure 4 shows the DFA of the regex `a*b[^a]*ca*b`. Again, some of the DFA states correspond to multiple different states of the original NFA. For instance, while the DFA is at state $\{0,2,4\}$, the original NFA would be in state 0, state 2, and state 4 if the NFA was simulated on a nondeterministic machine. Note that state 0 of the original NFA is always active.

Fig. 5 illustrates the use of a capturing group, where the subexpression `b[^a]*c` is captured using parantheses. Here, the delimiter `(?` marks the start of a capturing group, and the matching closing paranthesis marks the end of the capturing group. When one or more capturing groups are defined in a regex, for each regex match, the start and end offsets of the capturing groups are reported in addition to the start and end offsets of the complete regex. Fig. 5 illustrates a way of implementing this behavior assuming that the state machine receives two inputs: 1) an input character called cur_input, 2) the offset of the current input character in the input stream, which we call cur_offset (co). The transitions from state 0 to states 1 and 2 copy the cur_offset, into a start offset register. The transitions from states 0 and 1 to state 2 copy the cur_offset into a capturing group start offset register, and the transition from state 2 to state 3 copy the cur_offset into a capturing group end offset register.

3. PROPOSED ARCHITECTURES

In this section, we describe architectures for supporting start offset reporting and capturing groups in regex matching. Our architectures are based on parallel state machines that communicate through an optimized interconnection network.

3.1. State Transition Logic

We assume that multiple replicas of the same state machine are implemented in hardware. Figure 6 illustrates the way nondeterministic state transitions are supported within each replica. Each replica can be in an active mode or in an inactive mode, and this information is stored in a register ($active\_reg$). The inactive replicas are always at state 0 and no state transitions can occur in those state machines. For each active replica, a configuration register ($config\_reg$) stores the current state ($state\_reg$), a match signal indicating whether the current state results in a match ($match\_reg$), the start offset of the matching regex in the input stream ($start\_offset\_reg$), and start and end offsets for zero or more capturing groups. Figure 7 illustrates the organization of a configuration register, supporting K capturing groups.

Assume that an input character is consumed and a state transition occurs in every clock cycle. The next configuration computation uses the current input (cur_input), the current offset (cur_offset) and the current state ($state\_reg$) stored in the $config\_reg$ to produce an $active\_config$ output (see Fig. 6), which involves a state, a match signal, a start offset, and zero or more capturing group start and end offsets. The new values get stored in the $config\_reg$ of the replica in the next cycle. An active replica can become inactive if no valid state transitions are found at the current state for the current input character. For instance, at state 1 of Fig. 3, there are no state transitions defined for the input character c. In such cases, the $active\_flag$ output of the replica is set to 0, which makes it inactive in the next cycle.

Assume that the number of state machine replicas is N and the set of replicas are indexed by $\{0..N\text{-}1\}$. The hardware can be initialized by setting ($active\_reg = 1$) for replica 0, and by setting all other registers of replica 0 to 0, and by

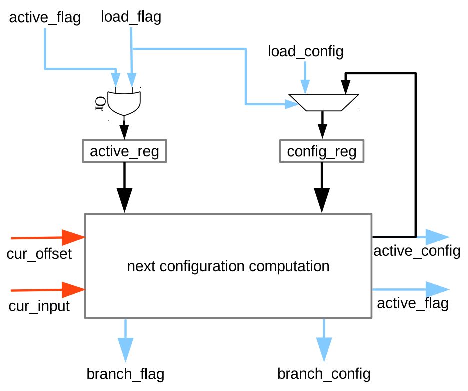

Fig. 6. Support for nondeterministic state transitions.

