TOKEN-BASED DICTIONARY PATTERN MATCHING FOR TEXT ANALYTICS

Raphael Polig, Kubilay Atasu, Christoph Hagleitner

IBM Research - Zurich
Rueschlikon, Switzerland
email: pol, kat, hle@zurich.ibm.com

ABSTRACT

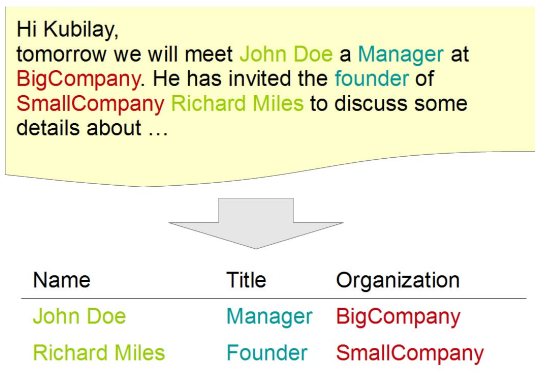
When performing queries for text analytics on unstructured text data, a large amount of the processing time is spent on regular expressions and dictionary matching. In this paper we present a compilable architecture for token-bound pattern matching with support for token pattern sequence detection. The architecture presented is capable of detecting several hundreds of dictionaries, each containing thousands of elements at high throughput. A programmable state machine is used as pattern detection engine to achieve deterministic performance while maintaining low storage requirements. For the detection of token sequences, a dedicated circuitry is compiled based on a non-deterministic automaton. A cascaded result lookup ensures efficient storage while allowing multi-token elements to be detected and multiple dictionary hits to be reported. We implemented on an Altera Stratix IV GX530, and were able to process up to 16 documents in parallel at a peak throughput rate of 9.7 Gb/s.

1. INTRODUCTION

Extracting structured information from large amounts of unstructured data is becoming an important aspect in today's enterprise. This task is referred to as text analytics (TA) and can be accomplished by software such as SystemT [1]. In SystemT, a user specifies queries to extract the desired information from unstructured text data. Formulating these queries can become a difficult task in itself and their complexity challenges today's computational systems to process sufficient data in adequate time.

A common operator in such queries is the detection of particular sets of words and patterns, which is referred to as dictionary matching (DM). Before DM is performed, a document is split into parts called tokens. This tokenization can be done by splitting the text at whitespaces and/or other special characters or more complex patterns.

Looking at the runtime of TA queries reveals that often more than half of the processing time is spent on DM operators. This is due to the fact that these operators scan the full document, whereas subsequent steps work on the results and parts of the document only. With the ever growing amount


Hi Kubilay,
tomorrow we will meet John Doe a Manager at BigCompany. He has invited the founder of SmallCompany Richard Miles to discuss some details about …

Name	Title	Organization
John Doe	Manager	BigCompany
Richard Miles	Founder	SmallCompany

Fig. 1. Example applying token based dictionary matching.

of unstructured text data, such as emails, web entries and machine data logs, performing the task of DM in a time- and energy-efficient way is becoming increasingly important.

String and pattern matching algorithms and their hardware implementations have been thoroughly studied for over 30 years [2]. Especially as part of network intrusion detection systems (NIDS) high-performance and large-scale pattern detection engines have been developed to keep track with the growing network data rates [3]. Besides matching multiple ten thousand patterns, TA queries require the detection of pattern sequences that cannot be represented as regular expressions. For this we present a novel architecture consisting of a deterministic finite-state automaton (DFA) to detect single token-based patterns and a non-deterministic finite-state automaton (NFA) to find token sequences or multi-token patterns. The DFA is realized as a scalable Balanced Routing Table finite-state machine (BFSM) [4] with additional controls to operate within single token boundaries defined by an external data stream. The NFA is compiled directly into FPGA logic emulating the Baeza-Yates [5] NFA by using shift and AND operations [6].

The main contributions of this paper are:

- A compilable architecture consisting of a programmable DFA and compiled shift registers to perform pattern sequence search for text analytics.

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

- The capability to report the start and end offsets of a pattern found for single and multi-token elements.
- An efficient result lookup structure to report all dictionaries matching a particular pattern and a compiler minimizing the result memory usage.

The paper is organized as follows. Section 2 gives an overview on pattern matching in text analytics. We present our system architecture and the compiler in section 3. Section 4 explains implementation details. The performance and resource evaluation are discussed in section 5. We conclude in section 6 and also discuss future work.

