DESIGN OF PATTERN MATCHING SYSTEMS: PATTERN, ALGORITHM, AND SCANNER

A Dissertation

by

HAO WANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jyh-Charn Liu
Committee Members,	Rabi N. Mahapatra
	Guofei Gu
	Jiang Hu
Head of Department,	Nancy Amato

December 2013

Major Subject: Computer Engineering

ABSTRACT

Pattern matching is at the core of many computational problems, e.g., search engine, data mining, network security and information retrieval. In this dissertation, we target at the more complex patterns of regular expression and time series, and proposed a general modular structure, named *character class with constraint repetition* (CCR), as the building block for the pattern matching algorithm. An exact matching algorithm named MIN-MAX is developed to support overlapped matching of CCR based regexps, and an approximate matching algorithm named Elastic Matching Algorithm is designed to support overlapped matching of CCR based time series, i.e., music melody. Both algorithms are parallelized to run on FPGA to achieve high performance, and the FPGA-based scanners are designed as a modular architecture which is parameterizable and can be reconfigured by simple memory writes, achieving a perfect balance between performance and deployment time.

DEDICATION

To My Wife and Parents.

ACKNOWLEDGEMENTS

I would like to express the deepest gratitude to my advisor, Dr. Jyh-Charn (Steve) Liu, for his guidance and financial support throughout my Ph.D. study. He was patient when I struggled with research challenges, and gradually helped me out by hints and inspirations, rather than directly pointing out the solution. From him, I learned one of the most valuable assets in my life, critical thinking. I would also like to thank Dr. Rabi Mahapatra, Dr. Guofei Gu, and Dr. Jiang Hu for their precious time to serve as my committee members, and their valuable comments and suggestions on improving the quality of my work.

Many thanks to the staff and faculty of Texas A&M University, who made it a warm and enjoyable experience to pursue my graduate study. I would also like to thank my colleagues at the Real Time Distributed System laboratory, for their helpful comments and feedbacks on my research topics.

Finally, I would like to thank my parents, wife and elder brother for their ceaseless support, encouragement and love, without which I would never have been able to finish my Ph.D. study.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

1. INTRODUCTION

Pattern matching has been a classical problem in the computing society, and it is often found at the heart of many contemporary applications such as data mining, search engine, information retrieval, and bioinformatics. The three major aspects of a pattern matching system are consisted of modeling of *pattern*s, *matching algorithms*, and *scanner architectures*. In typical scenarios, the user specified pattern is fed into a scanner which will perform content inspections against the input data stream to determine its presence or absence based on certain matching algorithms.

Patterns are representations of symbols that need to be matched (detected) from an input data stream. Widely used patterns include *fixed string*, *time series*, and *regular expression* (regexp), and their matching problems may vary in complexity due to their different levels of expressiveness. Matching algorithms are computing logics that accepts an input data stream to be matched with the patterns to detect presence of the patterns in the input data. A matching algorithm is called *exact* if it only generates a matching signal when the input data match the pattern exactly. Alternatively, the matching algorithm may be *approximate* if a certain number of mismatches are allowed in the matching process. A matching algorithm that supports *overlapped matching* can detect patterns embedded anywhere in the input data stream, because it treats any input data as the starting point of a new sub-stream that may potentially match the pattern, and it maintains simultaneous matching activities associated with all possible sub-streams. The patterns and matching algorithm needs to be transformed and configured into the formats most suitable for the scanner to match an input data stream at run time. Design

of the scanner architecture needs to take both speed and reconfiguration time into account to achieve the best balance between system performance and deployment time.

Previous works in the literature often focus on a single aspect of the three components of pattern matching systems, e.g., new matching algorithm design without implementation considerations for the proper scanner, or high speed scanner design without consideration of deployment time. Their partial design usually does not translate to a practical and complete pattern matching system. In contrast, in this dissertation we will visit all three components and show a complete view on design of a pattern matching system, which involves design factors such as algorithm design and tailoring for a target scanner architecture, and tradeoffs among various design options.

1.1 Design Challenges

In this dissertation we are mainly interested in complex patterns such as regexp and time series that can be re-represented in structures named *character class with constraint repetition* (CCR), which defines a set of *acceptable* characters and the range of times they can be matched for. CCRs can model complicated behaviors in compact forms, and its behavioral analysis is highly dynamic and recursive. Matching of CCR based patterns will become even more sophisticated in overlapped matching mode, where multiple matching processes for different sub-streams of the input data may coexist in the same CCR state. Therefore the matching algorithm needs to maintain dynamic matching activities of all existing matching processes, including, but not limited to, tracking matching progress, determining acceptance and asserting interaction signals between CCRs.

Pattern matching systems usually require fast response time, and we choose FPGA as the scanner architecture for its massive parallelism and outstanding performance. To achieve best performance, we try to exploit the parallelism provided by FPGA architecture as much as possible in design of the matching algorithm, which complicates the modeling and analysis of algorithm because it needs to handle concurrent executions of all CCRs. In addition to performance considerations, we note that many pattern matching systems need to update their patterns to deal with emerging situations and therefore the time required to update patterns on the scanner is critical, yet often not accounted for in the literatures. It is a major design challenge to achieve fast deployment, as FPGA is notorious for its time-consuming synthesis, placing and routing process.

Next we will discuss the requirements and technical challenges in detail for modeling of patterns, matching algorithms, and scanner architectures in the following subsections.

1.1.1 Pattern Expressiveness

The application of pattern matching dates back to 1960s when it is first used in text editors, and at that time the patterns involved are fixed strings of characters. These fixed-string patterns are easy to describe and understand, and their behaviors in the matching process can be easily modeled and captured, at the cost of a very limited level of expressiveness.

Since then, there had been an increasing demand on the ability to describe more flexible patterns, and regular expressions [1] are developed to accomplish this goal.

Regular expressions (regexps) are a sequence of characters with certain syntax and grammar specifications such that they can be used to describe a set of fixed strings conforming to certain criteria. Two most popular features are character class which defines a set of characters that are acceptable, and constraint repetition which is a quantifier defining the number of repetitions that a pattern can be matched. These two structures significantly expand the expressiveness in both dimensions of acceptable characters and their matching lengths. Their rich expressiveness has been widely applied to applications such as spam filters and network intrusion detection systems (NIDS) where a single regexp pattern can represent many variants of a fixed string pattern.

The aforementioned two types of patterns have been mainly used in computer related problems where signals are discrete and well defined. More recently, as sensor technologies advance, more and more data are sampled from real-world analog signals as time series which are a sequence of data points sampled consecutively in time with certain frequency, and there is a demanding need for mining of time series [2] patterns. Although they share the same general form as fixed strings, they are different from fixed string and regexp patterns because they have timing information embedded into the pattern itself, and the embedded tempos may vary from the ones of the input data stream. Time series patterns have a large domain of applications such as motion detection, handwriting recognition and melody matching.

1.1.2 Tolerance of Noises

In their early stages, most matching algorithms focus on exact matching of patterns, where any mismatches/errors will terminate the matching process and report a

mismatch. However, nowadays more and more systems are requiring the capability of approximate matching of patterns allowing certain amount of errors such as typos, signal noises, or mutations of DNA. As a result, new algorithms are developed to match patterns in an approximate manner with the capability to evaluate the similarity between patterns and input data streams under certain distance measure. These algorithms constitute the core of many scoring systems such as music retrieval and DNA sequencing.

Because exact matching algorithms do not need to maintain similarity information for the pattern during the matching process and all they need is a Yes or No match information, they usually have simper models and implantations than approximate matching algorithms. In the contrast, approximate matching algorithms have to keep track of the similarity measurements between the pattern and input data, and therefore they usually take more space to implement. Moreover, due to their early-termination policy, exact matching algorithms tend to have better performance than approximate matching algorithms because the later one needs to continue running even on a mismatch.

1.1.3 Overlapped Matching

Intrinsically, the pattern to be searched may appear at any position of the input data stream, and therefore all sub-streams of the input need to be inspected against the pattern to guarantee correct detections. This is referred to as overlapped matching, which means that every input data needs not only inspected as part of previous sub-streams, but also as the beginning of a brand new sub-stream, so that all possible sub-streams are

inspected and the complete search space is explored.

Many different ways have been developed to support overlapped matching, among which the simplest one is to initiate a new matching process at every position of the input data stream. This straightforward method will restart matching at position *i+1* if the sub-streaming starting at position *i* is determined to be a mismatch, leading to a significantly longer runtime that is *n* (the length of pattern) times slower, because in the worst case every matching process proceeds to the second to last position of the pattern and then fails and restarts. There are smarter algorithms that can re-use the information gathered from the previously failed matching process to avoid unnecessary work when starting the new matching process, such as the Knuth-Morris-Pratt algorithm [3] for fixed string patterns.

Similar algorithms exist for the more complex regexp patterns, where they are converted to non-deterministic finite automata (NFA) [4] that store the information of all on-going matching processes in their states. Due to the rich expressiveness and flexibility of regexp patterns, it is possible that multiple matching processes stay at the same state, and it is the NFA based matching algorithm's responsibility to resolve the ambiguities of all these concurrent matching processes started from different positions of the input data stream.

1.1.4 Architectures of Scanner

As silicon technologies evolve, more computing platforms have been developed to meet the needs of various problems. Von Neumann [5] based CPU architecture has been the traditional choice for most pattern matching algorithms, where the programs are

executed sequentially, and few, if not none, dependence issues need to be taken care of.

More recently, the community has seen a considerably growing trend of parallel computing, based on the massive parallelism offered by modern Graphic Processing Unit (GPU) [6] and Field Programmable Gates Array (FPGA) [7] devices. GPUs are based on the Single Instruction Multiple Thread (SIMT) [8] architecture where each core is responsible for execution of one thread and many cores run in parallel. The behaviors of cores are defined by the vendors, and they need to be programmed following the CUDA [9] or OpenCL [10] language specifications. The GPU cores are much more lightweight than traditional CPUs and the context switch time is significantly reduced such that many threads can be swapped in or out of the core and other threads can run when one is waiting for slow-to-access resources such as the global memory, i.e., hiding the latency. Although each GPU core is less powerful than a CPU, their parallel execution leads to a much higher processing throughput. The SIMT architecture implies that peak performance can be achieved only if all threads agree on the same execution path, otherwise multiple instructions must be issued for divergent execution paths and they are executed sequentially on different cores. This must be taken account into the design of the pattern matching algorithm.

Among the three, FPGA has the finest-grain of granularity where the scanner can be configured in the gate-level, and it is highly customizable which means that the designer has full control of every implementation detail. Programs executing on FPGA run on the bare hardware with minimum overhead caused by operation systems, drivers, and other abstraction layers. The designer is responsible for developing everything from

scratch including the circuitry of each function, as well as the interfaces and pipelines between function units to ensure correct timing. This is usually a much more tedious process than GPU and CPU based scanners because the later two have well-established and documented driver and application programming interface to handle the scheduling and communication of processes. However, FPGA usually achieves the best performance gain even if their operation frequency is slower than that of GPU and CPU, because the instructions are issued and executed in real-time single cycles.

1.1.5 Reconfiguration Time of Scanner

It is not uncommon to update the patterns after they have been fed into the scanner, such as the rule set update for network intrusion detection systems and the user query update for music retrieval systems. This implies implementing the new model as well as matching algorithm tailored for the new pattern onto the scanner architecture. For CPU and GPU based scanners, this is a simple matter of re-compilation of the source code which can usually be done in seconds. However, for FPGAs, the contents and routings of the hardware need to be reconfigured in the physical level, which translates to a reconfiguration time of hours.

Although CPUs and GPUs are easier and faster to reprogram, they may not be able to meet the performance requirement enforced by the applications. As for FPGAs, their time-consuming reconfiguration leads to long turnaround time which is unacceptable for online pattern matching engines such as NIDS or user-centric applications such as music melody matching system. Therefore, the reconfiguration time is of great concern in design of FPGA based scanner.

1.2 Research Work and Contribution

In this dissertation, we target at complex patterns such as regexps and time series and developed the corresponding matching algorithm for both exact and approximate matching. The models and algorithms for exact and approximate matching are illustrated using network intrusion detection system Snort and music melody matching system as examples, respectively. The matching algorithms are developed based on NFA to support overlapped matching, and FPGA is selected as the scanner architecture to exploit the intrinsic parallelism of the algorithm and achieve best performance. The scanner is designed into a modular architecture such that its parameters are stored in on-chip memories and therefore can be rapidly updated by simple memory writes.

Our research work and contributions regarding each component of the pattern matching system, i.e., pattern, matching algorithm and scanner architecture, are summarized as follows.

Pattern:

- General, uniform CCR module for regexp
- General, uniform ACCR module for time series
- Ease of parallelization

Matching algorithm:

- MIN-MAX algorithm for exact matching of regexp
  - Support of overlapped matching
  - Resolution of matching ambiguity
  - Parallel execution

- Elastic algorithm for approximate matching of time series
  - Evaluates similarity of pattern
  - Parallel execution
- Proof of correctness

Scanner architecture:

- FPGA based modular architecture
- Memory parameterizable
- Rapid reconfiguration in milliseconds

In addition to the abovementioned contribution, we also resolved other open problems encountered in the development of the matching algorithms, which will be detailed in their corresponding sections.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. In section 2 we visit the problem of exact matching for regexp patterns, targeting at the network intrusion detection system Snort. In section 3 we explore the problem of music melody matching, which can be modeled as approximate matching of regxp patterns. The dissertation is concluded in section 4.

2. REGEXP MATCHING SYSTEM*

In this section we discuss the design of a regexp pattern matching system. Regexps have been a popular way to extract the features of malicious behaviors of network intrusions, and in these applications they are often required to be matched exactly to claim detection of an attack. Also, the emerging large volume of internet traffic puts a high demand on the speed and throughput of the scanner. As such, we choose FPGA as the scanner architecture to meet the performance requirements, which imposes an extra design challenge to reduce the time-consuming reconfiguration process.

The following of this section is organized as follows. In subsection 2.1 we introduce some background information and related works in the domain of regexp matching system. We then propose our CCR model as the general modular building block to re-write regexps, and develop its corresponding matching algorithms in subsection 2.2. Design of the FPGA-based scanner is covered in subsection 2.3, and experiments and evaluations are conducted in subsection 2.4. Finally, subsection 2.5 summarizes the design of the regexp matching system.

2.1 Background and Related Works

Regular expression (regexp) matching is a pattern matching technique, where a set of patterns are represented in regular expressions, so that they can be used to check if the input string contains the target patterns. Various regexp matching engines have been

---

designed to achieve one or more specific objectives, e.g., speed, storage sizes, and complexity of patterns, etc. They are used for a wide range of applications: compilers, bioinformatics, and most recently Network Intrusion Detection Systems (NIDS) such as Snort [11], Bro [12], L7 [13] and spam filters such as SpamAssassin [14], etc. Rapid escalation of network security problems has led to explosive growth of the number and complexities of regexp based rules in these and other similar systems. For instance, the number of regexp rules in Snort has increased from 509 (Apr. 2006) [15] to 1667 (May. 2011), and over 60% of rules in SpamAssassin are involved with regexp matching [16]. Regular expressions have significantly expanded from their classical form [4] to support character class, which is a set of characters that are acceptable to a matching state, and constraint repetition, which is a quantifier defining the number of repetitions that a state can be matched. For example, both character classes (in 1253 regexps) and constraint repetitions including wild cards (in 1600 regexps) are heavily used in the latest version of Snort, and a similar conclusion can be made for SpamAssassin. When the two constructs are used together as a regexp term, it is called a Character Class with Constraint Repetition (CCR).

Regexp matching engines mainly fall into Deterministic Finite Automata (DFA) or Non-deterministic Finite Automata (NFA). Both NFA and DFA [4] have been widely studied for string matching. Memory efficiency is critical for conversion of an NFA to its equivalent DFA [17]. Rewriting and grouping of regexps were proposed to alleviate the state explosion problem of DFA [18].

DFA is a preferred choice for high-performance applications. It was used for content scanning of firewall [19] and spam filtering (conversion of SpamAssassin (SA) regexp rules into DFAs by JLex [20]). Ternary CAM was proposed for fast table lookup [21]. The notion of path ambiguity is the main cause of state explosion in DFA [22]. Modified CAM (m-CAM), memory banks, and data packing techniques were proposed to compact the DFA size [23]. Another table compression technique proposed in [24] can handle variable length inputs. The D^2FA explored states that share the same transition trigger input to reduce redundancy [25]. The δFA architecture stores only the differences between adjacent states to reduce the number of states and transitions [26]. S. Kumar et al. [27] used auxiliary variables to hold previous transition information. Merging non-equivalent DFA states by adding extra labels on those merged states' transitions was proposed in [28]. Other optimization techniques are also proposed, with the time complexity ranging from $O(n^2)$ in [25], [18], $O(n^3)$ in [22], to $O(n^3 \log n)$ in [28]. All these methods need to traverse the entire DFA state transition table to determine which transitions or states are qualified for optimization.