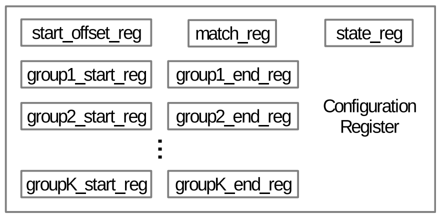

Fig. 7. Configuration Register.

setting all the registers to 0 for the remaining replicas. Assume also that the hardware is configured to implement the NFA of Fig. 3 . Whenever a character a is encountered in the input stream, a nondeterministic state transition is triggered: replica 0 remains in state 0, and another replica gets activated, which continues from state 1, and records the address of the latest character consumed in the input stream in its $start\_offset\_reg$. Similarly, whenever a character b is encountered in the input stream, a nondeterministic state transition is triggered: replica 0 remains in state 0, and another replica gets activated, which continues from state 2. Whenever a non-deterministic state transition is triggered by a state machine, the respective $branch\_flag$ output shown in Fig. 6 is raised, and a $branch\_config$ output is generated, which involves a state, a match signal, a start offset, and zero or more capturing group start and end offsets. The $branch\_flag$ and the $branch\_config$ outputs are then forwarded to the $load\_flag$ and the $load\_config$ inputs of a selected state machine, whose $active\_reg$ and $config\_reg$ registers get updated in the next clock cycle.

If the regex is non-anchored, the replica 0 always remains active, and at state 0. If the regex is anchored, the replica 0 remains active for one cycle only, and performs a single state transition that starts from state 0. Therefore, in case of replica 0, it is sufficient to implement only the state transitions for state 0 and omit the registers shown in Fig. 6.

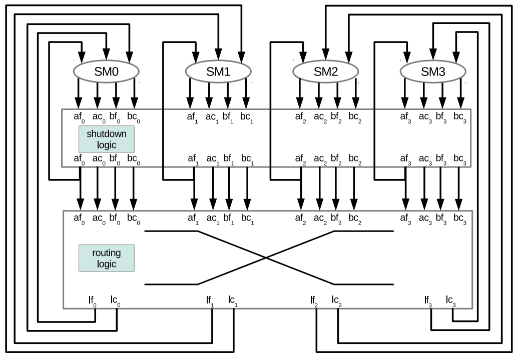

Fig. 8. Network of NFAs.

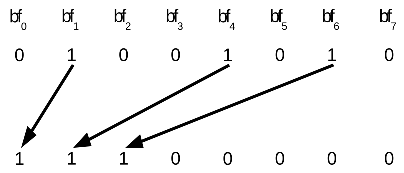

Fig. 9. Packing.

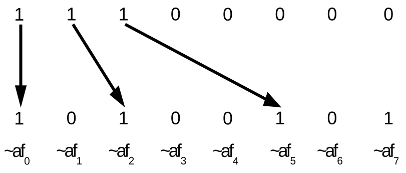

Fig. 10. Unpacking.

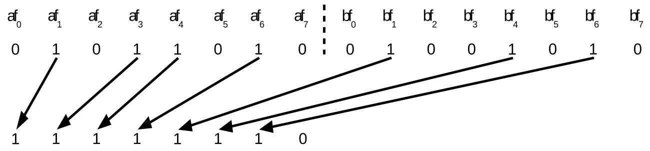
af0 af1 af2 af3 af4 af5 af6 af7 bf0 bf1 bf2 bf3 bf4 bf5 bf6 bf7
0 1 0 1 1 0 1 0 0 1 0 0 1 0 1 0
1 1 1 1 1 1 1 0

Fig. 11. A wide pack replaces the pack-unpack sequence.

3.2. Interconnection Network

Our architecture utilizes multiple replicas of a state machine that operate in parallel. An interconnection network that enables these replicas to communicate with each other is essential (see Figure 8). There is a single active state machine initially, and while processing the input stream, additional replicas get activated. The replicas that get activated are provided with initialization data stored in the $load\_config$ signals shown in Figure 6. While processing the input stream, some of the activated replicas can become de-activated. As a result, any combination of active and inactive replicas can be produced depending on the input stream. Note that whenever a $branch\_flag$ is asserted, there can be multiple inactive replicas, one of which must be selected and activated.

In general, there can be multiple non-deterministic states in an NFA, and multiple $branch\_flag$ signals can be activated in parallel by multiple active replicas. Handling all of these branches in parallel requires selecting multiple inactive replicas, and routing multiple $branch\_config$ values to the selected replicas in parallel. Implementing such a functionality without introducing multiple delay cycles into the state transition loop is a nontrivial task, and requires an efficient interconnection network. An architecture that combines multiple state machines with an interconnection network that enables routing of the $branch\_flag$ (bf) and $branch\_config$ (bc) values to the respective $load\_flag$ (lf) and $load\_config$ (lc) values is shown in Fig. 8. Note that the routing decisions are made based solely on the current values of $branch\_flag$ (bf) and $active\_flag$ (af) signals.