2. DICTIONARY MATCHING AND TEXT ANALYTICS

Exact pattern matching is an important function used in many applications, such as text search, DNA analysis [7] and anti-virus software. Given a set of patterns as a dictionary a pattern matching algorithm finds and returns all occurrences of any given pattern in the data evaluated. The data itself can be plain text, network traffic or binary files.

Text analytics adds specific details to pattern matching. One addition is the tokenization mentioned in section 1. The text data gets split into parts, so-called tokens, by a pre-processing step. The algorithm responsible for this tokenziation is unknown to the pattern matching step. The output of the tokenizer is a set of tuples, each consisting of a start and end offset of a token within the text data. Matches found by the pattern matching algorithm are only valid if the start and end offsets of the found pattern match the offsets for the given token. This is similar to the regular expression $\backslash bJohn \backslash b$ where *John* is only valid at word boundaries. However, for text analytics it is not sufficient to have sets of word and non-word characters to define token boundaries as the token definition can be more complex and is defined externally.

Patterns consisting of a sequence of tokens are referred to as multi-token patterns. This is necessary for example to extract the name *John Doe* from a text as seen in Fig. 1. Assuming *John* and *Doe* are defined as tokens it does not matter by what characters they are separated as long as these two tokens appear in sequence. This cannot be represented by a regular expression as it require knowledge about the tokenization process.

In contrast to string matching text analytics allows the use of simple regular expressions in dictionaries. For example, a dictionary may contain various date formats. In such a case, the pattern is not restricted to a specific sequence of characters but to a sequence of character classes.

Finally text analytics also requires exact operation. This excludes the use of bloom-filters or other approaches causing false positive results.

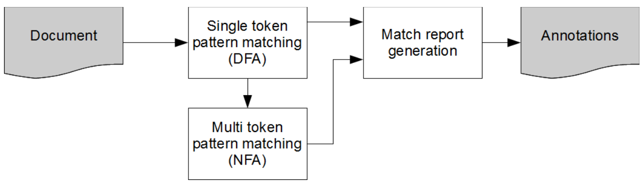

Document
Single token pattern matching (DFA)
Multi token pattern matching (NFA)
Match report generation
Annotations

Fig. 2. High-level view of the architecture.

3. ARCHITECTURE

The main idea of our architecture is the use of a decomposed automaton to detect single token patterns and multi-token sequences (see Fig. 2). For single token pattern detection, a DFA-based architecture is used to provide a deterministic scan rate of the document text data. The DFA needs to detect all single token elements of a dictionary as well as the individual tokens of multi-token elements. To detect the sequence of patterns an NFA based circuitry is used which advances its states on a token by token basis. All its detected patterns must be reported as a combination of the start offset, the end offset and the dictionary identifier value. All dictionaries must be reported containing the pattern.

The overall hardware architecture of the dictionary matcher presented consists of three main sections as indicated in Fig. 3. Two individual DMA engines are responsible for retrieving the document text data and the token offset values from main memory. The text data is stored as 8-bit ASCII characters, whereas the token offsets are defined as a tuple of two 32-bit integer values representing the start and end offsets of a token. A counter is used as character pointer to keep track of the current position in the document text. The boundary detector is responsible for determining the start and end characters of a token by comparing the character pointer with the incoming offset stream. It controls the operation of the BFSMs by issuing start-of-token and end-of-token pulses. The operation of the BFSMs is discussed in more detail in section 3.1.

The results of the BFSMs are passed on to a decoder which will generate the appropriate transition signals for the multi-token chains implementing the NFA discussed in section 3.2. As a single token may trigger multiple matches on the multi-token NFA, a priority scheduler is necessary to put the results into sequence. This scheduler needs to process all results before the next token can be dispatched by the BFSMs to keep the results in order and avoid information loss. For this purpose it can generate a backpressure halting its input logic.

The actual results are produced by a cascaded result lookup described in section 3.3. The Result Generator is responsible for assembling the correct offsets with the appropriate dictionary id before writing the results to main memory. Like the scheduler, the generator needs to process all results first before a new instruction can be consumed.