Overlapped matching requires the automata's starting state to be always *active*, i.e., checking for matching of every input character, so that each incoming character will be regarded as the starting of a new string. DFA architecture cannot support overlapped matching because it allows only one single active state. In contrast, NFA allows multiple active states to exist simultaneously, and therefore is better suited for matching of complex, dynamic patterns.

As regexp based matching algorithms are being used for advanced applications, parallel processing architectures, especially FPGAs, are being used for their implementation. Code generation proved to be an effective technique, e.g., translation of PCRE op-codes into VHDL codes [29]. A Python-based compiler was proposed to translate regexps into instructions for a VLIW architecture on FPGA [30]. Selection of basic building blocks is an important design issue for code generation. The four building blocks *single character*, *OR*, *concatenation*, and *kleene star* proposed in [31] have been widely used in different studies [15],[32].

A shift register based architecture was proposed in [15] for *{n,}* (constraint repetition for matching at least *n* times), *{m,n}* (constraint repetition for matching no less than *m* times and no more than *n* times), and *{n}* (constraint repetition for matching exactly *n* times) to count the matching repetitions of regexps with memory size of O(log *n*) for *{n,}* and O(*n*) for *{m,n}* and *{n}*. A counter based design proposed in [33] reduced the memory cost to O(log *n*), but this design does not support resolution of character class ambiguity, which refers to the condition that when an incoming character can be matched by (accepted to) two adjacent CCRs, the assignment choice may lead to different matching outcomes.. To the best of our knowledge, CES is the first scanner architecture to resolve character class ambiguity in overlapped matching mode, at the memory cost of O(log *n*) for constraint repetitions.

Resource minimization is important for design of FPGA based regexp scanners. Main techniques include sharing of logic (for sub-prefixes, -infixes, and –suffixes) [34], reduction of memory size [35], mitigation of resource explosion for large regexp rule

sets [36], and multi-character NFA decoder [37], etc. The authors of [38] explored several recently developed algorithms and techniques such as edge-minimization, alphabet- reduction and stride-increasing, and evaluated their joint application on FPGAs and memory-based ASICs.

2.2 CCR Matching Algorithms

In this section we first propose the CCR model as a general modular representation for regexp patterns in subsection 2.2.1, and then develop the MIN-MAX matching algorithm in subsection 2.2.2 based on the matching burst model. A further study on the dynamic behaviors of matching bursts reveals the fact that CCRs may suffer from matching collisions and lead to incorrect matching results, and in subsection 2.2.3 we develop several heuristic rules to analyze the regexp pattern and guarantee that matching collisions will never occur to regexps conforming to these rules. The formal proof of the MIN-MAX matching algorithm and the collision-free rules are detailed in Appendix A, and based on the rigorous proof process we developed several retrospection rules in subsection 2.2.4 to track back matching lengths after a match is claimed.

2.2.1 CCR Model

In this section, we propose a new structure, named *character class with constraint repetition* (CCR) to combine the two complex features of regular expressions and represent complex patterns in a compact format. To ease the discussion, we consider regexps consisting of only CCRs in our discussion. Consider a CCR based regexp R_n=CCR_1•…•CCR_n and an input string I_m={c_1, …, c_m}, where CCR_i has the general format of a character class CC_i followed by the constraint repetition $\{b_i^L, b_i^U\}$, and c_j is

the j^{th} input character of I_m. We use the term *cycle* to denote the duration for processing of one character by a matching system, i.e., c_j is processed at cycle *j*. The concept of cycle will also facilitate the subsequent discussion in temporal domain. Real world Perl/POSIX [39][40] regexps are usually not in pure CCR format. However, according to the grammar of *regular language* [39][40], they can always be rewritten as a sequence of CCR terms interconnected by '•' (CONCATENATION) or '|' (OR) operators. *Parentheses* (also known as *groups*) are not actual operators, but only used to force certain precedence orders.

We developed a syntax tree parsing tool to extract CCR terms and their operators from a Perl/POSIX regexp, where the syntax tree is optimized for easy mapping to the scanner architecture. *Tokenization* is the first step to translate a Perl/POSIX regexp to its equivalent CCR based representation. To start, the implicit concatenation operators are represented explicitly as '•''. For example, a Perl regexp based Snort rule,

R_1: [^\s\x0d\x0a]*\x2e\x2e(\x2f|\x5c),

is represented as the expression "[^\t\n\r\f\x0d\x0a]* •\x2e • \x2e • (\x2f | \x5c)". After tokenization, the syntax tree can be generated by backward parsing the token set. Succinctly put, the syntax tree structure has the following characteristics.

(1) Internal nodes represent logic operators ('•' or '|'), and leaf nodes represent CCRs.

(2) A sub-tree is constructed when a sub-regexp is surrounded within a pair of parentheses. The sub-root node represents the pair of parentheses.

(3) Internal nodes have two children nodes. If both of them are leaf nodes, then the left (right) child represents its parent logic operator's left (right) operand. Otherwise, the

left (right) child must be an internal (leaf) node which represents its parent logic operator's preceding operator (right operand).

After the syntax tree is generated, different types of constraint repetitions are translated into the "*Between*" format {*m*,*n*}, where *m* and *n* represent the lower and upper bounds of the constraint repetition, respectively. Using MAX_INT to denote the largest integer supported by the scanner, "*Exactly*" ({*m*}) is translated to {*m*,*m*}, "*AtLeast*" ({*m*,}) to {*m*, MAX_INT}, '*' to {*0*, MAX_INT}, '+' to {*1*, MAX_INT}, and '?' to {*0*,*1*}.

As an illustration, Figure 1 depicts the syntax tree for the following Perl regexp based Snort rule

R_1: [^\s\x0d\x0a]*\x2e\x2e(\x2f|\x5c)

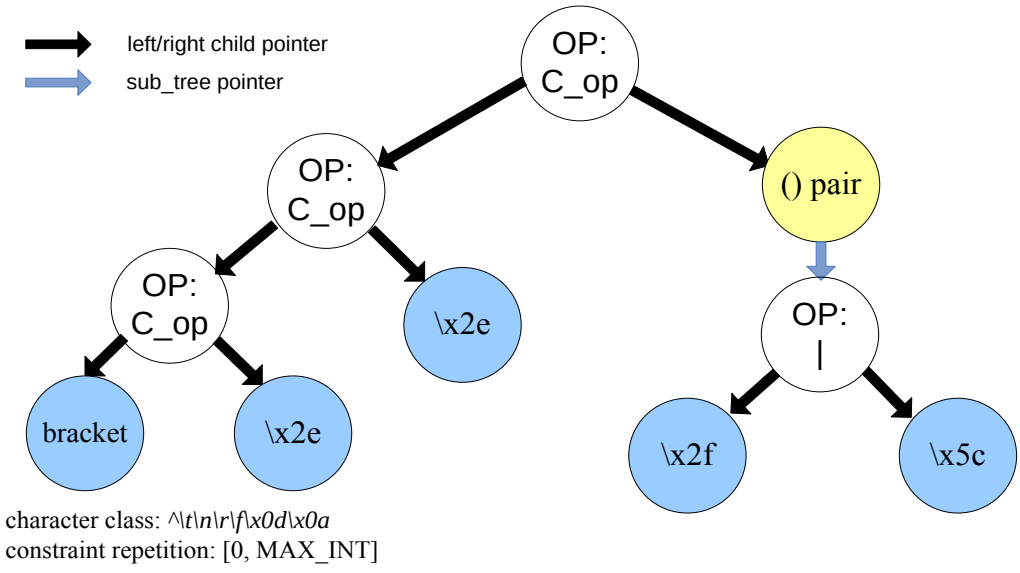

Figure 1. Syntax tree for R_1

2.2.2 MIN-MAX Algorithm

Matching of R_n against I_m is combination of an ordered matching problem (in which the tail of string cannot be matched before the head has been accepted), and a combinatorial search problem (where at each matched CCR location, one will need to decide to stay on the same CCR or move forward to the next one when both of them can accept the input symbol).

Succinctly put, we need to determine whether or not I_m can be split into substrings $S_n = \{s_1, \ldots, s_n\}$, such that for all s_i in S_n, where $1 \le i \le n$, s_i is *acceptable* to CCR_i. This means that element characters of s_i are all in CC_i and the length of s_i, denoted by L_i, falls in the range of $[b_i^L, b_i^U]$. Enumeration is the only way to get all feasible length assignments of L_i, and each feasible assignment is referred to as a *matching path*. We develop a combination of *interaction rules* between CCRs, and also *counting rules* for the number of matches made in a CCR for regexp matching. We further develop a *retrospection rule* to store useful information in the forward scanning direction, which can then be used by *matching length calculation rules* to calculate L_i in a backward fashion after a matching is declared at the last stage of the CCRs (i.e., retrospection of matching path).

While one or more matching paths may be found when a regexp is matched, it is cost prohibitive to find all feasible splits of I_m (i.e., retrospection of all possible matching paths). Instead, our design aims to generate one single matching signal at cycle m if I_m can match R_n, and this usually suffices for practical applications.

There has been research efforts [33] to use counters to check whether or not it satisfies the repetition constraint, which however may suffer from false negatives because it only searches part of the search space. In their design, the matching count of an active state CCR_i, denoted by $counter_i$, is incremented by 1 when an acceptable character arrives. CCR_i activates CCR_{i+1} to start evaluating subsequent characters when $counter_i$ reaches b_i^L. CCR_i becomes inactive when $counter_i$ reaches b_i^U or the input character is unacceptable. This scheme works fine when adjacent CCRs do not have any common acceptable characters, otherwise the design cannot resolve character class ambiguity. This is because after CCR_i matches a character c_i, and its repetition count has exceeded b_i^L but not b_i^U, it could continue to check for the next character c_{i+1}, or alternatively activate CCR_{i+1} to start checking from c_{i+1}. With only one single counter, there is no way to keep track of both possibilities, and therefore it is subject to false negatives.

This problem can be illustrated by the following example taken from the tracker ID rule of the SpamAssassin regexp rule set, where the three CCR terms have overlapped character class subset [a-z0-9]:

body TRACKER_ID /^[a-z0-9]{6,24}[-_a-z0-9]{12,36}[a-z0-9]{6,24}\s\z/is*

Let $R_2 = CCR_1CCR_2CCR_3$ denote the three CCR terms in this example, and we reduce their constraint repetitions to "{2,4}, {3,4}, and {2,5}" respectively for ease of discussion. That is, we use

$$R_2: [a\text{-}z0\text{-}9]\{2,4\}[\text{-}\_a\text{-}z0\text{-}9]\{3,4\}[a\text{-}z0\text{-}9]\{2,5\}$$

as the example throughout this section.

Table 1 gives an example to illustrate the false detection when character class ambiguity is present, where R_2 is used to match against the input string "*abc-1-_3d*". When the scanner processes character '*c*' which is acceptable to both CCR_1 and CCR_2, two possibilities exist: 1) '*c*' is matched by CCR_2 and $counter_2$ increments to one, and 2) '*c*' is matched by CCR_1 and $counter_2$ remains zero. Based on the lazy matching model, the matching engine in [33] will pick up option (1) and follow the searching path of {"*ab*", "*c-1*", "*-_3d*"}, which leads to the incorrect conclusion that the example string cannot match R_2 because "*-_3d*" is not acceptable to CCR_3. However, choice (2) will get the feasible searching path of {"*abc*", "*-1-_*","*3d*"}.

Table 1. Single counter example

abc-1-_3d	CCR_1	CCR_2	CCR_3
Counter	1→2→3→0→1 →0→0→1→2	0→0→1→2→3→4 →0→0→0	0→0→0→0→0→0 →0→0→0

To solve the character class ambiguity problem, we propose using two counters (MIN_i, MAX_i) to track the minimum and maximum number of characters that CCR_i may have matched. In other words, the two counters together define a feasibility zone of all possible counts of matching for CCR_i. When a new (or old) possible matching count of CCR_i needs to be added (or deleted), we can adjust the bounds of the feasibility zone by

simple adjustment of the two counters. The counting and interaction rules are designed to implement adjustments of (MIN,MAX) to guarantee correct matching outcome in the forward matching direction. The retrospection rule saves useful counter information which will be later used by matching length calculation rules to compute matching lengths in the backward direction. By relaxing the tighter condition "explicitly track every possible matching count" to "group all possible matching counts into a range", we avoid tracking every possible matching count for each CCR during the matching process, and the space complexity to implement constraint repetition of n is O(log n), as compared to O(n) for conventional NFA based methods [31][15]. As it will become clear shortly, this technique is guaranteed to match a regexp if it exists in the input string.

In addition to resolving character class ambiguity, overlapped matching is another important feature to eliminate subtle false negatives, as the example illustrated in [17] where the regexp of "*telephone* | *phonebook*" is used to match against the string of "*telephonebook*". Without overlapped matching support, only "*telephone*" can be matched. But with overlapped matching enabled, both "*telephone*" and "*phonebook*" can be matched. Overlapped matching is critical for detection of embedded strings such as malicious (executable) contents embedded in digital media packets. In the overlapped matching mode, CCR_1 needs to be permanently active so that it is always ready to match against a new incoming substring.

Modeling, analysis and proof of solutions for character class ambiguity and overlapped matching for CCR based regexps are highly recursive. To eliminate notational ambiguity, next we will first introduce the structure of a CCR engine and its

operational rules, i.e., the MIN-MAX algorithm, and then proceed with their modeling and analysis. For simplicity, we illustrate the algorithm using linearly concatenated CCRs, and support for CCRs ORed in parallel will be covered later.

CCR_i consists of two counters (MIN_i, MAX_i), two stored constraint repetition parameters (b_i^L, b_i^U), a memory-based character class CC_i, and an active-state flag $ACTIVE_i$. A CCR can be in one of three states: *idle*, *polling* and *busy*, where the latter two are collectively called the *active* state in later discussions. Externally, CCR_i has an input (output) activation signal AS_{i-1} (AS_i) from (to) CCR_{i-1} (CCR_{i+1}). The state transition diagram for a CCR_i is shown in Figure 2, in which CCR_i transits from idle to active when it receives an asserted AS_{i-1} (condition b). An idle CCR does nothing while an active CCR inspects input characters for matching. An active CCR remains in the polling state and makes no change to any counter, before it makes the first match (condition a). After that, it enters the busy state and updates its (MIN, MAX) counters. It activates its successor if the matching count satisfies the constraint repetition (condition c and d).

Note that there is a one cycle delay between "AS_{i-1} is asserted" and "$ACTIVE_i$ is set to one". That is, when AS_{i-1} is asserted after constraint repetition of CCR_{i-1} is satisfied, $ACTIVE_i$ will not be updated until the next cycle. Also note that the constraint repetition of CCR_{i-1} may be satisfied for a burst of consecutive cycles, in which case AS_{i-1} is fired at each cycle for the duration.

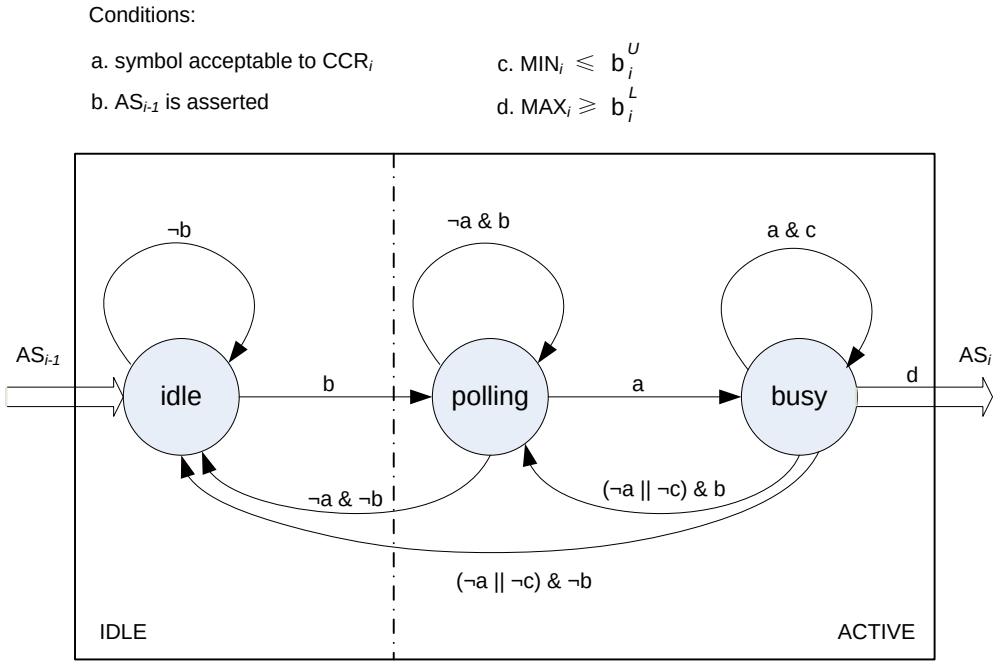

Figure 2. State transitions in a CCR

The state transitions in Figure 2 are derived from interaction rules and matches counting rules that run in each CCR. These two sets of rules in the MIN-MAX algorithm are used to control activation signals (AS) between CCRs and track matching counts for each CCR. They are formalized as follows.