First, the $branch\_flag$ (bf) and $branch\_config$ (bc) pairs are packed using the $branch\_flag$ signals, as illustrated in Fig. 9. The parallel pack operation can be efficiently implemented using a reverse butterfly network, and additional control circuitry that uses parallel prefix computation [15]. After the parallel pack operation, a parallel unpack operation is used to forward the $branch\_flag$ (bf) and $branch\_config$ (bc) pairs to the $load\_flag$ (lf) and $load\_config$ (lc) inputs of the selected replicas. The unpack operation is performed based on the negated $active\_flag$ (af) signals as illustrated in Fig. 10. The parallel unpack operation can be efficiently implemented using a butterfly network, and additonal control logic that uses parallel prefix computation [15]. The critical path of the butterfly and reverse butterfly networks, and the parallel prefix computation grow only logarithmically with the number of replicas. This enables scalable hardware implementations for pack and unpack operations. Alternative implementations can be found, for instance, in [16].

It is possible to reduce the delay of the interconnection network at a limited area overhead. Note that the delay of a butterfly or a reverse butterfly network is $log(N)$, and the respective area cost is $(N/2)log(N)$, where N is the number of nodes (i.e., replicas) interconnected by the network. The overall delay of the interconnection network is $2log(N)$, and the overall area cost is $Nlog(N)$. Figure 11 illustrates that the pack and unpack sequence can be replaced by a wide pack operation that combines all $2N$ values associated with $active\_flag$ (af) and $branch\_flag$ (bf) signals. The delay of such a pack operation is $log(2N) = log(N) + 1$, and the overall area cost is $Nlog(2N) = Nlog(N) + N$.

Our architecture includes a shutdown logic to avoid redundant next state computations and to support leftmost match semantics (see Figure 8). If one or more transitions produce the same state in their $active\_config$ or $branch\_config$ data, the redundancy is eliminated by setting the $active\_flag$ or $branch\_flag$ signals to 0 in all of these configurations except one. If leftmost match semantics are used, only the

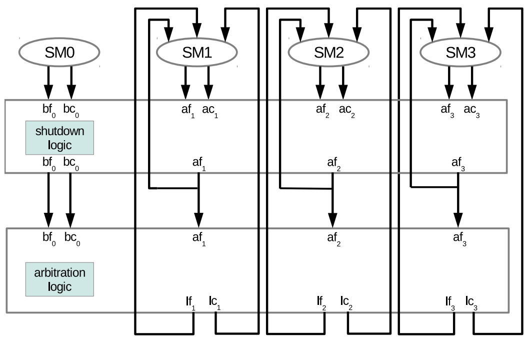

Fig. 12. Simplified Network of NFAs.

configuration with the smallest start offset remains active. Note that such an operation is executed immediately after the next state computation and before any routing starts.

3.3. Computing The Size of The Network

The maximum number of NFA states that can become active concurrently can be computed statically by applying a power-set algorithm and computing the mapping of the NFA states to the DFA states. As an example, the maximum number of NFA states that become active concurrently is three in Fig. 4, where the DFA states with the maximum number of NFA states that are mapped are {0,2,3} and {0,2,4}.