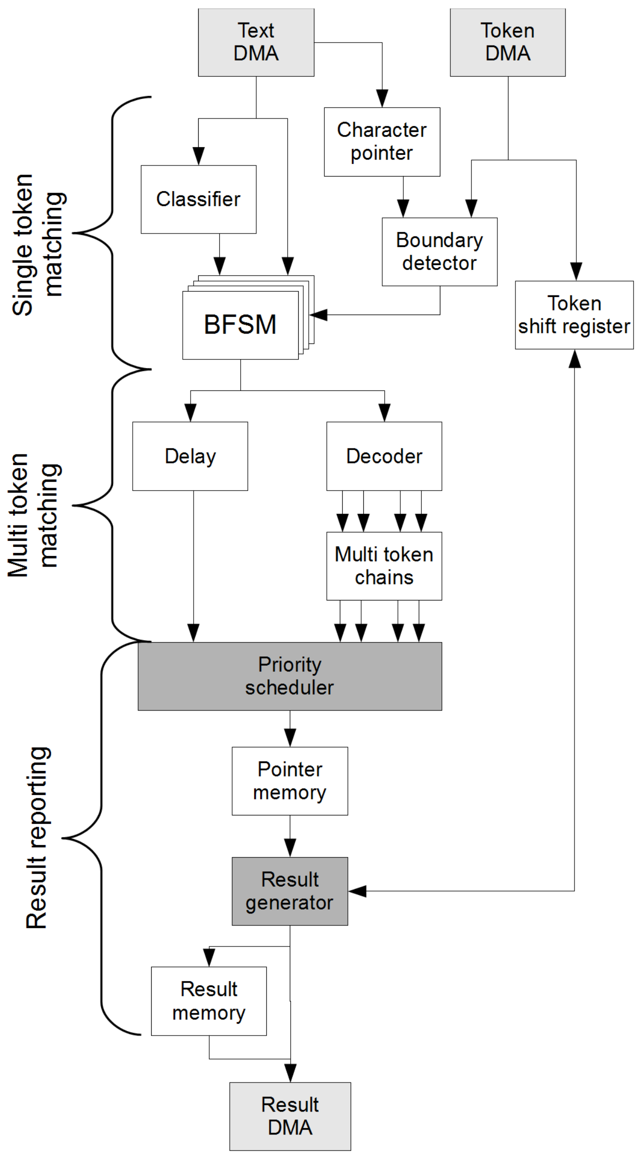


Fig. 3. Detailed view of the architecture. Dark grey blocks can cause backpressure when multiple elements or dictionaries need to be reported.

3.1. BFSM

A BFSM [4] is used as DFA-based pattern detection engine. A BFSM is a programmable finite-state machine (FSM) technology for regular expression matching with deterministic performance. It uses an AC-DFA to detect the patterns compiled for it and stores its transitions as a set of default and transition rules in local memory. The default rules define the extra links that are necessary in an AC-DFA to return from failing patterns and only depend on the incoming character. Transition rules depend on both the current state and the incoming character and define the positive transitions in the DFA. Using this concept the BFSM achieves a high storage efficiency, maintaining deterministic performance consuming one character per cycle.

The size of the rule memory is fixed, allowing the implementation to have a stable achievable frequency regardless of the dictionary size. If a dictionary is too large to fit into a single BFSM instance, additional instances can be used. The compiler will distribute the patterns accross all instances, minimizing the necessary rule memory. By default four instances are used.

The BFSM is kept in its default state until the start of a token is detected by the Boundary Detector, which compares the character pointer value to the start offset field for the current token. If these values match a start of token signal is activated, allowing the BFSM to leave its default state using the current character. The subsequent transitions are carried out without any external interaction. When the end of a token is detected by the Boundary Detector, an end of token signal is activated. If the BFSM reaches a match state with a character marked as end of token then and only then is the match valid and sent to the output. When the end of a token is signaled the BFSM will be reset to its default state regardless of the calculated next state.

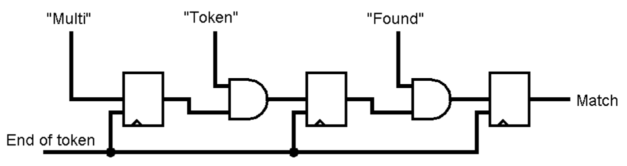


Fig. 4. Multi-token chain

3.2. Multi-token NFA

For detecting multi-token elements, an NFA circuit using single token transitions is used. It is compiled as custom hardware logic as described by Sidhu and Prasanna [6]. Each multi-token element has its own small machine consisting of registers connected by AND gates, as shown in Fig. 4. The number of registers in such a chain is equal to the number of tokens in the multi-token element. The data is shifted at each end of a token when a potential single token has been detected. A single token match id is decoded and fed to the individual register chains, where it is combined with the previous matches. When a set of single token matches appears in the correct sequence, a multi-token match is signalled.