CCR Interaction Rules:

IR-1 (a) CCR_1 is always set as active, i.e., $ACTIVE_1 = 1$, and (b) MIN_1 is always 0.

IR-2 AS_i is asserted when $MAX_i \geq b_i^L$ and $MIN_i \leq b_i^U$.

IR-3 When CCR_i asserts AS_i (activates CCR_{i+1}), $ACTIVE_{i+1}$ is set to 1 and MIN_{i+1} is reset to 0.

IR-4 An activated CCR_i uses three conditions to decide change of its state: (a) the incoming character is not acceptable to CC_i, (b) MIN_i has exceeded b_i^U, and (c) it receives an activation signal from CCR_{i-1}. When (c) holds, CCR_i remains active. When none of (a), (b) and (c) holds, CCR_i remains active. When (c) does not hold, and (a) or (b) holds, CCR_i changes to inactive state, and (MIN_i, MAX_i) counters are reset to 0.

IR-5 A match is reported when the final CCR asserts its activation signal.

Matches Counting Rules:

CR-1 When $ACTIVE_i$ = 1, MAX_i is incremented by 1 if the incoming character is acceptable to CC_i.

CR-2 When MIN_i is 0, it increments to 1 if (a) ($ACTIVE_{i-1}$, $ACTIVE_i$) = (1, 1) and the incoming character is acceptable to CC_i, but not CC_{i-1}, or (b) ($ACTIVE_{i-1}$, $ACTIVE_i$) = (0, 1), and the incoming character is acceptable to CC_i.

CR-3 After MIN_i becomes non-zero, it keeps increasing whenever the incoming character is in CC_i.

CR-4 MAX_i is reset to b_i^L when its value increases to MAX_INT, where MAX_INT is the highest value of (MIN and MAX) counters that can produce.

After the regexp matching reaches the last CCR, another set of rules is used for retrospection of matching lengths in each CCR. To support retrospection, (MIN_i, MAX_i) counters and current cycle will need to be saved into some memory before they are reset during the forward matching process, and details on that will be discussed in subsection 2.2.4.

To facilitate analysis of overlapped matching, next we introduce the notion of *matching bursts* based on structures of the input string. By definition, CCR_1 needs to be set permanently active in the overlapped matching mode (i.e., IR-1.a) to check every input character. When CCR_1 matches a consecutive sequence of acceptable characters, it remains at the busy state and MAX_1 keeps increasing per CR-1, where each increment represents a new (overlapped) matching process. CCR_1 will reset its MAX_1 to 0 when it receives an unacceptable character *x*. MAX_1 will increase to one again at arrival of its next acceptable character, which also indicates starting of a new matching process. In other words, an input string S can be modeled as *blocks* of characters B_i acceptable to CCR_1, segmented by *gaps* of characters G_i that are not acceptable to CCR_1, i.e., $S_{gb} := G_0, B_1, G_1, B_2, G_2, \ldots. B_i$ is associated with a burst of matching processes, *matching burst* MB_i, where each of these processes is started at a different input character location. These matching processes can correctly advance in parallel only when they do not cause incorrect changes of the configuration (locations, states of CCRs) of any other matching process.

A matching burst MB_i is *started* at the first character of B_i, and it is *terminated* when its matching front and tail become idle, where *matching front* MF_i denotes the active CCR in MB_i that is located most close to the final state, and *matching tail* MT_i denotes the busy CCR in MB_i that is located most close to CCR_1. After the birth of MB_i, MT_i stays at CCR_1 throughout the cycles of B_i. Its location "advances" to CCR_k at the beginning of G_i, where CCR_k becomes busy and there is no active CCR between CCR_1

and CCR_k. In the meantime MF_i continuously checks input characters to advance as far as possible.

Figure 3 illustrates state transitions of CCRs with respect to two hypothetical matching bursts. In this figure, the input character sequence is read from left to right (the *x* axis on the top row) and the CCR regexp R_n is matched from top to down (the *y* axis on the left column). The (busy/idle/polling) states of CCRs corresponding to input characters are marked in different colors, column by column with respect to each input character. The input characters are divided into blocks and gaps based on their acceptance to CCR_1, and each block of S_{gb} represents an input character. Each block of regexp R_n represents a CCR term. The matching tails are denoted by solid lines, and matching fronts are denoted by dotted lines in the CCR state diagram. The color of CCR_1 changes to red (green) at the first character of each block (gap).

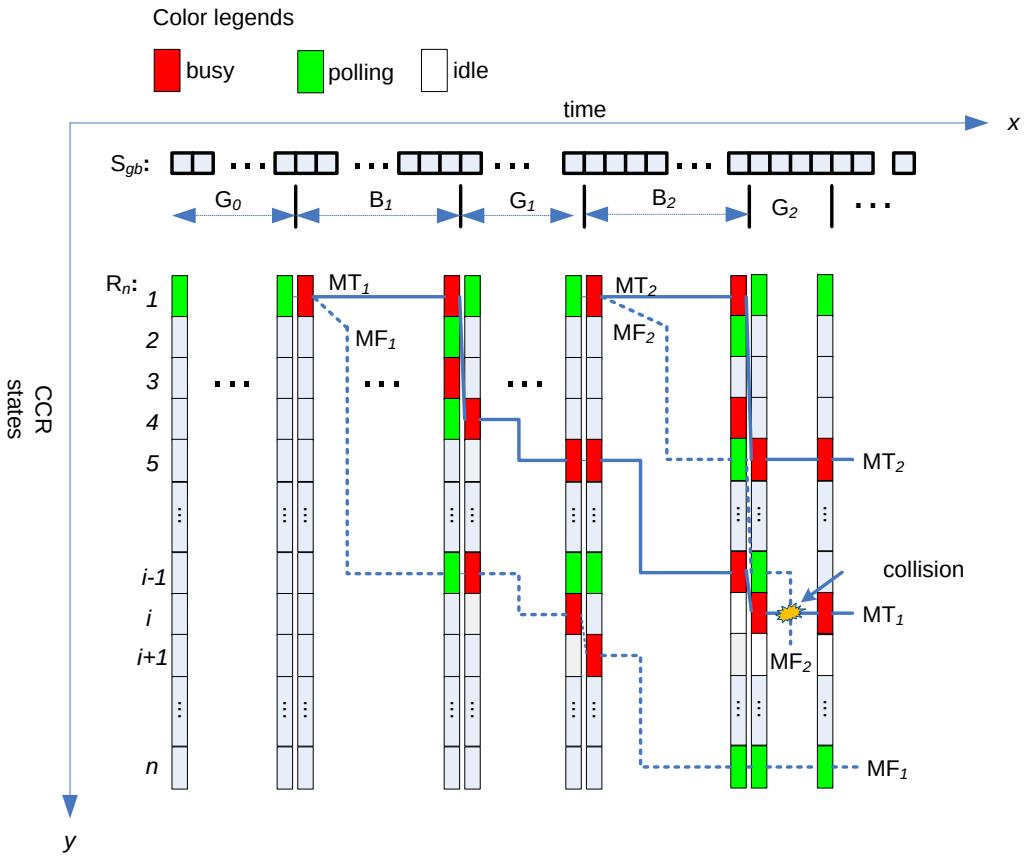

Figure 3. Illustration of matching burst, front, and tail

In general, the location of MT_i advances monotonically with time, but the location of MF_i may move forward (at acceptance of input characters) or backward (at rejection of an unacceptable character). Once AS_n is asserted after MF_i reaches CCR_n, a match of the regexp is detected, i.e., IR-5. For the two illustrated matching bursts, a collision is marked at the last column (intersection of MT_1 and MF_2), when MF_2 attempts to enter CCR_i which is still occupied by MT_1, i.e., a *collision* between MB_1 and MB_2. Here, CCR_i is still performing string matching for MB_1 while its predecessor

CCR_{i-1} has reported another matched substring in MB_2, and wants to reset CCR_i by asserting AS_{i-1}. It will become clear in the detailed discussion of next subsection that the burst level model significantly simplifies analysis of collisions and subsequent derivation of collision detection criteria.

Unlike traditional NFA/DFA based designs, which usually assign a unique state to each possible matching count of CCR_i, in the MIN-MAX algorithm we use a pair of counters (MIN_i, MAX_i) to represent the minimum and maximum number of characters that CCR_i can match in all matching processes within a matching burst. This way, when a new (old) possible matching count of CCR_i needs to be added (removed), we do not need to store (delete) it to (from) a dedicated register, but to relax (tighten) the bounds of CCR_i so as to accommodate the new matching count. Satisfaction of the constraint repetition of CCR_i is determined by the semantics of (MIN_i, MAX_i) counters, i.e., the maximum possible matching count has to be no less than b_i^L and the minimum possible matching count has to be no more than b_i^U (i.e., IR-2). Otherwise, the substring matched by CCR_i is either too short or too long. This technique reduces the resource requirement from the worst-case complexity $O(n)$ (for traditional NFA) or $O(2^n)$ (for conventional DFA) to $O(\log n)$, where n is the upper bound of a constraint repetition.

Regarding matches counting rules, MAX_i can begin to increase when CCR_i becomes active and the input character is acceptable (i.e., CR-1). This means that at least one matching process in the current matching burst has advanced to CCR_i, so that its state changes to busy. By IR-3, MIN_i remains 0 when AS_{i-1} is asserted, which implies that the matching tail has not reached CCR_i and some matching process of the current

matching burst might be still staying at an earlier CCR. It increases to 1 only when CCR_{i-1} gets a mismatch or MIN_{i-1} reaches b_{i-1}^{U} (i.e., IR-4 or CR-2), which means that the matching tail must advance to CCR_i and no matching processes of the current matching burst can stay at an earlier CCR any longer. Thereafter, the matching tail cannot move backward and has to remain in CCR_i or its successors, which leads to CR-3 that a non-zero MIN_i needs to be incremented by 1 for each input character acceptable to CC_i. To support the semantics of unbounded wildcard such as "*", "+", and "$\{b_i^L,\}$" with a limited upper bound MAX_INT, MAX_i is reset to b_i^L whenever it reaches MAX_INT (i.e., CR-4), so that CCR_i can match as many characters as possible and will always be activating CCR_{i+1} once MAX_i reaches b_i^L.

As an example, when R_2 is used to match against the example string of "*abc-1-_3d*" using the MIN-MAX algorithm, state transitions and (MIN_i, MAX_i) counter values are illustrated in Table 2. After processing the whole string, MAX_3 is 2 which is equal to b_3^L, indicating a successful match according to IR-2 and IR-5.

Table 2. State transitions of CCRs based on MIN-MAX

abc-1-_3d	CCR_1	CCR_2	CCR_3
ACTIVE	1→1→1→1→1 →1→1→1→1	0→0→1→1→1→ 1→1→0→0	0→0→0→0→0 →1→1→1→1
MIN	0→0→0→0→0 →0→0→0→0	0→0→0→1→2→ 3→4→0→0	0→0→0→0→0 →0→0→1→2
MAX	1→2→3→0→1 →0→0→1→2	0→0→1→2→3→ 4→5→0→0	0→0→0→0→0 →0→0→1→2

2.2.3 Matching Collisions and Collision-free Conditions

An analysis of IR-3 and CR-3 reveals that resource contention may occur when a matching process wants to increment MIN_i while another one attempts to reset it. This contention may arise between matching processes of different matching bursts, or within the same one. We will discuss collisions between adjacent matching bursts, and then those within the same matching burst.

Two matching bursts have a *collision* when the front of a new matching burst runs into the tail of an earlier matching burst, as the example illustrated in Figure 3, where the collision occurs at CCR_i at certain point in G_2. At collisions, the "minimum number of matching counts" of a matching burst is lost, which may lead to false detections. It is easy to verify that a false positive will occur when R_2 scans a (non-matching) string "*ab_def_44*", as shown in Table 3. This is because when MB_1 (starting from '*a*') goes past "*ab_de*", MIN_2 is reset from 3 to 0 as CCR_1 detects an overlapped matching of substring "*de*" in MB_2 (starting from '*d*'). As a result, MIN_2 fails to observe that substring "*_def_*", which is considered to match CCR_2, has the length of 5, exceeding its upper bound.

Table 3. False positive with overlapped matching

ab_def_44	CCR_1	CCR_2	CCR_3
ACTIVE	1→1→1→1→1→ 1→1→1→1	0→0→1→1→1→ 1→1→1→1	0→0→0→0→0→ 1→1→1→1
MIN	0→0→0→0→0→ 0→0→0→0	0→0→1→2→3→ 0→1→2→3	0→0→0→0→0→ 0→0→0→0
MAX	1→2→0→1→2→ 3→0→1→2	0→0→1→2→3→ 4→5→6→7	0→0→0→0→0→ 1→0→1→2

In [41] we proposed using a pair of checkpoint registers to save the contents of (MIN, MAX) counters in order to support two contending matching bursts. One may add more checkpoint registers to handle multi-collisions in a CCR as needed. That being said, we observe that MIN-MAX can be used to directly support overlapped matching without additional checkpoint registers if the regexp is inherently free of collisions. Our case study on two major network security systems Snort and SpamAssassin suggest that indeed most of their rules are free of collision, and therefore MIN-MAX can work in the overlapped matching mode for most of their regexp rules. Next, we will discuss novel techniques that can prove absence of collisions in regexps.

On the basis of the matching burst model, we conduct some sufficient condition based analyses and derive two heuristic lemmas for identification of collision-free CCRs. A third lemma is derived to identify CCRs that will not suffer from resource contention between any two matching bursts, even if collisions are present. CCRs conforming to these three lemmas are called *safe*, because they will not generate false

detections in the overlapped matching mode. These three lemmas can cover over 99.8% of regexp rules for Snort and SpamAssassin in our evaluation.

In the following descriptions, the abbreviations PR and CN stand for *premise* and *conclusion*, respectively. Detailed meanings of each condition are listed as follows:

PR_i.(a) : None of CCR_i's predecessors has a collision.

PR_i.(b) : CCR_i has a predecessor CCR_x ($1 \leq x < i$) whose character class has no intersection with CCR_{x+1}, CCR_{x+2}, …, and CCR_i, and $b_x^L \neq 0$.

PR_i.(c) : Any of the following conditions is satisfied: (1) $b_i^U = 1$, (2) $b_i^L = b_i^U$, or (3) $b_{i-1}^L \geq b_i^U$".

PR_i.(d) : $b_i^U = \infty$.

CN_i : CCR_i is collision free.

CN_i' : CCR_i is safe and will never suffer from false detections.

Lemma 1. *CN_1 always holds. For $1 < i \leq n$, when premises PR_i.(a) and PR_i.(b) hold, CN_i holds.*

Lemma 2. *For $1 < i \leq n$, when PR_i.(a) holds and PR_i.(c) holds, CN_i holds.*

Lemma 3. *For $1 < i \leq n$, when PRi.(d) holds, CN_i' holds even if CN_i does not hold.*

Formal proofs for these three lemmas are given in Appendix A.(1,2,3).

In summary, Lemmas 1 and 2 separate collision-free CCRs from those subject to collisions by analyzing character classes (i.e., PR_i.(b)) and constraint repetitions (i.e.,

PR_i.(c)) of CCR terms, respectively. Then, for these collision-prone CCRs, Lemma 3 determines whether or not they will lose useful state information at actual collisions.

The above discussion covers collisions between adjacent matching bursts, and next we will consider collisions between matching processes within the same matching burst. Note that in the burst level view, there is only one actual matching front and tail, but in between there may be multiple active CCRs that are segmented by idle CCRs. As such, an intermediate active CCR_i with non-zero MIN_i may receive an activation signal from CCR_{i-1}, which leads to a resource contention to MIN_i. A common feature of matching tails is that if it is not located in CCR_1, its MIN register must be non-zero. Therefore, we can regard CCR_i as a *pseudo* matching tail and apply the same analysis as conducted above to determine whether CCR_i is safe or not. If one of the criteria is satisfied, no false detections will occur.

A regexp is collision free if all of its CCRs are collision free. The testing process to determine absence of matching collisions for a regexp is summarized as follows. Note that CN_i implies PR_{i+1}.(a), and Lemma 1 and 2 shows an inductive relationship:

$$PR_i.(a) \xrightarrow{PR_i.(b|c)} CN_{i+1} \rightarrow PR_{i+1}.(a),$$

where the symbol "→" denotes that the derivation holds, and "$\xrightarrow{PR_i.(b|c)}$" denotes that the derivation holds if PR_i.(b) or PR_i.(c) holds. Thus, by testing PR_1.(b|c), PR_2.(b|c), …, PR_{n-1}.(b|c), we can obtain true/false results for CN_2, CN_3, …, CN_n. Knowing that CN_1 always holds, when CN_i holds for $1 < i \leq n$, one can conclude that the regexp is collision free, regardless of the input string. For regexps that failed the above collision check, Lemma 3

can be further applied to check their violating CCR terms and determine if the safety property is preserved.