3.4. Start Offset Reporting Using Arbitration

In this section, we are going to show that if the only required regex feature is start offset reporting, the interconnection network architecture described in Section 3.2 can be significantly simplified. First of all, we apply the powerset algorithm partially on the NFA of the regex to eliminate the nondeterminism from all the states except state 0, as illustrated in Fig. 3. Note that in such a case, the only replica that can assert the $branch\_flag$ is the replica that implements the state 0. Hence, $branch\_flag$ and $branch\_config$ signals can be eliminated from the remaining replicas for resource optimization. Whenever the $branch\_flag$ signal is asserted by the replica 0, if there are multiple inactive replicas, an arbitration logic must be used to select the replica that will be activated. In our architecture, the $load\_flag$ signal gets asserted in the selected replica, and the $branch\_config$ signal is routed to the $load\_config$ signal of the selected replica. A straightforward implementation of this solution can simply select the replica with the lowest index among all the inactive replicas, which can be efficiently implemented using a simple arbiter. In this way, the complex interconnection network that utilizes pack/unpack operations is eliminated. The shutdown logic must still be utilized as a preprocessing step prior to the arbitration logic. Fig. 12 depicts the interconnection structure of the resulting simplified architecture.

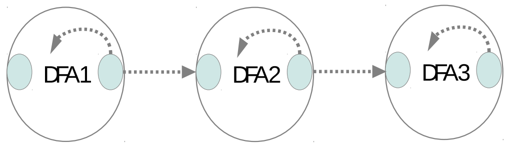

Fig. 13. Given the regex `R1(?R2)R3`, where R1, R2, and R3 are subexpressions, and subexpression R2 is captured, we convert R1, R2, and R3 into DFA 1, DFA 2, and DFA 3, respectively, and add state transitions between the interface states that are shown in small circles. The interface states can be nondeterministic due to the possible back edges, resulting in up to two possible next states per input character.

3.5. Start Offset Reporting Using A Simplifed Pack

A simplified pack network that packs $N-1$ $active\_config$, $active\_flag$ pairs and a single $branch\_config$, $branch\_flag$ pair into $N-1$ $load\_config$, $load\_flag$ pairs can be utilized for start offset reporting, which reduces the area and the latency of the pack network with respect to the network described in Fig. 11. Of course, using such a simplified pack network is still more expensive than using an arbiter. However, using the pack network guarantees that a replica with an index i always has a smaller start offset than a replica with an index j when $i < j$. This property greatly simplifies the implementation of leftmost-match semantics in the shutdown logic. We are omitting the details for brevity.

3.6. Capturing Group Support

The architectures described in Section 3.4 and in Section 3.5 are not general enough to support the capturing groups mainly because nondeterminism cannot be completely removed from all NFA states without loosing the capturing group information. In other words, it is not possible to produce an NFA with a single nondeterministic state (i.e., state 0) in the general case. Therefore, we have to use the more complex interconnection scheme, as described in Section 3.2. However, mapping regexs with capturing groups to such an architecture is still a non-trivial task because our architecture allows a limited amount of nondeterminism per state. For instance, as depicted in Figure 6, for each state/input combination, our state machine elements can produce at most two possible next states that are stored in $active\_config$ and $branch\_config$ outputs. Increasing the width of the $branch\_config$ output to embed multiple next state configurations is possible, but requires a wider interconnection network that is more costly to implement. We address this problem by transforming the NFA as illustrated in Figure 13, where we split the NFA into subexpressions that start or end with the delimiters associated with the capturing groups and we convert the resulting subexpressions individually into deterministic state machines. Such a transformation guaran-

Table 1. Percent resource consumption for three regex sets from text analytics and network intrusion detection domains.

Benchmark	# regexs	Architecture 1		Architecture 2		Architecture 3		Capturing Groups	
		LUTs	Registers	LUTs	Registers	LUTs	Registers	LUTs	Registers
Text Analytics	25	4.4	4.3	6.6	5.0	10	6.4	9.6	8.4
L7 Filter	101	29	20	40	22	59	30	-	-

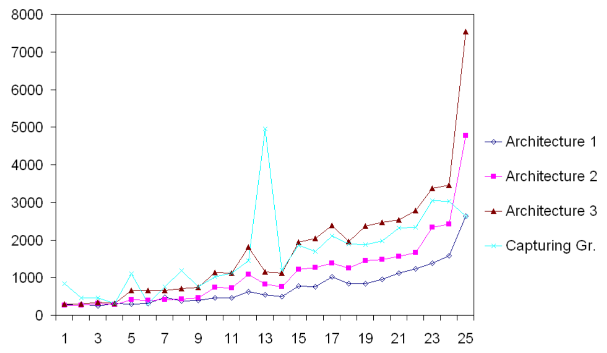

Fig. 14. Comb. LUT usage for 25 Text Analytics regexs.