An advantage of this approach is the ability to share resources among multi-token elements using the same prefix. For example, the multi-token elements *John Doe* and *John Wayne* will share the first register for *John*.

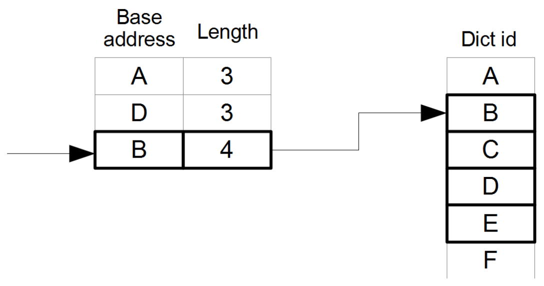


Fig. 5. Cascaded result lookup: The pointer memory selects a block of results in the results memory.

3.3. Result reporting

After the successful detection of a dictionary element the results need to be produced. A single result is composed of a 32-bit dictionary identifier and the start and end offset position of the element found in the current document. As a single element may be contained in multiple dictionaries, multiple results need to be produced. This implies that the amount of data generated by the dictionary matcher may exceed the size of the actual document processed. A cascaded result lookup is used to efficiently store the information in which dictionaries an element is contained in.

A match in the pattern detection engine results in an address derived from its matching state to the pointer memory. This memory contains a single entry for each match from the BFSM. Each entry consists of an address to the result memory and a result length. The result memory contains the actual dicionary IDs, which are grouped by a particular element appearing in such a group. For instance if an element appears in dicionaries A, B and C then these form a continous group in the result memory. A second element appearing in dictionaries D, E and F forms a separate continous group. But for a further element appearing in B, C, D and E, the compiler will create a group overlapping the previously created ones, thus minimizing the necessary storage.

The result generator recieves the base address and the result length and generates the addresses to the result memory. It also assembles the correct token offsets from the token shift register and writes the result back to main memory.

3.4. Compiler

The compiler for the architecture presented takes the dictionaries as plain text files as input. Each file is considered as one dictionary, and each line in a file as a dictionary element. Multi-token sequences are defined in a dictionary file by separating the tokens by a whitespace character.

The compiler first consolidates all single tokens that have to be detected. It then determines the number of necessary BFSM instances to implement the AC-DFA and calls the BFSM compiler [8]. After the BFSM compiler has finished, the multi-token chains are generated as Verilog files. The compiler combines the BFSM match states to form the corresponding token sequences as a multi-token chains. It then generates a decoder transforming the match state to a single bit signal activating the token transition. Next, the result lookup memory is generated. The compiler tries to optimize the use of the result memory by overlapping as many dictionary combinations as possible. Finally the compiler assembles the pointer memory and inserts the pointers into the BFSM rule lines.

4. IMPLEMENTATION

Our reference system is implemented in Verilog on an Altera Stratix-IV GX530 FPGA with a target frequency of 250 MHz. The primary goal is to maximize the aggregated throughput of document data. This means that latency can be neglected if a continuous stream of document data can be maintained at high frequency. To achieve the desired frequency of 250 MHz, the BFSM was modified to run in a stream-interleaved fashion.

The use of dual-port BlockRAM allows the reuse of the same transition rule memory for a second BFSM instance. In this way only the logic resources need to be duplicated to process twice as many streams in parallel while memory usage stays the same. Depending on the dictionary size, multiple such dual BFSMs are instantiated to further increase the number of parallel streams.

The multi-token NFAs need to be instantiated per stream and run pipelined at full frequency. The number of pipeline stages is determined by the single token decoder and the multi-token result multiplexer. The decoder grows with the number of single token patterns used in the multi-token patterns, and the size of the multiplexer is determined by the number of multi-token patterns.

5. EVALUATION

To evaluate our system, we compiled various dictionaries [9] with different sizes and properties to hardware. We used the Altera Quartus 12.1 software to generate the FPGA bitstreams. The resource numbers were generated using a two-threaded dual BFSM capable of processing two by two interleaved streams. For the performance tests, we used a set of test documents with sizes ranging from 100 B to 10 MB.