We applied the above tests to linear concatenated regexp rules of Snort 2.9.5 and SpamAssassin 3.3.1, and the results are listed in Table 4. Remaining regexp rules including ORed sub-regexps and context-dependent features (i.e., zero-width patterns and back-references), which will be discussed in later sections and Appendix B, respectively.

Table 4. Statistics of regexp rule sets

Counts of	Snort	SpamAssassin
All regexp rules	1667	1917
Linear regexp rules	1027	1344
Regexp rules subject to collisions (by Lemmas 1&2)	30	33
Regexp rules subject to false detection at collision (by Lemma 3)	2	3

The results show that Lemmas 1 and 2 can identify 997 collision free rules from a total of 1027 linear regexp rules in Snort, and also 1311 collision free rules of 1344 linear regexp rules in SpamAssassin. For the 30 (33) linear regexp rules subject to

collision, only 2 (3) of them are not guaranteed to be false-detection free in Snort (SpamAssassin), and would need either checkpoint register [41] or shift register [15] type of scanner support. 1025 (1341) of the 1027 (1344) linear regexp rules in Snort (SpamAssassin) can be directly support by MIN-MAX for overlapped matching.

2.2.4 Retrospection of Matching Lengths

At this point, we have discussed regex *matching* in the forward scanning direction. Now we focus on *retrospection* of one matching path after the matching signal is triggered at the last stage of CCRs. As stated earlier, we only aim at finding one feasible matching path, rather than enumerating all possible ones.

Let $S_n = \{s_1, s_2, \ldots, s_n\}$ denote a set of consecutive string segments acceptable to $R_n = CCR_1 \bullet CCR_2 \bullet \ldots \bullet CCR_n$, where s_i is acceptable to CCR_i and the length of s_i is denoted as L_i, $1 \leq i \leq n$. A string s_i is acceptable to CCR_i if and only if each character of s_i is acceptable to CC_i, and L_i falls in the range of $[b_i^L, b_i^U]$. Next, we will give two theorems which guarantee correctness of the MIN-MAX algorithm.

Theorem 1. *If R_n is collision-free and S_n is acceptable to R_n, MIN-MAX will report a match. (Sufficient condition)*

Theorem 2. *If R_n is collision-free and MIN-MAX reports a match, there must have been a string S_n that has been matched by R_n. (Necessary condition)*

Theorem 1 and 2 guarantee that MIN-MAX algorithm will report a match if and only if S_n is acceptable to R_n, and thus MIN-MAX is free of false negatives and false positives. Formal proofs of these two theorems are given in Appendix A.4.

A byproduct of the proof process of Theorem 2 is for MIN-MAX to retrospect one feasible matching length configuration (matching path) when a matching burst triggered the matching signal. To enable this feature, a memory stack B_i is needed for CCR_i to store 2-tuple *B-code*: (current cycle value, MAX_i) when CCR_i has been matched and AS_i is asserted. MIN registers are not included because they are not used for matching length calculation. The choices of L_i values can be summarized into the following rules based on the value of MAX_i when AS_i is asserted.

Matching Length Calculation Rules:

LR-1 $L_i = MAX_i$ if $b_i^L \le MAX_i \le b_i^U$ at assertion of AS_i.

LR-2 $L_i = b_i^U$ if $MAX_i > b_i^U$ at assertion of AS_i.

When retrospection is needed, it is necessary for CCR_i to store B-codes at each assertion of AS_i in a stack B_i so that they can be used in later length calculations. That is, the following retrospection rule needs to be added in the forward matching stage.

Retrospection Rule:

RR-1 When AS_i is asserted, push the B-code of CCR_i into the B_i stack.

One can retrospect (L_n, L_{n-1}, …, L_1) along the backward direction of CCRs, starting from the last stage CCR_n. We will use R_2 of subsection 2.2.2 to illustrate the process. Even though R_2 does not conform to the tests of Lemmas 1-3, it does not produce false detection for the example input string in Table 2, because no collision would occur to this case. B-codes and other state information generated from the matching process along the forward direction of CCRs are given in Table 5. RR-1 is applied at the cycles when activation signals are asserted, and a new entry is pushed into

the B-code stack for every such cycle. For example, CCR_1 asserts AS_1 to CCR_2 at cycle 2, 3, and 9, according to IR-2, and the MAX counter values at each corresponding cycle are 2, 3 and 2, respectively. Therefore by RR-1, we got a B-code stack B_1 of (9,2), (3,3) and (2,2) from top to down. B_2 and B_3 are generated in a similar manner.

Table 5. Backtracking procedure

abc-1-_3d	CCR_1	CCR_2	CCR_3
ACTIVE	1→1→1→1→1→ 1→1→1→1	0→0→1→1→1→ 1→1→0→0	0→0→0→0→0→ 1→1→1→1
MAX	1→2→3→0→1→ 0→0→1→2	0→0→1→2→3→ 4→5→0→0	0→0→0→0→0→ 0→0→1→2
Activation Signal	0→1→1→0→0→ 0→0→0→1	0→0→0→0→1→ 1→1→0→0	0→0→0→0→0→ 0→0→0→1
B-code Stacks (B_1, B_2, B_3)	(9,2) (3,3) (2,2)	(7,5) (6,4) (5,3)	(9,2)

Next we will illustrate the retrospection process in a backward manner, starting from B_3 of CCR_3 to B_1 of CCR_1 and applying the matching length calculation rules correspondingly.

B_3: At cycle 9, from the top of B_3, we can conclude that LR-1 is satisfied and thus $L_3 = MAX_3 = 2$. Next, we traverse to B_2 of CCR_2, and set the cycle value to 7= 9-L_3.

B_2: At cycle 7, B-code at the top of B_2 satisfies LR-2, because $MAX_2 = 5 > b_2^U$, and thus $L_2 = b_2^U = 4$. Next, we traverse to B_1, and set the cycle value to 3=7- L_2.

B_1: At cycle 3, going down from the top of B_1, its second entry has the cycle value of 3. It satisfies LR-1, and therefore $L_1 = MAX_1 = 3$.

In summary, the retrospected matching path is that {“abc”,”-1-_”,”3d”} matches CCR_1•CCR_2•CCR_3, respectively. To the best of our knowledge, the MIN-MAX algorithm is the first work that is able to report the matching lengths of constraint repetitions with support of overlapped matching for collision free regexps. However, for simplicity of implementation, we did not include this retrospection feature in the scanner design reported in this dissertation.

2.3 CES Scanner Design

In this section we discuss the design of the CCR based regExp Scanner (CES) targeting at FPGA implementation. We first describe the internal architecture of a CCR module in subsection 2.3.1, and then discuss the interconnection of CCR modules in CES and the mapping of a regexp pattern to CES in subsection 2.3.2.

2.3.1 CCR Architecture

The architecture of a CCR engine is shown in Figure 4. The MIN-MAX algorithm is implemented in the *MIN-MAX Logic* module. Counters, registers, and activation signals are updated based on the aforementioned operational rules.

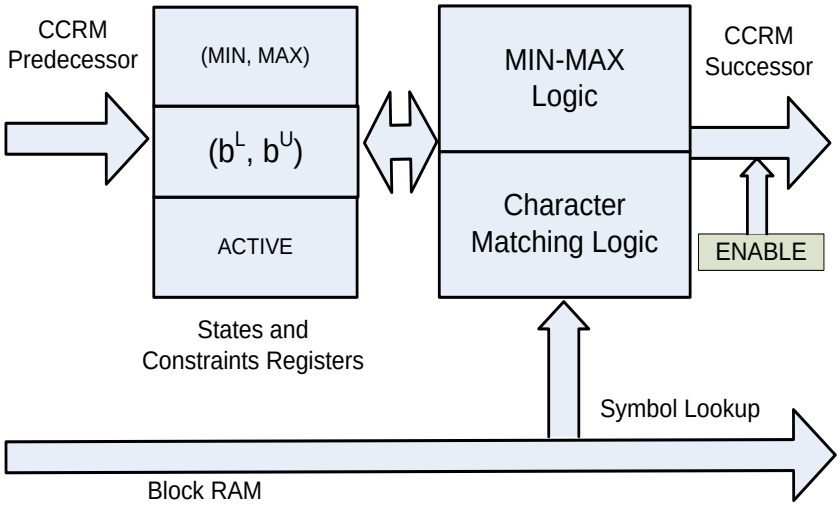

Figure 4. The architecture of a CCR engine

For implementation of the *Character Matching Logic*, we used a technique first proposed by [35][42] to store 72 CCRs worth of information in one Xilinx Block RAM (BRAM), which has 18Kbits on the Virtex5 device. By configuring BRAM to be 72 bits wide and 256 entries deep, and using the 8-bit ASCII value of the input character as the memory address, we can get the 1 bit accept/reject information for 72 CCRs in one memory read. This way, a single BRAM block is shared among 72 CCRs. In all, 72**n* CCRs fed by *n* BRAMs can simultaneously check whether the incoming character is acceptable or not.

The *ENABLE* bits are employed to switch different activation signals on or off when OR ('|') operators are present in the regexp. OR operators require the ability to track multiple searching paths concurrently. As shown in Figure 5, which represents the

regexp R_3 below, splitting of CCR terms by the OR ('|') operator can be implemented by a fan-out topology, combined with an enable register.

$$R_3: CCR_1(CCR_2|CCR_3|CCR_4)CCR_5$$

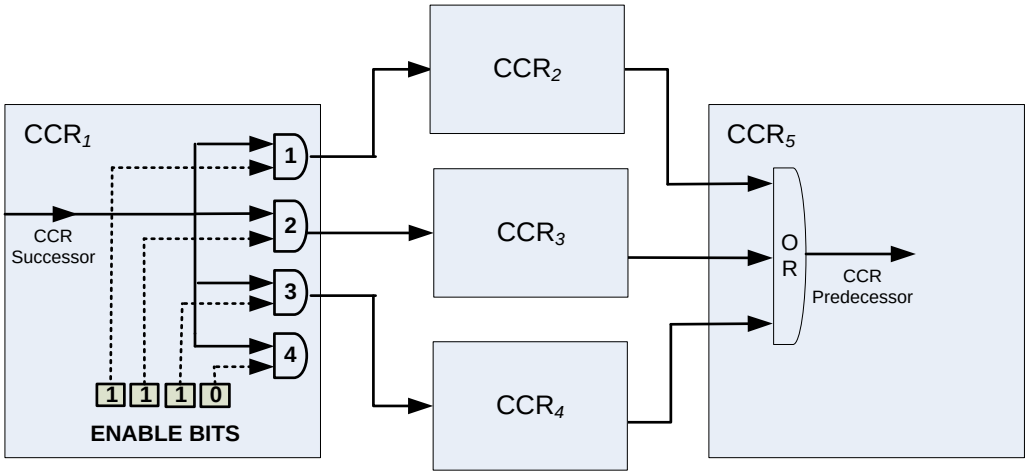

Figure 5. Interconnection architecture of R_3

By setting the three most significant bits of the ENABLE register to 1, the matching process propagates to CCR_2, CCR_3 and CCR_4 when CCR_1 asserts AS_1. It then converges back to CCR_5 when any of them gets matched.

The CCR engine can be extended to support context-dependent patterns, including zero-width patterns and back-references. However, as indicated by Table 6, there are much fewer this kind of regexps than linearly concatenated regexps in real

world applications. Thus we only give conceptual designs (see Appendix B), but did not include them as a generic element in our implementation of CES.

Table 6. Number of context-dependent regexp rules

# of	Snort	SpamAssassin
All Regexp Rules	1667	1917
Zero-Width Patterns	187	482
Back-References	66	20

2.3.2 CES Architecture

To implement a regexp rule set on CES, we first analyze its statistical information such as the lengths and widths in terms of CCR. Based on the rule level analysis, we partition the rule sets into subgroups of different topologies, e.g., long concatenated CCRs and wide ORed CCRs. Various topologies of CES are implemented in different FPGA chips, and regexps of similar topologies can be clustered together into the same CES. For implementation of each regexp rule, it is first parsed into CCR syntax trees as in subsection 2.2.1, which will then be mapped onto an appropriate CES. Mapping and interconnecting configurations are both represented by memory bits which can be downloaded to CES by simple memory writes. For the communication between PC and FPGA, we employed the Simple Interface for Reconfigurable Computing (SIRC) [43], an extensible software-hardware communication and synchronization API

developed by Microsoft.

As an illustration, we converted Snort rules into their CCR forms and analyzed their lengths and widths distribution to get a more in-depth understanding on characteristics of real-world regexps. We partitioned the rule set into several subgroups based on the statistics shown in Figure 6.

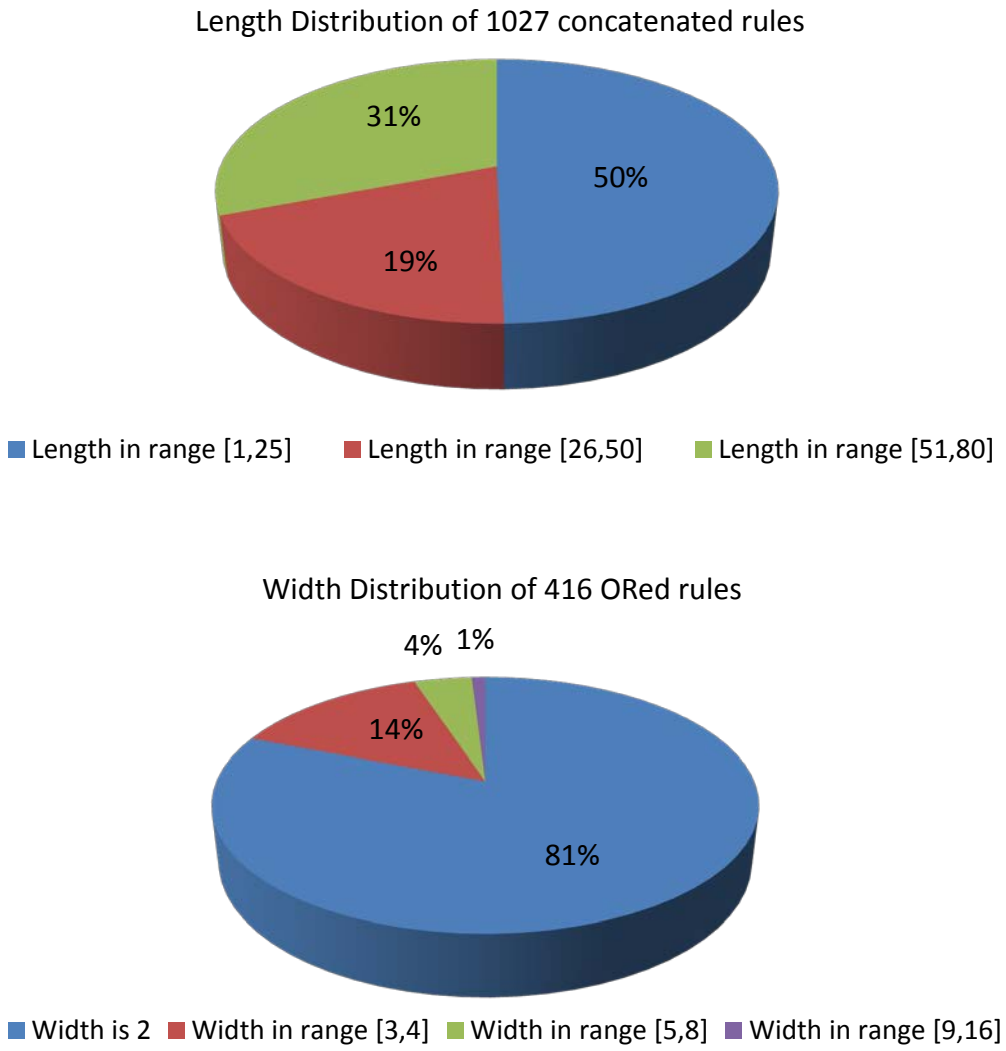

Figure 6. Statistics of Snort rule set (accessed September 2009)

It is more efficient to use linear (or parallel) CES topologies to implement concatenated (or ORed) CCR based regexp rules, respectively. Within the same type of topology, for example, linear topology, CCR rules with very different lengths should be assigned to different CESes to improve area efficiency. From the statistics we can see that most linear rules have less than 50 CCRs, and the majority of parallel rules have width of 2. We implemented code generators for both CCR engine and CES topology for a specific group of regexps. For example, one group has purely linear rules with lengths under 25, and the code generator will produce Verilog code for the CCR engine with no fan-out routes. It then generates code for the CES topology which consists of rows of 25 concatenated CCR engines, each row implementing a regexp rule. For the other group with width 2, the code generator produces another CCR engine with two fan-out routes and builds upon it another CES topology. This way, both area efficiency and modularity of CES can be conserved at the same time.