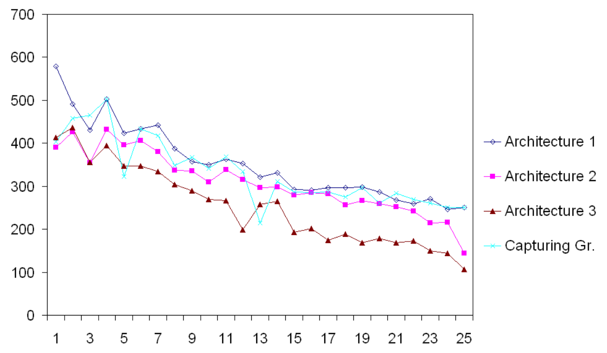

Fig. 15. Clock frequency (MHz) for Text Analytics regexs.

tees at most two possible next states per input character at the interface states, while all other states are deterministic.

4. IMPLEMENTATION & EXPERIMENTS

To optimize the clock frequency for a wide range of regexs, we have pipelined our regex processing into two stages. In the first stage, the active state machines compute their next states. In the second stage, a pack operation or an arbiter is used to route the branch signals into the load signals. A single regex pipeline is time-shared by two interleaved text streams, accepting an 8-bit character from each stream every two cycles. This results in a single-stream throughput rate of 1 Gb/s and an aggregate throughput rate of 2 Gb/s per pipeline assuming a clock frequency of 250 MHz. The aggregate throughput rate can be further improved by replicating the pipelines until the FPGA resources are saturated. The aggregate throughput rate is what matters most in text analytics systems because such systems can operate over thousands of independent text documents in parallel.

In our implementation, we used hardwired state machines, but our architecture could also utilize programmable state machines [17]. We made use of character classifier tables, similar to those described in [17], to reduce the number of state transition rules, and to reduce complexity of the next state computation. Our compiler generates the Verilog code for each regex depending on the desired regex features.

We performed experiments on one regex set extracted from a real-life text analytics query [2] and on a regex set that is part of the application layer packet classifier for Linux™ (i.e., the L7 filter) [18]. We synthesized the architectures that were automatically generated by our compiler using the Quartus II 12.1 software for an Altera Stratix IV GX530 FPGA. Table 1 shows our synthesis results for three different architectures outlined in Section 3: Architecture 1 implements the arbitration-based interconnection scheme described in Section 3.4; Architecture 2 implements the simplified pack network described in Section 3.5; Architecture 3 implements the wide pack network described in Section 3.2. All three architectures support start offset reporting. In addition, we have inserted capturing groups into the text analytics regexs, and evaluated the respective resource consumption. To support capturing groups, additional configuration registers (see Figure 7) are used. If the state transition graph includes only a single nondeterministic state after the transformations illustrated in Fig. 13, Architecture 1 is instantiated. Otherwise, the more costly Architecture 3 is used.

The combinational logic resources used and the clock frequency achieved by the 25 text analytics regexs, in the increasing order of complexity, are given in Figure 14 and in Figure 15, respectively. Note that the complexity of a regex grows with the number of state machines and with the number of state transition rules implemented by each state machine. Architecture 1 is the most efficient choice in terms of the logic resource usage, and Architecture 3 is the least efficient choice. In case of Architecture 1, all Text Analytics regexs met the 250 MHz timing constraint. While supporting capturing groups, all Text Analytics regexs but one met the 250 MHz timing constraint. Some text analytics regexs reached only 144 MHz using Architecture 2, and some reached only 106 MHz using Architecture 3. Based on the resource consumption numbers given in Table 1, Architecture 1 can achieve a throughput rate of 16 Gb/s using eight hardware pipelines that process 16 document streams in parallel. The throughput rate of the software that provides the same functionality is around 50 Mb/s while executing 16 software threads on an Intel™ Xeon E5530 processor, running at 2.4 GHz. Thus, our FPGA-based solution can achieve a 320 fold speed-up over the equivalent software.