5.1. Resource utilization

First we examine the impact of the size of the dictionary containing single token elements only. The number of single token elements dominates the size of the transition rule memory. If the number of transition rules gets too large

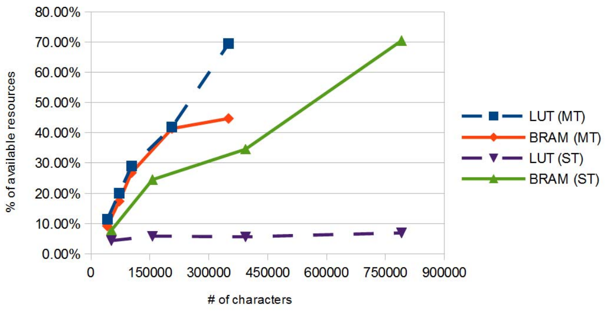


Fig. 6. Resources for a four-stream implementation: Single token (ST) resources are dominated by memory usage whereas detecting multi-token patterns (MT) requires a large number of lookup tables (LUTs).

Table 1. Total resource usage of single token dictionaries

Dictionary	Elements	Size	LUTs	Mbit
Given-Names	8608	67.2kB	18477	1.648
Roget	17474	186kb	24672	5.202
CRL	44880	471kB	23935	7.338
Antworth	89523	947kB	29295	14.962

the compiler will instantiate an additional BFSM instance, increasing lookup table (LUT) consumption. But Fig. 6 shows that the logic resources are negligible compared with the memory requirements of the DFA.

To analyze the impact of multi-token elements on the resource consumption, we compiled artificial dictionaries in such a way that no multi-token element shares a sub-token with another one. This ensures that no register merge occurrs. Figure 6 shows the resulting strong increase of logic resources for token sequences. The memory resources follow a similar pattern as for the single token detection. The additional memory cost comes from the decoder structure to enable the token-chains.

5.2. Throughput

For a single stream, a character is consumed every second cycle. Thus the core throughput rate can be calculated as follows: $T = \frac{f}{2} * n$, where T is the throughput in Bytes per second for ASCII characters, f denotes the frequency and n is the number of streams processed. A single instance of the dual BFSM implementation is able to process four streams, resulting in a theoretical peak throughput rate of 4 Gb/s at 250 MHz.

We have measured the system throughput for 4, 8 and 16 parallel document streams by measuring the time from sending the documents to the FPGA to the arrival of the interrupt

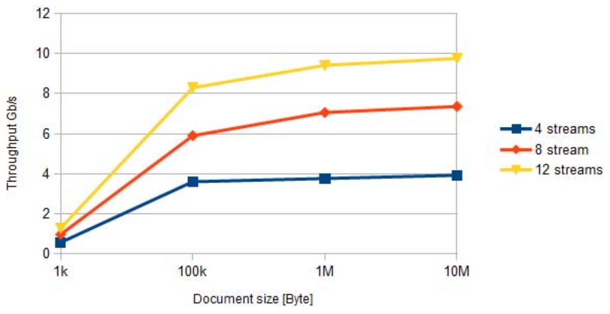


Fig. 7. Throughput for different parallel streams vs. document size. Smaller documents imply a higher DMA setup cost.

signaling the end of processing. This includes all penalties, such as DMA setup and control communication. The results are shown in Fig. 7 in respect of the test documents size. It can be seen that only for documents larger than 100 kB can the full performance be achieved because of the overhead of control communication.

5.3. Comparison with original software

As performance baseline, the original software implementation is used. When running the same dictionaries on an Intel® XEON® [1] E5530 with 2.40 GHz using 16 threads we achieved a maximum throughput of 6.43 MB/s regardless of the document size. Compared with the four stream implementation that is a 14x to 75x improvement, depending on the document size.

5.4. Comparison with related work

We compare the architecture presented with related work using three key figures. The throughput is an indicator of the performance for a given architecture, but depends on the technology available at implementation time. Storage efficiency is characterized using logical cells per character (LC/char) and memory bits per character (bit/char). Table 2 summarizes various architectures.