The PCRE regexp rules are then parsed to CCR syntax trees by the parsing tool introduced in subsection 2.2.1. CCR terms and logical operators are assigned to leaf and non-leaf nodes of the syntax tree, respectively. This phase is fast: on a commodity dual-core PC, the syntax tree generation time for 1667 Snort regexp rules was 125.3 milliseconds, and for 1917 SpamAssassin regexp rules was 77.4 milliseconds.

The information from the CCR syntax tree is then read and translated into memory bits by a CCR mapper utility, which is responsible for generating all configuration bits of the CCR engines, including character classes, constraint repetitions, and enable bits. The topological information of the regexp rule is acquired by traversing

the syntax tree in inorder. In the inorder traversal, a '•' operator causes the right child CCR node to be placed in the next position of current row (conceptually, to the right of the previous CCR). An '|' operator causes the right child CCR node to be placed in the next position of current column (conceptually, below the previous CCR). Sub-regexps bounded by an OR operator are placed in the same column, with each sub-regexp placed in a different row. Activation signal path from their common predecessor CCR engine controls all rows simultaneously. When lengths of branches expanded from an OR operator are different, we configure the unused CCR engines on the shorter branch to bypass mode so that activation outcomes of different branches can be routed at equal length and they do not need to wait for each other, as the example shown in Figure 7 to implement R_7:

$$R_7: CCR_{11}(CCR_{12}CCR_{13}|CCR_{22})CCR_{14}$$

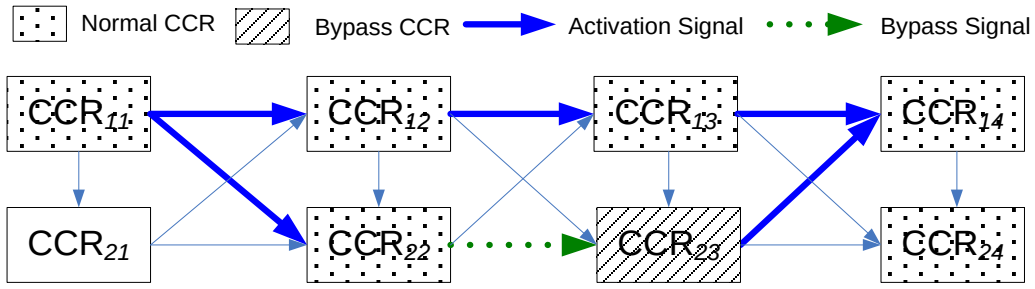

Figure 7. Implementation of R_7

The CES controller is responsible for chip level management and communications with the PC. We employ SIRC as the Ethernet communication channel, which has a software API for the PC side and a hardware API for the FPGA side. It features an input buffer to hold input strings and CCR configuration information from the PC side, and an output buffer to latch the matching results from FPGA. After receiving each packet, the CES controller is responsible for dispatching the input string or CCR configuration information to each CCR.

As of now, our discussion on CES has been based on a modular architecture, so that a regexp rule can be updated rapidly because its configuration information can be translated into memory bits and downloaded into FPGA in milliseconds when it can be embedded into the base CES topology. In other application contexts, for example when rule updates are infrequent, one can trade the fast reconfigurability for more regexp rules implemented in the same FPGA fabric.

2.4 Experiments and Evaluation

In this section we conduct some experiments to evaluate the performance of CES. Subsection 2.4.1 shows the synthesis results reported by the FPGA design tool, and live experiments are conducted in subsection 2.4.2. Finally a case study on Snort is discussed in subsection 2.4.3.

2.4.1 Synthesis Results

We have performed simulations [41] and live tests on a Virtex 5 LX110T device (17280 slices, each containing four LUTs and four flip-flops). The synthesis, place and route of CES are implemented by the *ISE 10.1* software running on an Intel Core 2 Quad

Q6600 PC with 4GB of memory. The FPGA and PC are interconnected over a 1 Gigabit Ethernet connection.

Using Xilinx® Synthesis Technology (XST) as the synthesis tool, we tested the CES with linear topology of length 25 and the synthesis report shows that on average one CCR utilizes 13 registers and 20 LUTs, and can run at a maximum frequency of 321.337 MHz, corresponding to 2.57 Gbps of processing throughput. The timing performance drops slightly with increases in the length of the concatenation chain, i.e., see Figure 8.

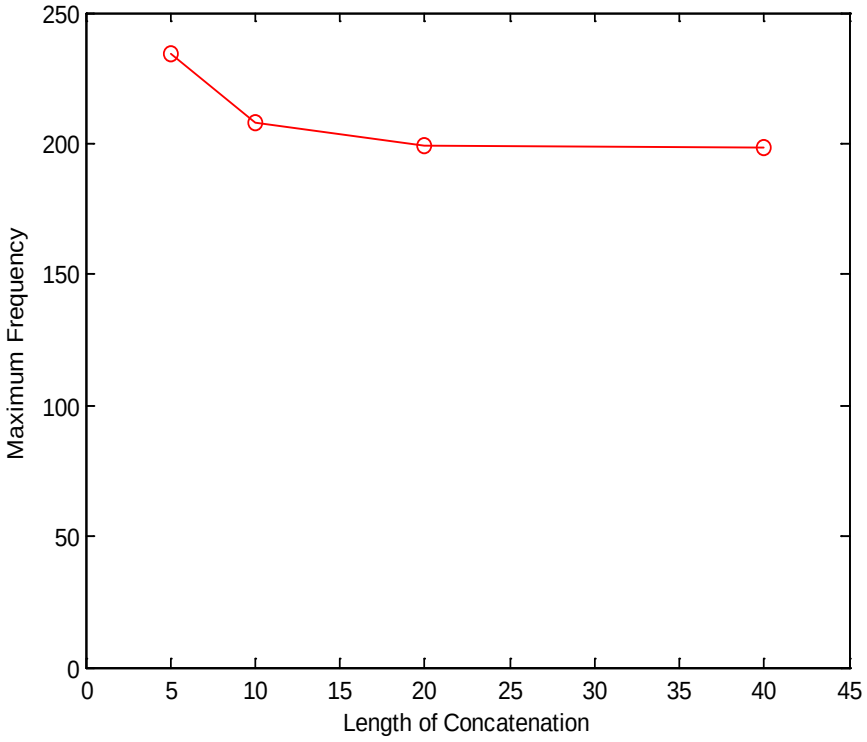

Figure 8. Maximum frequency vs. length of concatenation

Next, we examined the relationships between the CES size and the required FPGA area, and the FPGA implementation (synthesis, map and place and route) time. The hardware size vs. the number of CCRs is given in Figure 9, which clearly shows a linear relationship between the number of CCRs and the amount of resources (slice registers and LUTs). On the other hand, the implementation time grew sharply when the number of CCRs approaches the board's capacity, as illustrated in Figure 10.

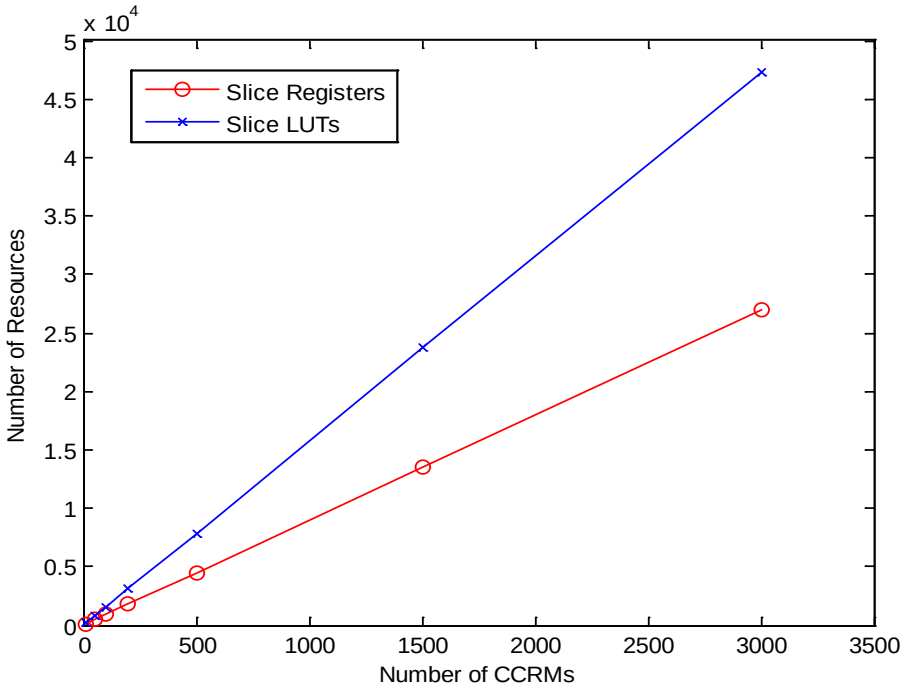

Figure 9. Resource utilization vs. number of CCRs

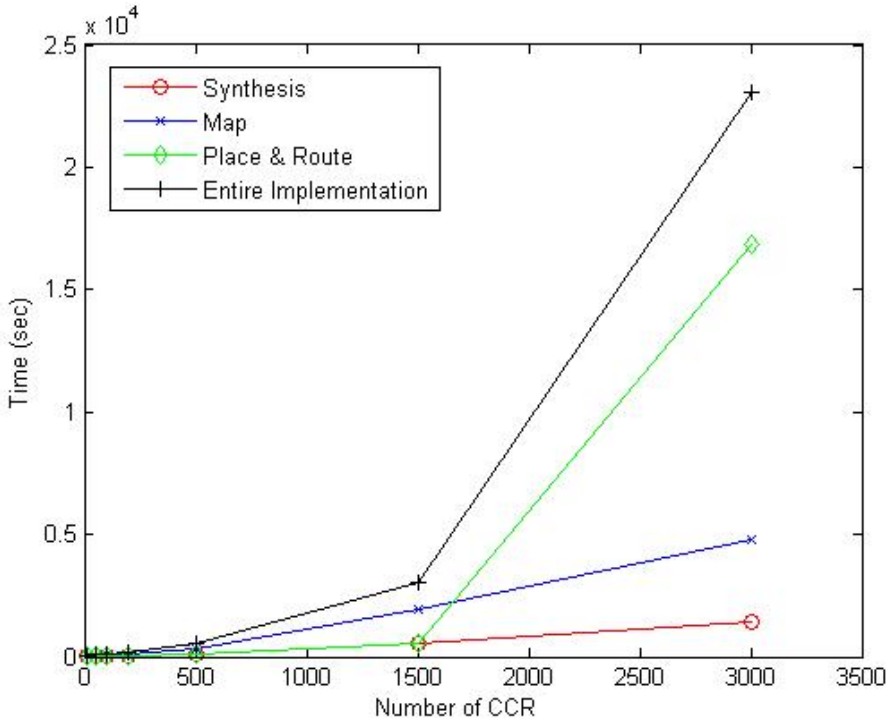

Figure 10. Implementation time vs. number of CCRs

From the curve we can see that it took more than 6 hours to implement the design. On the other hand, after the one time implementation of the base CES, subsequent regexp reconfiguration needs to transfer at most 5328Kbits (maximum BRAM size on LX110T), or 5.5 milliseconds through the Gigabit Ethernet, plus negligible packet dispatching time. In comparison, in conventional FPGA based NFA designs, any regexp rule updates will require hours of time to re-implement the entire scanner code.

2.4.2 Live Experiments

For live experiments, we connected a Virtex 5 evaluation board (XUPV5-LX110T) with a PC running Windows XP through a Gigabit Ethernet switch. The FPGA

was configured into a CES topology with rows of 25 linearly concatenated CCR engines. The software/hardware APIs provided by SIRC significantly reduced the coding effort for the data transfer between PC and FPGA. We set the data-transfer packet size to 512 Kbytes, which enables SIRC to achieve ~950 Mbps (95% of theoretical maximum of Ethernet communication) bandwidth [43]. In this configuration, it took only ~5.5 ms to update all 5328 Kbits BRAM of the Virtex 5 LX110T chip. The packet of data received by FPGA contains a flag indicating the purpose of this chunk of data, either for CES reconfiguration or string matching. Correspondingly, the CES controller either copies the packet data into the configuration registers of the CCR engines, or feeds the payload symbols as addresses to BRAM for acceptance lookup.

The unused CCR of each row of the CES topology is configured in bypass mode so that the matching result of each regexp rule appears at the final CCR of their corresponding row, which contributes one bit of the whole matching results vector. Matching results at each cycle are sequentially latched into the output buffer of SIRC, and are reported back to the host PC as a whole packet after processing of the entire string. This way, the PC side can easily trace back the matched regexp rule after receiving the matching results, as illustrated in Figure 11.

To better illustrate the procedure of string matching and results reporting, we have implemented R_2 together with seven noise regexps in CES and used "*abc-1-_3d*" as the input string, i.e., the same setting as in Table 2. The input string packet (from PC to FPGA) and the output match vector packet (from FPGA to PC) are captured by Wireshark [44], as shown in the top and bottom half of Figure 11, respectively. A 32-bit

header is used to associate matching result packets with the corresponding input string. At each clock cycle (after matching one input symbol), the matching outcomes of the eight regexp rules are assembled into a single byte, where 3rd least significant bit represents R_2. The PC side can easily conclude that R_2 reports a match at cycle 9 after matching symbol '*d*', the same result as shown in Table 2.

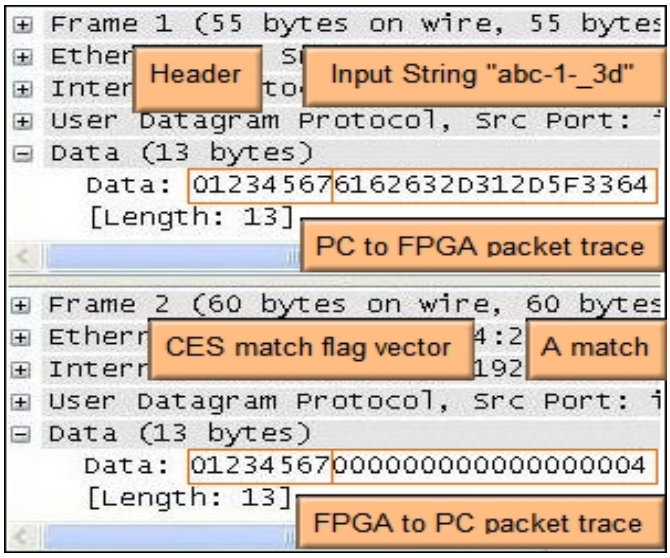

Figure 11. Input string and output match vector packet

2.4.3 Case Study: Snort

We analyzed the Snort regexp rule set to estimate the cost of implementation using CES. For the ease of implementation, we only tested the linear regexp rules. We partitioned the Snort linear regexp rules into two subgroups, regexp rules that are

composed of *simple* CCRs with single appearance ({1,1}) or wildcards ('*', '+' and '?'), and the others containing *complex* CCRs with complex constraint repetitions ({*m*,*n*},{*n*},{*n*,}). The former group is significantly easier to implement because each simple CCR has at most three states, i.e., matched with 0 times, 1 time, or multiple times, where the last case can be abstracted as a single state because '+' and '*' does not distinguish matches with different counts, as they can never exceed the upper bound (infinity). In practice, simple CCRs bounded by '*' or '+' can be implemented by simply adding a self loop transition. However, the same statement does not hold for complex CCRs with complex constraint repetitions, as they have to remember the matching counts to determine whether the lower and upper bounds have been reached or not. The number of regexp rules and CCR terms in each subgroup is reported in Table 7.

Table 7. Regexp and CCR statistics of linear Snort rules

# of / Regexp Type	Regexp Rules	Simple CCRs	Complex CCRs
Subgroup 1	891	33930	N/A
Subgroup 2	136	1528	152

Due to the different complexity of the two types of CCRs, we constructed a simpler implementation for simple CCRs, which only require 4 registers and 6 LUTs. The

LX110T device can accommodate 10000 – 15000 simple CCRs depending on the interconnection topology. As such, we estimate that the first subgroup of regexp rules can be implemented on five LX110T chips, taking into account the redundant space needed for future regexp rule updates. For the second subgroup of regexp rules, a single LX110T chip is sufficient. Generally speaking, it is very likely that CES may require more resources compared with conventional NFA techniques, because it has to pre-allocate ample resources for each building block of the system to maintain modularity, even if they may not be fully utilized in execution time. The gain of this method lies in flexibility and reconfiguration time, as traditional NFA designs have to re-synthesize and Place&Route the design even if there is a minor change in the regexp rules.

Next, we analyzed the distribution of complex CCRs among three types of complex constraint repetitions, i.e., $\{m,n\}$, $\{n\}$ and $\{n,\}$, as well as the number of NFA states needed to implement them in conventional NFA based designs. We used n for the estimates of NFA states required for $\{m,n\}$ and $\{n\}$, because in conventional NFA design, one state is needed for one possible matching count in the range of constraint repetition. However, for $\{n,\}$, only n states are needed based on an analysis similar to that of '*' and '+'. From the statistics listed in Table 8, it is clear to see that although the number of complex CCRs is two orders of magnitude smaller than that of simple CCRs (152 vs. 35458), the total number of NFA states needed to implement them are roughly similar: 28835 vs. 35458. This confirmed our assertion that constraint repetition is the most resource-consuming syntax to implement on conventional NFA architectures.