For L7 filter regexs, the minimum clock frequency reported for Architecture 1 is 146 MHz, while the resource usage is below 50%. Therefore, Architecture 1 can achieve a throughput rate of at least 1 Gb/s at 125 MHz using a single hardware pipeline that processes two streams in parallel.

5. RELATED WORK

Ruehle [19] describes a system for hardware processing of regexs. State information associated with one or more states of an NFA is stored in a register bank and a crossbar network is used to interconnect the states. State information, such as transitions and spin counts updated while processing an input stream. Such a network can be very expensive to implement because the number of states in an NFA grows linearly with the number of characters in a given regex.

Srinivasan and Starovoytoy[20] describe a method for determining the length of one or more substrings of an input string that matches a regex. First, the input string is searched for the regex using an NFA and, upon detecting a match, the NFA is inverted, so that it embodies the inverse of the regex. The match string is also reversed and searched for the inverted regex using the inverted NFA. A main disadvantage of such an approach is that the input stream has to be scanned twice for each regex match, which can significantly reduce the throughput rate of the regex matching.

6. SUMMARY AND CONCLUSIONS

Support for advanced regex features, such as start offset reporting and capturing groups has been largely neglected in the design of regex accelerators. However, such features are of utmost importance in emerging text analytics workloads. We propose a novel regex matching architecture that supports such advanced features while operating in a streaming manner without any backpressure. We also present compilation methods that produce optimized regex architectures. Our experiments on an Altera Stratix IV FPGA, running at 250 MHZ, demonstrate a 1 Gb/s single-stream processing rate, and an up to 16 Gb/s aggregate throughput rate for regexs that are part of a real-life text analytics query, resulting in a 320 fold speed-up over the equivalent software.

In our future work, we hope to reduce the resource consumption of our accelerator architecture to make it more scalable. In addition, we plan to extend our architecture to cover additional regex features, such as backreferencing.

7. REFERENCES

[1] SNORT network intrusion detection system. http://www.snort.org/. Accessed: 2013-04-24.

[2] Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. SystemT: a system for declarative information extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[3] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.

[4] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc. ANCS*, pages 93–102. ACM, 2006.

[5] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. SIGCOMM*, pages 339–350, 2006.

[6] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proc. SIGCOMM '08*, pages 207–218. ACM, 2008.

[7] Jan van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *Proc. MICRO*, pages 461–472, Dec. 2012.

[8] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using FPGAs. In *Proc. FCCM '01*, pages 227–238, 2001.

[9] Ioannis Sourdis, Joao Bispo, Joao M. Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *J. Signal Process. Syst.*, 51(1):99–121, 2008.

[10] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *Proc. ANCS*, pages 30–39, 2008.

[11] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proc. ANCS*, pages 127–136. ACM, 2007.

[12] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. CoNEXT*, 2007.

[13] Yi-Hua E. Yang and Viktor K. Prasanna. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In *Proc. INFOCOM*, pages 1853–1861, 2011.

[14] Hiroki Nakahara, Tsutomu Sasao, and Munehiro Matsuura. A regular expression matching circuit based on a decomposed automaton. In *Proc. ARC*, pages 16–28, 2011.

[15] Yedidya Hilewitz and Ruby B. Lee. Fast bit compression and expansion with parallel extract and parallel deposit instructions. In *Proc. ASAP*, pages 65–72, 2006.

[16] G. Dimitrakopoulos, Christos Mavrokefalidis, K. Galanopoulos, and D. Nikolos. Sorter based permutation units for media-enhanced microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(6):711–715, 2007.

[17] Jan van Lunteren and Alexis Guanella. Hardware-accelerated regular expression matching at multiple tens of Gb/s. In *Proc. INFOCOM*, pages 1737–1745, 2012.

[18] Application layer packet classifier for Linux. http://l7-filter.sourceforge.net/. Accessed: 2008-11-23.

[19] Michael D. Ruehle. Detection of patterns in a data stream. US Patent No.: US 8,190,738 B2, May 2012.

[20] Maheshwaran Srinivasan and Alexey Stravoytoy. Determining regular expression match lengths. US Patent No.: US 8,051,085 B1, Nov 2011.