The architecture presented behaves similar to the MNFAU [10] architecture in terms of storage efficiency because both architectures are based on a decomposed automaton. Keeping the additional resources need to implement the DFA control and result reporting structures in mind, the efficiency decrease is justifiable. The MNFAU architecture can be used to detect regular expressions and could be used to replace the single token pattern matching stage. To enable token-bound

[1]Intel and Intel Xeon are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other product or service names may be trademarks or service marks of IBM or other companies.

Table 2. Comparison with related work

Method	Type	Gb/s	LC/char	bit/char
Sourdis 05 [13]	Dict	12.67	16.86	n.a.
Brodie 06 [14]	Regx	4.0	22.22	3182.2
Bispo 06 [15]	Regx	2.9	1.28	0
Le 10 [16]	Dict	3.5	n.a.	8.4
Nakahara 11 [10]	Regx	1.6	0.25	21.4
Agarwal 13 [11]	Dict	17.8	0.4	21.2
BFSM+NFA	Regx	9.7	0.35	25.5

matching on the MNFAU architecture all shift registers need to be flushed at the end of a token.

In terms of performance the implementation by Agarwal [11] tops the chart. It uses a hash-based approach for dictionary matching. Although it uses special characters to identify token boundaries, it could be easily extended to use the external token definition stream. A limitation is the limited width of a single dictionary element.

Another possibility for string matching is to keep the entire dictionary in a content addressable memory (CAM). An implementation using a TCAM [12] has achieved good results in terms of memory efficiency and deterministic throughput rate. But the number of dictionary patterns is directly limited by the size of the CAM, which limits it to a few thousand patterns.

6. CONCLUSION

A compilable architecture for dictionary matching in text analytics was presented that supports operation on predefined tokens. Furthermore it supports the detection of token sequences that cannot be expressed using regular expressions. Results show throughput rates up to 9.7Gb/s for 12 parallel streams. The implemented architecture can to hold up to 100.000 single token and around 30.000 multi-token elements.

These results encourage us to further improve our approach. Next steps are an efficient enablement for UTF-8 support and possible multi-character consumption. Another interesting requirement by text analytics is the detection of a regular expression across a specified range of tokens. This implies starting a new scan operation at each token start while continuing from the scan state reached by preceeding tokens.

7. REFERENCES

[1] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu, "SystemT: a system for declarative information extraction," *ACM SIGMOD Record*, vol. 37, no. 4, pp. 7–13, 2009.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[3] C. J. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of snort," in *DARPA Information Survivability Conference & Exposition II.*, vol. 1. IEEE, 2001, pp. 367–373.

[4] J. Van Lunteren, "High-performance pattern-matching for intrusion detection," in *IEEE INFOCOM*, vol. 6, 2006, pp. 1409–1421.

[5] R. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Communications of the ACM*, vol. 35, no. 10, pp. 74–82, 1992.

[6] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2001, pp. 227–238.

[7] D. Herath, C. Lakmali, and R. Ragel, "Accelerating string matching for bio-computing applications on multi-core cpus," in *International Conference on Industrial and Information Systems (ICIIS)*. IEEE, 2012, pp. 1–6.

[8] J. Rohrer, K. Atasu, J. van Lunteren, and C. Hagleitner, "Memory-efficient distribution of regular expressions for fast deep packet inspection," in *Proc. CODES+ISSS*, 2009, pp. 147–154.

[9] CERIAS, "Purdue university dictionary collection," 2013. [Online]. Available: ftp://ftp.cerias.purdue.edu/pub/dict/wordlists/dictionaries/

[10] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching circuit based on a decomposed automaton," *Reconfigurable Computing: Architectures, Tools and Applications*, vol. 6578, pp. 16–28, 2011.

[11] K. Agarwal and R. Polig, "A high-speed and large-scale dictionary matching engine for information extraction systems," in *Application-specific Systems, Architectures and Processors (ASAP2013) (to be published)*, 2013.

[12] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using tcam," in *International Conference on Network Protocols (ICNP)*. IEEE, 2004, pp. 174–183.

[13] I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10 gbps fpga-based nids," *New Algorithms, Architectures and Applications for Reconfigurable Computing*, pp. 195–207, 2005.

[14] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 191–202.

[15] J. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in *IEEE Conference on Field Programmable Technology (FPT)*. IEEE, 2006, pp. 119–126.

[16] H. Le and V. K. Prasanna, "A memory-efficient and modular approach for string matching on fpgas," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2010, pp. 193–200.