Table 8. CCR type and state distribution

# of CCR type	CCRs	NFA states (sum of *n*)
{*m*,*n*}	15	217
{*n*}	130	25829
{*n*,}	7	2789

In addition to being used as the building block of CES, the proposed CCR architecture and MIN-MAX algorithm can also be readily plugged into existing systems. For example, for the implementation of complex constraint repetitions in Snort, the CCR architecture can save 74% of memory bits as compared to the solution proposed in [15], where shift registers are used to implement {*m*,*n*} and {*n*}, and counters are used to implement {*n*,}. While the number of bits needed for {*n*,} is effectively reduced from 2789 to *log* (2789), the number of bits to implement the other two types remains the same, i.e., 25829 + 217 = 26046 bits. On the other hand, we need 152 CCR engines to implement those CCR terms, each of which consists of four 11-bit registers (because the maximum repetition observed in Snort is 1253), i.e., b^L , b^U , MIN and MAX, with total register of 4 * 11 * 152 = 6688 bits. That is 74% reduction in terms of register bits.

2.5 Summary

In this section we presented the counter based CCR model for complex regexp patterns, its corresponding MIN-MAX algorithm for exact matching, the CES architecture design, and experiments performed on a FPGA chip. MIN-MAX provides a cost-effective solution for matching of complex patterns that are of dynamic contents and lengths. We proved that the MIN-MAX algorithm can resolve the character class ambiguity problem by using a pair of counters, and support overlapped matching when regexps are inherently collision free or safe. We developed heuristic criteria and their proofs to determine the absence of collisions for a given CCR based regexp. Based on these criteria, we showed that the vast majority of regexp rules in two major network security tools Snort and SpamAssassin are immune from collision, so that they can be directly supported by MIN-MAX. Our case study on the Snort regexp rule set shows that the proposed (MIN,MAX) counter design can save 74% memory bits compared to conventional NFA based designs. The memory based architecture allows rapid update of regexp without re-synthesis of the entire design, provided that the new regexp can be embedded into the pre-loaded CES topology.

3. MELODY MATCHING SYSTEM

In this section we discuss the design of a melody matching system, which can be modeled as time series based pattern matching. Time series are digital data sampled from analog signals at certain frequency, which are common scenarios for many signal analysis systems with intensive interactions with outside real-world signals. A notable feature of time series pattern is that they have the timing information embedded into the data sequence itself, and when they are used as the pattern to detect similar sequence from the input data stream, quite often two matched parties may be slightly out-of-pace but still considered similar. Therefore, for the design of a melody matching system we need to consider the possible tempo variations, and also the match needs to be made approximately because analog-to-digital conversions are subject to noises. As for the choice of the scanner architecture, FPGA is selected to meet the performance requirements imposed by the large volume of music database. Moreover, the user query (modeled as time series) needs to be rapidly updated to ensure the best user experience, and therefore the reconfiguration time must be minimized.

The rest of this section is organized as follows. In subsection 3.1 we introduce some background information and related works in the domain of melody matching system. We then propose the system modeling for the MIDI music database and the user query in subsection 3.2 to convert the melody matching problem to approximate regexp matching problem. The elastic matching algorithm for time series patterns are described in subsection 3.3, and design of the FPGA-based music melody scanner is covered in subsection 3.4. Experiments and evaluations are conducted in subsection 3.5 and

subsection 3.6 summarizes the design of the melody matching system.

3.1 Background and Related Works

Nowadays, music is much more accessible in a digital file format than a physical copy. With the ever-growing disk space, people can easily store thousands of songs on their personal computers, and most internet media providers maintain a music database with millions of songs. The large scale of music database makes its efficient and effective retrieval more and more challenging. The traditional way to organize music files is to use auxiliary text metadata such as song title and artist name, so that well-developed string matching algorithms can be employed to efficiently retrieve the desired song given its exact metadata. However, such systems would provide poor user experience or even fail if the user forgets the exact metadata. Under such circumstances, support of *query by humming* [45] will be invaluable for a music information retrieval (MIR) system, where the music is retrieved not by its auxiliary metadata but by its acoustic content. This content-based music retrieval technique has become more and more appealing because melody is the natural and unique signature of a music piece, and retrieval by singing the melody is considered a lot more user-friendly than inputting its non-semantically-related textual metadata.

Many content-based MIR systems [46][47][48] have been developed to meet the emerging needs of query by humming. The traditional way of melody matching was to model both database music pieces and the user sung query as character sequences and then apply string matching algorithms. Some researchers have developed methods for modeling [49][50] and retrieval [51] of polyphonic music, but many more are focusing

on monophonic ones, because for polyphonic music it is very hard to clearly define the next character in the sequence, as there may be multiple channels of sounds simultaneously on. In the context of monophonic music retrieval, the most commonly used format to store database music files are MIDI [52], which represents music in its score level. A typical MIDI file is composed of sequential note events specified with a pitch, onset time, and duration. The user sung queries are also captured as monophonic WAV files, whose pitch sequence along the time axis can be acquired by fundamental frequency estimators, based on either time domain autocorrelation methods [53][54][55] or frequency domain cepstrum analysis approaches [56].

There are two major directions to discretize the database and user query into digital formats for their matching of each other, namely note-based and frame-based methods. The prior one attracts considerable research efforts because users naturally sing a music piece in notes, and notes are also the fundamental building blocks of MIDI files. Signatures can be extracted as strings [57], *n*-grams [58][59][60] or hidden Markov chains [61][62], and similarity between user query and database files can be calculated by approximate string matching algorithms [63][64] or in a probabilistic manner. Most systems of this category depend on accurate note segmentation [65] from the pitch sequence of a user query, which is sensitive to noises and therefore limits the system's retrieval accuracy. This leads many other researchers to focus on matching the user query and database files directly at the frame level, where the pitches of both sides are sampled into a time series and their edit distance [66] is calculated by approximate string matching algorithms. This way, the rhythm information is encoded in the time series,

and no explicit note segmentation is required. For this reason, frame-based systems can usually achieve better retrieval accuracy than note-based systems. However, their finer-grain time resolution results in larger database and user query, and therefore have a longer run time.

Several research efforts have been proposed to improve performance of frame-based systems, such as recursive alignment [67], score-level fusion of multiple classifiers [68], and hierarchical filtering [69]. A hybrid system is proposed in [70] where note-based methods are first used to filter out most unlikely database candidates, and the rest are compared to user input using frame-based methods. An evaluation method and testbed for content-based MIR systems is proposed in [71], and various types of retrieval techniques are evaluated and compared in [72].

No matter which direction a MIR system follows, the tolerance of key transpositions and tempo variations between the user sung query and its original melody is essential for its proper functioning, because few users can sing songs perfectly in their original keys and rhythms. Key transpositions can be taken care of by shifting the query to have the same average pitch as the database song [69], or by interval coding [59] where the differences between adjacent pitches, rather than their absolute values, are used for matching. To compensate rhythm variations, linear scaling [73] and dynamic time warping (DTW) [74][75] are commonly used to stretch/shrink the user query. Linear scaling tunes the user query faster or slower at certain preset ratios, while DTW allows finer grain continuous alignments within a tempo variation range.

3.2 System Modeling

The database music pieces are stored as MIDI files, and the user sung queries are captured by the microphone as WAV files. In order to evaluate their similarity, both parties are re-represented using the same underlying frame structure. Each frame is associated with a pitch value and all frames have the same time duration. This way, both database and user query are parsed into frame sequences, which are further modeled as strings and time series in subsections 3.2.1 and 3.2.2, respectively. The time series pattern is further modeled as CCR based regexps to account for tempo variations.

3.2.1 String Model for MIDI Files

MIDI (Musical Instrument Digital Interface) is an industrial specification of commands for electronic musical instruments and peripheral devices to interface with each other. A MIDI command can specify a note event such as pitch and onset time, a control signal such as pitch bend and audio panning, or a clock signal such as tempo. A MIDI file (file extension .midi) is a stream of MIDI commands coded in binary format which is playable by a music player to generate the acoustic effects perceivable by an end user. MIDI files are more like musical scores rather than recorded music performances such as MP3 files, which makes them very compact in size and easily distributable via the Internet.

The database we used in this study are all monophonic MIDI files, which only contain very basic tempo information and note events. The notes are non-overlapping with each other, meaning that a note can begin only after a previous note has ended. The database MIDI files are parsed by MIDICSV [76] to extract musical information into

comma separated values (CSV) text files. To illustrate this process, we show the famous "Happy Birthday to You" song in both professional sheet format and CSV format. We first show its musical score [77] in Figure 12 as follows.

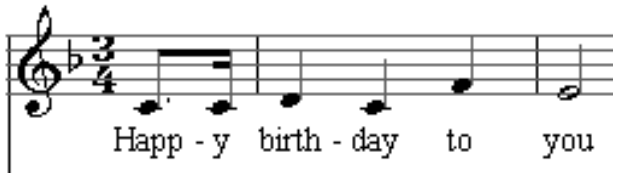

Figure 12. Musical score of "happy birthday to you"

The corresponding CSV version of the same song is listed in Table 9, where the left column is the timeline in clock pulses, the middle column contains the events, and the right column are the events' corresponding values. The first two rows are setup parameters specifying that a quarter note spans 480 clock pulses and lasts 600,000 microseconds, i.e., a clock pulse occurs every 1250 microseconds. With this information we can convert the timeline from clock pulses to seconds. In the rest of rows, a Note_on event signifies the onset of a note with its pitch specified in the value field, and a Note_off event denotes the end of a note.

Table 9. CSV version of "happy birthday to you"

Time	Event	Value
0	Header	480
0	Tempo	600000
960	Note_on	60
1320	Note_off	
1320	Note_on	60
1440	Note_off	
1440	Note_on	62
1920	Note_off	
1920	Note_on	60
2400	Note_off	
2400	Note_on	65
2880	Note_off	
2880	Note_on	64
3840	Note_off	

MIDI standards specify a pitch range from 0 to 127, i.e., the same range of ASCII values of characters. Therefore, when parsing the MIDI file into frame sequence, we can use characters to represent the frame's pitch values. As for the time duration of one frame, we choose the value of *l* milliseconds to balance the resultant database size and retrieval accuracy. As will become clear later, too short a frame length will result in a larger database, while too long a frame length will be unable to capture all essential

musical information. An appropriate value for *l* will be determined in the experiments of subsection 3.5.2.

With the above frame settings, the database MIDI file can be represented as a string, where each character lasts *l* milliseconds with the pitch being its ASCII value. For example, if *l* is set to 100 milliseconds, the "Happy Birthday to You" song can be converted into the following string

S_H: <<<<<<<>>>>>>><<<<<<AAAAAA@@@@@@@@@@@@ (1)

The silent parts of the MIDI files, if any, are removed because they do not carry meaningful music information. This way, all essential music information of the entire MIDI files database are converted into a set of database MIDI strings. To ease the later processing of key transpositions of user queries, the database MIDI strings are all shifted to have the average pitch of 60.

3.2.2 CCR Model for User Query

The user sung queries are recorded by microphone, which converts the continuous analog signal into a discrete digital format stored as WAV files (file extension .wav). The query corpus data used in this study are all monophonic files sampled at 8000 Hz. The musical sounds are comprised of periodic signals, which have different frequencies for different pitches. To extract pitch values from such signals, we need a pitch detector which estimates the fundamental frequency (f_0) of the signal at each point of the timeline to get its dominant pitch. As this part is more involved with signal processing and is beyond the scope of this dissertation, we directly adopted the autocorrelation-based YIN algorithm [54] as the pitch detector, whose MATLAB source

code is publically accessible at [78]. YIN generates f_0 estimation for each sample point of the input WAV file, which can then be converted to a pitch value using the following formula:

$$p = 69 + 12 \times log_2(\frac{f_0}{440\ Hz}) \quad (2)$$

where A_4 (440 Hz, the A note above middle C) is assigned with pitch number 69 and an octave is evenly divided into 12 semitones. We use the aperiodicity measurement generated by YIN algorithm to identify voiced sounds, and all silent parts are filtered out as we do to the database MIDI files. As the final output, we get an array of pitch values, with each array element lasting 125 microseconds, i.e., inverse of the sample rate of the WAV file. To match the frame settings of database MIDI strings, we re-write the pitch array into a frame sequence using the same frame length *l*, generating a time series of user query, denoted as S_u.

It is noteworthy that the users will seldom sing the query in the same key and tempo as its original melody. However, even with key transpositions and tempo variations, a user sung query may still be perceived by other users as similar to its original melody, if they share roughly the same "pitch contour". Therefore, a robust content-based MIR system needs to retrieve user sung queries in a transposition and tempo invariant manner.

Although a pitch sequence can be made transposition-invariant by interval coding [59], it has the disadvantage of assessing only a minor penalty for a transposition in the middle of melody, which however will usually be perceived as a major change. An alternative is to pitch-shift the user query string S_u to S_u^i, where S_u^i has an average pitch

of i so that it has the same average pitch as the melody to be matched. Recall that in subsection 3.2.1 we shift all database MIDI strings to have the average pitch of 60, and therefore by shifting S_u to S_u^{60} we can have a user query with the same average pitch 60 as all database MIDI files. However, as the user will typically sing only a portion of the melody, and there is a good chance that the local average of user sung portion differs from the global average of the complete melody, we compensate their possible difference by further shifting S_u^{60} up and down by up to k. This way, we generate a total of $2k+1$ copies $\{S_u^{60-k}, \ldots, S_u^{60+k}\}$ of user query strings. With the average-pitch variation threshold k being sufficiently large, it is safe to claim that one of the $2k+1$ variants of user query will align well with its matched portion of the original melody. However, too big a value k will result in an unnecessarily large search space which translates to a long processing time. An appropriate value of k will be tuned in the experiments of subsection 3.5.2 to cover adequate search space while not wasting processing power.

For analysis of tempo variations, we extract notes from the pitch array as the analysis object, because users naturally sing the query in a note-by-note basis and they will usually change the rhythm of all notes during the recording. Although it seems opposite to the claimed frame-based method, keep in mind that the concept of note is utilized here solely for the purpose of formal discussion, and in implementations the underlying structure is still frames.

To reconstruct notes from a pitch array, adjacent elements with the same pitch value are assembled together as one note, which is then represented as a character class with constraint repetition (CCR). Recall that here are three types of constraint

repetitions, namely $\{x\}$ (matching exactly x times), $\{x,y\}$ (matching no less than x times and no more than y times) and $\{x,\}$ (matching at least x times). In the context of this section, the CCR representation of a user sung note would have a character class of only one character, and a constraint repetition of type $\{x\}$. The entire user sung query of n notes can be modeled as a regexp composed of n CCR terms as follows.

$$R = p_1\{x_1\} \bullet p_2\{x_2\} \bullet \ldots \bullet p_n\{x_n\} \quad (3)$$

where p_i is the character corresponding to the pitch value of note i, $\{x_i\}$ is the constraint repetition of note i, and '•' is the concatenation operator. This way, the user query string S_u can be rewritten as the following CCR based regexp, if the user sings exactly the same as the sheet music of Figure 12 and $S_u = S_H$:

$$R_u : <\{7\} \bullet >\{6\} \bullet <\{6\} \bullet A\{6\} \bullet @\{12\} \quad (4)$$

With the user sung note modeled as CCR, its variation in tempo can be tolerated by expanding the constraint repetition from type $\{x\}$ to type $\{x, y\}$, so that instead of matching the database MIDI note of exactly the same duration, it can now match a note of any duration within the specified tolerable range. For a CCR note with constraint repetition $\{x\}$, empirical results suggest that expansion to $\{y, 4y\}$ would cover most tempo variations and lead to the best matching accuracy, where $y = \lceil 0.6x \rceil$ and "$\lceil\rceil$" denotes the ceiling function. This way, the original R_u can be relaxed in tempo as follows:

$$R_u' = <\{5,20\} \bullet >\{4,16\} \bullet <\{4,16\} \bullet A\{4,16\} \bullet @\{8,32\} \quad (5)$$

The CCR model of note is advantageous over linear scaling in the aspect that each note is treated individually so that local rhythm distortion can be tolerated and will not always lead to high global dissimilarity.

With key transposition and tempo variation considered together, a user query can be modeled as a set of CCR based regexps using the process shown in equation (6), so that both variations in key and tempo can be tolerated, leading to a robust MIR system.

$$\textit{User query} \xrightarrow{digitize} \mathrm{S}_u \xrightarrow{key\ transpositions} \{S_u^{60-k}, \ldots, S_u^{60+k}\} \xrightarrow{tempo\ variations} \{R_u^{60-k}, \ldots, R_u^{60+k}\} \quad (6)$$

3.3 Elastic Matching Algorithm

With the aforementioned string model of database music and CCR-regexp model of user sung query, we can design a regexp matching algorithm for melody matching, with consideration of the following two points.

Firstly, the user will usually only sing the portion of melody that he or she is most familiar with as the query, which can start anywhere in the original melody. Therefore, when matching against the query regexp, the database MIDI string needs not to be matched in its entirety, and the presence of a matching substring should be sufficient to assert the matching. This is referred to as *overlapped matching*, where each character of the input string needs to be treated not only as the "next symbol" for any on-going matching processes, but also as the "beginning symbol" of a substring for a new matching process. Secondly, the user query may not exactly match the database melody even after key transposition and tempo variation, because noises are always inevitable. Therefore we need an approximate matching algorithm to measure the degree of

dissimilarity introduced by noises and imperfect singing, rather than the traditional exact matching algorithms which only generate a yes or no matching signal.

Taking the above-mentioned points into account, in this section we introduce an approximate regexp matching algorithm named Elastic Matching Algorithm (EMA) in the overlapping matching context, which can evaluate the similarity between the user query regexp and the database MIDI string, using *edit distance* as the similarity measure. Its sequential version is first discussed in subsection 3.3.1, which is then parallelized in subsection 3.3.2 to fit the FPGA architecture for acceleration.

3.3.1 Sequential Version

The term *edit distance*, previously known as *Levenshtein distance* [66], has been widely applied in quantitative analysis of similarity. In overlapped matching context, the edit distance between a string and a regexp is defined as the least cost of edits needed to convert any of its substrings to exactly match the regexp. Unless otherwise specified, all "edit distance" terms used later in this section are referred in the overlapped matching context. In its traditional definition, edits include insertion of a new character, deletion of an existing character, or substitution of a non-matching character to a matching one, and each edit has its associated edit cost. However, in design of the elastic matching algorithm for a content-based MIR system, we only allow substitution edits and we exclude insertion and deletion edits. This is justified because the database MIDI strings and user query regexp only carry essential music melody information, which we claim should not be deleted or inserted to make a matching. The edit cost associated with substitution edits is defined as the ASCII value difference between the acceptable

character and the actual input character, and the match of the two is regarded as a substitution with no cost. This way, more penalties will be assessed as the user sung query deviates more from its original melody, and no cost will be incurred if the melody is sung perfectly.

We will elaborate the elastic matching algorithm in the rest of this section, and the following notations are introduced to facilitate the discussion. Consider a database MIDI string of m characters $S_m=\{c_1, \ldots, c_m\}$ where c_k is the k^{th} input character and it is processed at cycle k. Let S_k denote the substring $\{c_1, \ldots, c_k\}$ of S_m. The query regexp of n notes is rewritten into CCR format $R_n=CCR_1 \bullet \ldots \bullet CCR_n$, where each CCR_i is called a state which represents the progress that R_n has been matched up to, and it is in the format of $p_i\{y_i,4y_i\}$ as introduced in subsection 3.2.2. Recall that although CCR is a note representation, its underlying structures are still frames. For its frame-level analysis, we decompose state CCR_i into sub-states of $CCR_i^0 \bullet \ldots \bullet CCR_i^{4y_i}$, where each sub-state is a frame and CCR_i^j denotes the sub-state that p_i has been matched j times. The sub-states of CCR_i can be divided into three groups by the matching progress they represent. CCR_i^0 is the *initial* sub-state where the matching process just starts for CCR_i, $CCR_i^{y_i} \bullet \ldots \bullet CCR_i^{4y_i}$ are the *qualifying* sub-states where the matching process has satisfied the constraint repetition and is ready to move forward to CCR_{i+1}, and $CCR_i^1 \bullet \ldots \bullet CCR_i^{y_i-1}$ are the *non-qualifying* sub-states where the matching process is still making its way towards qualifying sub-states. Note that CCR_i^0 does not contain any music information and it only serves as an interaction method between adjacent CCRs. Let R_i^j denote the sub-regexp

that starts with CCR_1^0 and ends at CCR_i^j. Each CCR_i^j carries two counters $ed_{i,j,k-1}$ and $ed_{i,j,k}$ to store its edit distance values of previous cycle *k-1* and current cycle *k*, respectively, where $ed_{i,j,k}$ denotes the edit distance between S_k and R_i^j.

With the above sub-state model of query regexp and its corresponding notations, we can draw the state transition diagram between CCR_1 and CCR_2 in Figure 13. Although only the first two CCRs of R_n are shown, the same transition diagram is applicable to all other adjacent CCRs.

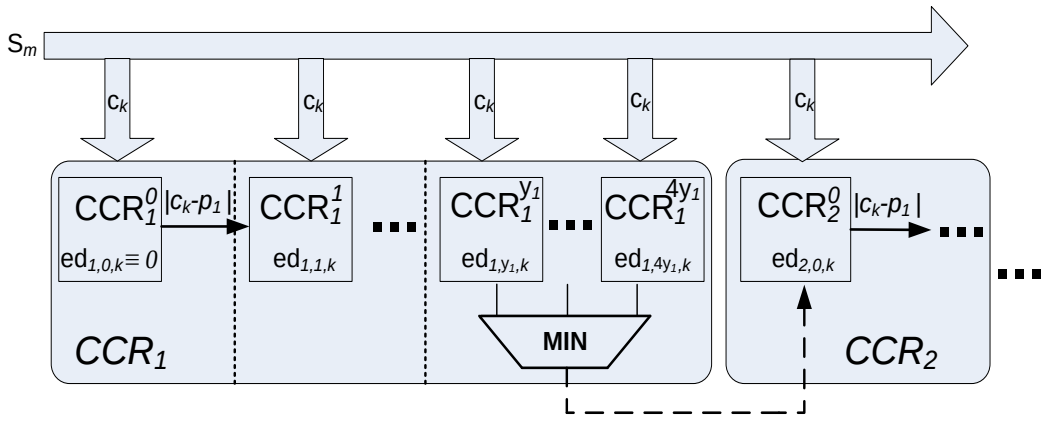

Figure 13. State transition diagram

Sub-state CCR_1^0 is the initial sub-state of the whole regexp, and its edit distance counter is set to be constant zero to support overlapped matching, i.e., the following equation:

$ed_{1,0,k} = 0$ for all *k*. (7)

This way, CCR_1^0 serves as a brand new starting point for any substring of S_m starting at any input character.

Each input character c_k is broadcasted to all sub-states at cycle k, triggering an intra-state substitution edit transition and incurring an edit cost, which is denoted by the labeled solid arrow. The edit distance counters of sub-states are updated accordingly by the following equation:

$$ed_{i,j,k} = ed_{i,j-1,k-1} + |c_k - p_i| \text{ , } 1 \leq j \leq 4y_i \quad (8)$$

The optimality of equation (8) can be proven by a simple induction analysis. The *MIN* function then selects the minimum one from all qualifying sub-states and propagates it to the initial sub-state of its successor CCR_{i+1} via an inter-state transition (denoted as dotted arrow), i.e., the following equation:

$$ed_{i+1,0,k} = min\ (ed_{i,y_i,k}\ , \ldots, ed_{i,4y_i,k}),\ 1 \leq i \leq n\text{-}1 \quad (9)$$

This way, CCR_{i+1} will base its processing on the best result of the past, and therefore the basis of the induction analysis for CCR_{i+1} holds. Combined together, equation (8) and (9) guarantee the optimality of all calculated edit distance values, and hence the correctness of the proposed algorithm shown in Figure 14. To improve the readability of the pseudo-code, we introduce the following two variables:

$$curr\_min_{i,k} = min\ (ed_{i,y_i,k}\ , \ldots, ed_{i,4y_i,k}), \text{ and} \quad (10)$$

$$overall\_min_{i,k} = min(curr\_min_{i,1}, \ldots, curr\_min_{i,k}). \quad (11)$$

The semantic meaning of these two variables can be better illustrated in the case of $i = n$, i.e., the final CCR of R_n. At cycle k, different substrings ending with c_k may make different matching progress towards the end of R_n, and those that successfully

propagated to qualifying sub-states of CCR_n can claim an approximate match with their associated edit costs, i.e., $ed_{n,y_n,k}$, …, or $ed_{n,4y_n,k}$. Among them, the minimum one is selected as $curr\_min_{n,k}$ and the minimum of $curr\_min_{n,k}$ over all k cycles are selected as $overall\_min_{n,k}$, which by definition is the edit distance between R_n and S_k in overlapped matching mode, because it is the minimum chosen from all possible substrings of S_k. The semantic meaning of the two variables for cases of $i < n$ can then be understood in a similar way, by regarding CCR_i as the final CCR of regexp R_i. Finally, $overall\_min_{n,m}$ is the edit distance between user query regexp R_n and database MIDI string S_m.

---

(1) $ed_{1,0,0} = 0$; //By equation (7)

(2) All other $ed_{i,j,0} = SYS\_MAX$; // The maximum value

(3) //allowed in the system

(4) $overall\_ed_{i,0} = SYS\_MAX$ for $1 \le i \le n$;

(5) foreach (c_k of S_m) {

(6) $ed_{1,0,k} = 0$; //By equation (7)

(7) for ($i = 1$; $i \le n$; i++) { //Iterate through CCRs

(8) for($j = 1$;$j \le 4y_i$;j++) { //Iterate through sub-states

(9) $ed_{i,j,k} = ed_{i,j-1,k-1} + |c_k - p_i|$; // By equation (8)

(10) }

(11) $curr\_min_{i,k} = min\ (ed_{i,y_i,k}$, …, $ed_{i,4y_i,k})$;

(12) $overall\_min_{i,k} = min(overall\_min_{i,k-1}, curr\_min_{i,k})$;

Figure 14. Pseudo code of elastic matching algorithm

(13) for($j = 0; j \leq 4y_i ; j$++) { //Store current cycle ed

(14) $ed_{i,j,k-1} = ed_{i,j,k}$; // to previous cycle ed, for

(15) } //use of next cycle

(16) if ($i < n$)

(17) $ed_{i+1,0,k} = curr\_min_{i,k}$; //By equation (9)

(18) }

(19) }

(20) Report $overall\_min_{n,m}$ as the edit distance

Figure 14. Continued

The elastic matching algorithm can find the best alignment between the user query and the MIDI string to minimize their edit distance, in the overlapped matching mode. For example, Figure 15 shows the fundamental frequency estimated by YIN for a user query of "My heart will go on", and the fastest and slowest allowed tempo for this user query is shown in Figure 16.(c) and Figure 16.(d), respectively. When matched against the MIDI string shown in Figure 16.(a), the elastic matching algorithm is able to find a best match starting from the middle, i.e., the matched portion shown in Figure 16.(b), and the minimum edit distance is achieved by following the tempo-adaptive alignment shown in Figure 16.(e). In other words, the elastic matching algorithm is able to find the balancing point between the fastest and slowest allowed tempos.

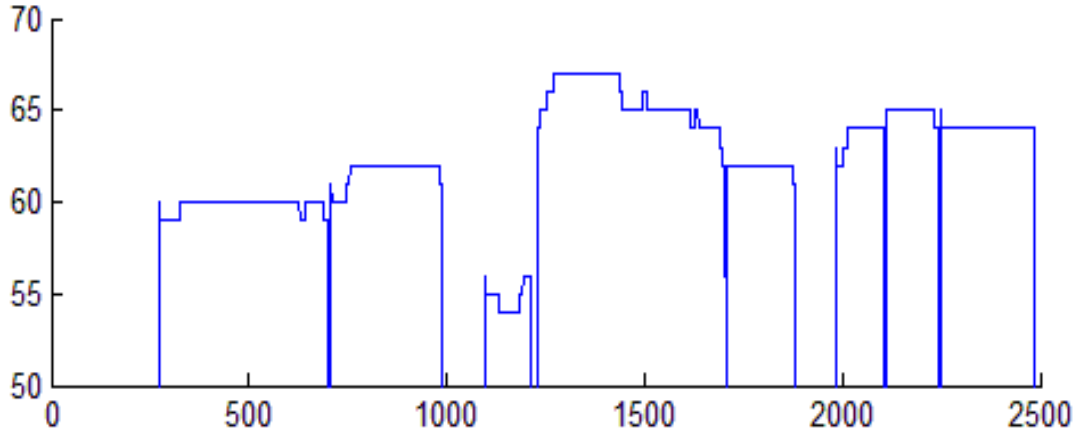

Figure 15. User query of "my heart will go on"

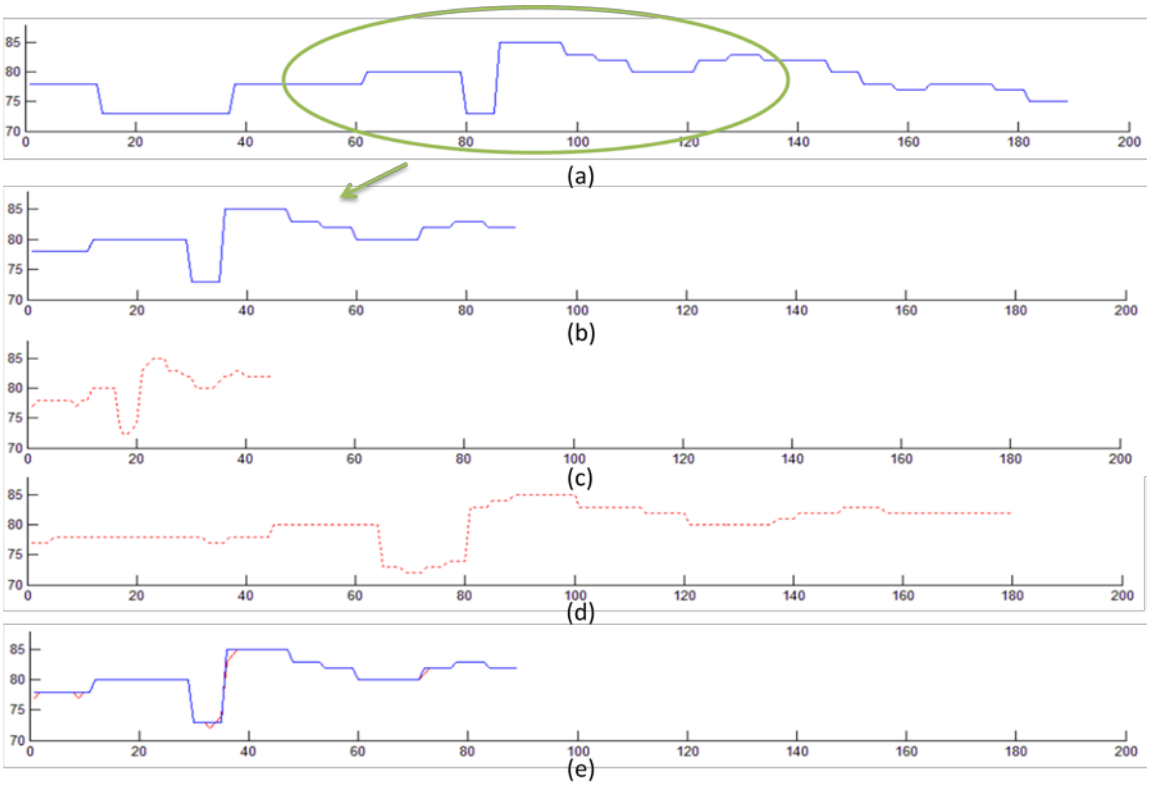

Figure 16. (a) Original MIDI string (b) Matched portion of MIDI string (c) Fastest allowed tempo of user query (d) Slowest allowed tempo of user query (e) Tempo-adaptive alignment between MIDI string and user query

3.3.2 Parallelization

Note from line (9) of Figure 14 that the edit distance value of a sub-state at cycle k is solely dependent on its direct predecessor sub-state's edit distance values at cycle $k-1$, and the input character c_k. Because c_k is broadcasted to all sub-states, they can update their edit distance values concurrently, a favorable manner for FPGA implementation using Verilog non-blocking assignments. However, several other issues have to be taken into account before the algorithm is implemented in FPGA for hardware acceleration. Firstly, different CCRs may have different constraint repetitions, and therefore the *MIN* function at line (11) may have different number of inputs. This leads to different execution path and time for CCRs, which makes their synchronization very difficult. Secondly, when the FPGA needs to implement a new user query, its regexp structures need to be reflected in the hardware description language (HDL) code as new logic functions, due to the changes in the number of sub-states and the way to loop through them. This translates to rerun of the synthesis, map, place and route process of the entire system for each new user query, which usually takes hours of time to finish. As such, the wait time for the end user is intolerable.

We can solve both problems by restructuring the CCR based regexp into a modular format, where all CCRs have the same constraint repetition. This way, all CCRs will have the same number of sub-states, and they can share the same MIN and loop functions. Recall that CCR_i is in the format of $p_i\{y_i,4y_i\}$, where the upper bound of the constraint repetition is always four times the lower bound. Therefore, we propose to use $p_i\{1,4\}$ as an atomic and modular building block. This way, we can break the original

"molecule" CCR into a sequence of "atom" CCRs denoted as ACCR. All ACCRs share the same architecture and logic functions, and they are only different in the acceptable characters, which however can be stored in registers.

This leads to a modular, parameterized design where the parameters (acceptable characters) of ACCRs can be rapidly updated to implement a new regexp, without modifying their underlying logic functions. The time-consuming reconfiguration process can therefore be avoided by pre-implementing a system of ACCR modules at a one-time cost, a tremendous value for time-sensitive scenarios such as the MIR system.

3.4 Melody Matching Engine Design

In this section we will describe the co-design details of a software/hardware system consisting of a frontend PC and a backend FPGA. The PC generates the database MIDI strings and converts the user sung query to an ACCR-based regexp, as described in previous sections. The FPGA implements the *melody matching engine* (*MME*), a modular architecture composed of pre-implemented ACCRs. The database MIDI strings are transferred to the FPGA's onboard DDR2 memory, and the query regexp can be implemented by MME through fast parameter updates. This way, the most computation-intensive works, i.e., calculations of edit distances between regexp and strings, are offloaded to FPGA for hardware acceleration. The calculated results are returned back to PC, and the 10 MIDI files with top-10 smallest edit distances are reported to the user as the retrieved songs. Details of the FPGA implementation and its integration with PC end are discussed in subsection 3.4.1 and 3.4.2, respectively.

3.4.1 FPGA End

A major design goal when designing MME is the reconfiguration time to implement a new user query, because the response time for an end user includes both regexp-updating and regexp-matching time. This is different from traditional FPGA designs where the objective is usually timing performance. For this matter, we developed a modular architecture for MME to support fast reconfiguration, where an ACCR engine is designed in subsection 3.4.1.1 as the general implementation of an arbitrary ACCR term, and an array of ACCR engines is pre-implemented in subsection 3.4.1.2 to accommodate an arbitrary ACCR-based regexp. To eliminate the confusion when discussing the relationship between the algorithm and its implementation, we use *ACCR term* to denote its regexp model, and *ACCR engine* to denote its hardware implementation.

3.4.1.1 ACCR

An $ACCR_i$ engine implements the behavior of an $ACCR_i$ term, which is in the form of $p_i\{1,4\}$. Its internal architecture is illustrated in Figure 17, which contains five edit distance registers to store $ed_{i,0,k}$, … , and $ed_{i,4,k}$, respectively, a $curr\_min_i$ register to store the minimum of edit distances at the current cycle, and a $overall\_min_i$ register to store the minimum of $curr\_min_i$ over all cycles. These registers store runtime variables which will be dynamically updated as the matching proceeds. In addition, $ACCR_i$ engine also contains static parameter registers p_i to store its acceptable character, i to store its index in the query regexp R_n, and n to store the number of ACCR terms in R_n. The latter two parameters together determine if $ACCR_i$ engine is implementing the final $ACCR_n$

term, and accordingly it should either report its $curr\_min_i$ to its successor for future processing, or report its $overall\_min_i$ as the final result. Beyond these registers, $ACCR_i$ also contains functional units, such as a subtractor to calculate the substitution edit cost, four adders to implement equation (8), and a comparator tree to implement the *MIN* function of equation (9). The MIN function is implemented in combinational logic so that the $curr\_min_i$ can be calculated at the same cycle after ($ed_{i,1,k}$, … , $ed_{i,4,k}$) are updated. It will then be propagated to $ed_{i+1,0,k}$ of its successor $ACCR_{i+1}$ for further processing from the next cycle on. This way, $ACCR_i$ will update its edit distance registers, finish the MIN function, and propagates $curr\_min_i$ to its successor in a single cycle time, so that at the next cycle all ACCR engines can process the input character in parallel.

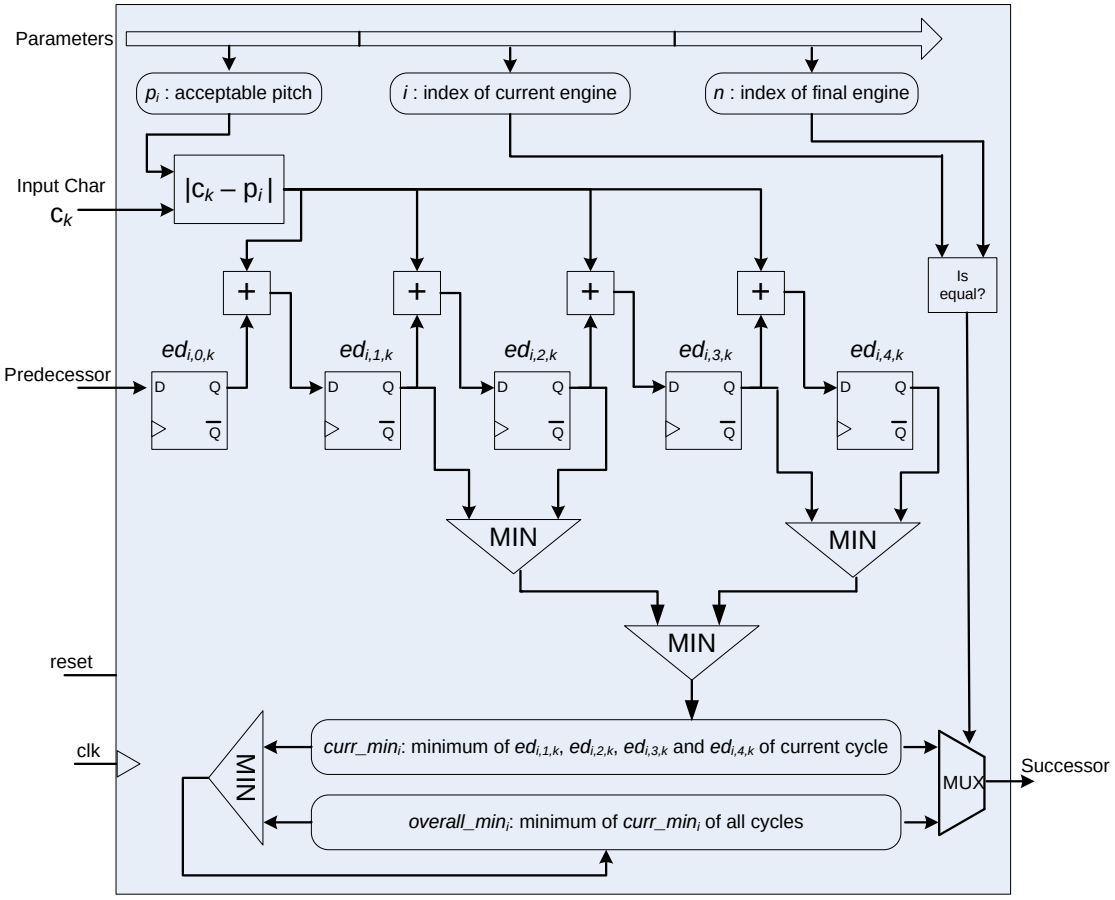

Figure 17. Architecture of $ACCR_i$ engine

Note that the edit distance counters $ed_{i,0,k-1}$, … , $ed_{i,3,k-1}$ (which store edit distance values of last cycle) are essential for line (9) of Figure 14, because otherwise an edit distance value will be updated by its predecessor sub-state before its old value is used to update its successor sub-state, i.e., a race condition. However, they can be safely removed in the $ACCR_i$ engine, because edit distance values are stored in flip-flops and can be updated by non-blocking assignments in parallel, without race conditions.

The $ACCR_i$ engine reads its parameter configurations from a register file, implemented in Block RAM (BRAM), using *i* as the address. All ACCR engines will be reset after receiving an input character of '\n', the delimiter of database MIDI strings, so that MME can start matching a new melody.

3.4.1.2 Melody Matching Engine

With its general and modular design, we can pre-implement an array of *t* ACCR engines to build a modular melody matching engine (MME) as shown in Figure 18. An appropriate value for MME size *t* will be determined later in subsection 3.5.2.

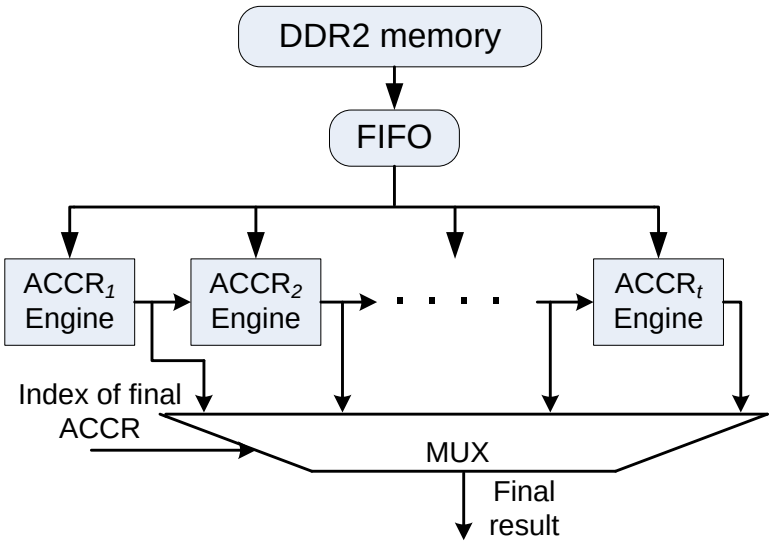

Figure 18. Architecture of MME

The database MIDI strings are stored in high-density on-board DDR2 memory for fast access, and each character is broadcasted to all ACCR engines for parallel processing. The database MIDI strings are read in sequence, in which mode the DDR2 memory can achieve peak throughput as this access pattern does not have command conflicts or bank conflicts. The DDR2 memory controller works at a clock frequency different from that of the ACCR engines, and we used a FIFO to address the meta-stability issue [79] caused by signal transitions across different clock domains.

The MME architecture can accommodate query regexps of different lengths, because all ACCR engines share the same architecture and they are all capable of implementing the final $ACCR_n$ term of R_n. A multiplexer is employed to locate the final $ACCR_n$ engine and take the appropriate $overall\_min_n$ value as final result, using n as the selection signal.

A problem naturally arises for regexps of more than t ACCR terms, which exceeds the capacity of MME. In such circumstances, we will sort the CCR terms of the regexp in ascending order by their constraint repetition, before we break the CCR-based regexp into ACCR sequence. CCR terms with smaller constraint repetitions (which represent user sung notes with shorter durations) will be trimmed off until the resultant regexp contains no more than t ACCRs. This is justified by the fact that short notes of the user query are more likely to be noises or irregular sounds, and therefore carry less essential musical information than longer notes do.

The proposed MME is a modular architecture, where all ACCR engines work concurrently in parallel to provide substantial speedup compared to software

implementations, while at the same time they can be easily and rapidly reconfigured by reading new parameters from on-chip Block RAM. This way, MME can achieve both hardware-level speed and software-level flexibility.

3.4.2 Integration with PC End

The PC side is responsible for all frontend works, including user interface, database MIDI string generation, and user query parsing. The flow diagram of the integrated software/hardware system is shown in Figure 19, where the right hand side bounded by dotted box is implemented in FPGA, while all other steps are running on the PC. We adopted SIRC (Simple Interface for Reconfigurable Computing) [43] as the communication framework between PC and FPGA. SIRC is developed by Microsoft as an attempt to ease the usage of the raw Ethernet functionality provided by the FPGA board, and it frees researchers from the burdens of building low level software drivers and hardware interface logics, so that they can focus more on the high level system functions. On the hardware side, it utilizes on-chip BRAM to implement three fast buffers, namely the input buffer to store the input data transferred from PC, the parameter buffer to store MME's parameter configurations, and the output buffer to store the calculated results. SIRC provides a set of hardware application programming interfaces (API) to read data from and write data to these buffers. MME starts processing after receiving the start signal from PC, and it triggers a done signal after finishing processing. On the software side, SIRC also provides a set of APIs to send data to input buffer and parameter buffers, retrieve data from output buffer, send start command, and detect done signal.

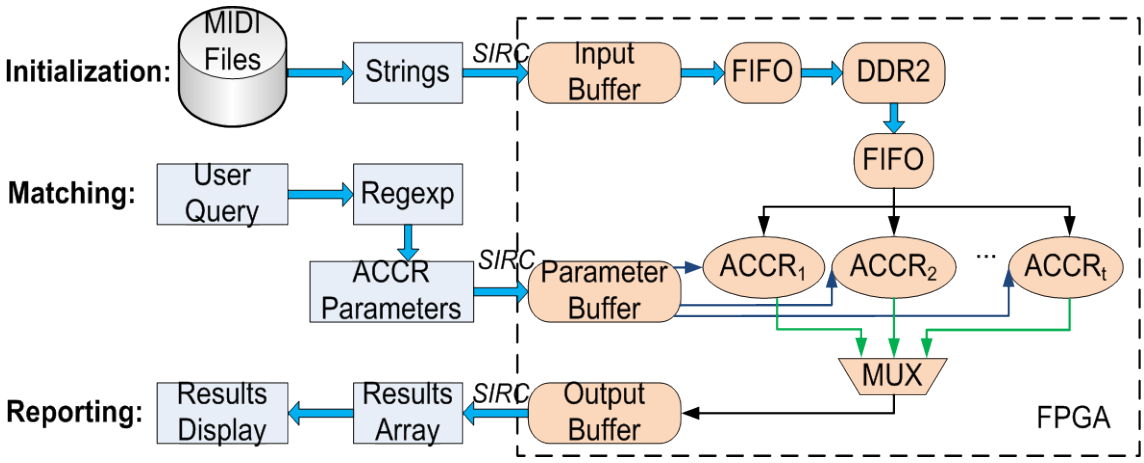

Figure 19. System flow chart

In the initialization phase, the database MIDI files are processed as discussed in subsection 3.2.1, and the resultant strings are downloaded into the input buffer in multiple batches, because the input buffer is not big enough to accommodate the entire database. All batches are further transferred to DDR2 memory and reassembled back there as a whole. SIRC hardware API and DDR2 memory controller work at different clock frequencies and the resultant meta-stability issue is addressed by using a FIFO.

In the matching phase, the user sung query is parsed into a regexp of n ACCR terms as discussed in subsection 3.2.2 and 3.3.2, and the parameters of each $ACCR_i$ term including its acceptable pitch p_i, index i, and the index of the final ACCR term n are encapsulated into a packet and sent over to the parameter buffer. MME will read parameters to update each ACCR engine, start processing the database MIDI strings, and finally write edit distance results into the output buffer.

In the reporting phase, edit distance results stored in the output buffer are retrieved by PC, and the MIDI songs with top-10 minimum edit distances are reported to the user as the retrieval results, i.e., they are the 10 most possible songs that the user has sung.

Through the high speed Gigabit Ethernet, MME can update its database music rapidly, and a new user query (typically several hundred bytes) can be downloaded and implemented in almost real time. By storing the database locally, MME forms a self-contained system and its communication with PC is minimized. Therefore MME is a highly-scalable architecture, and many FPGAs can be easily clustered together to serve multiple user queries simultaneously, with virtually no overhead.

3.5 Experiments and Evaluation

We conducted experiments in this section to evaluate the performance of the proposed system in terms of both accuracy and speed. The PC end software runs on an Intel Xeon quad core CPU working at 2.8 GHz with 16GB of DDR3 memory, and the FPGA end is implemented on the popular XUPV5 evaluation board with Virtex 5 LX110T (69120 registers and LUTs) FPGA chip and 256 MB DDR2 memory. Synthesis results will be discussed in subsection 3.5.1 to show the FPGA device utilization of MME. We then tune the parameters in subsection 3.5.2 to determine the appropriate values for frame length l, average-pitch variation threshold k, and MME size t. Finally, accuracy and speed performance of the proposed system are given and compared with related works in subsection 3.5.3.

3.5.1 Synthesis Results

The Verilog codes of ACCR engine and MME are synthesized, mapped, placed and routed by Xilinx ISE design suite 10.1, and the reports show that each ACCR engine utilizes 57 slice registers and 159 slice LUTs and can run at 80 MHz. With the SIRC hardware interface and DDR2 memory controller altogether utilizing around 4% of the chip area, the Virtex 5 LX110T device can implement a total of 300 ACCR engines at an occupancy ratio of 87%. The relatively slow operating frequency of 80 MHz is a result of multiple factors, including, but not limited to, the long combinational logic path of the MIN function in each ACCR, the large MUX of MME with a hundred 16-bit wide inputs, and the long data path from the FIFO to ACCR and ACCR to MUX. For example, the interconnections and routings of the MUX, the input character from FIFO to all ACCRs, and the output from all ACCRs to their successors and the MUX are shown in Figure 20, Figure 21 and Figure 22, respectively.

To improve performance, we introduced several buffers between the broadcasting input and CCR engines which act as repeaters to help reduce the global routing from broadcasting input to CCR engines to local routing from repeater to CCR engines. We also added multiple buffer stages for the MUX of MME to help timing without affecting the execution pipeline, because MUX is only used in the result reporting phase which needs not to finish in a single cycle. Without negative impact on accuracy, cutting the data width from 16 bits to 8 bits also significantly reduces the number of signals that need to be routed. After these performance optimization techniques, the operating frequency of MME is improved from 80 MHz to 100 MHz.

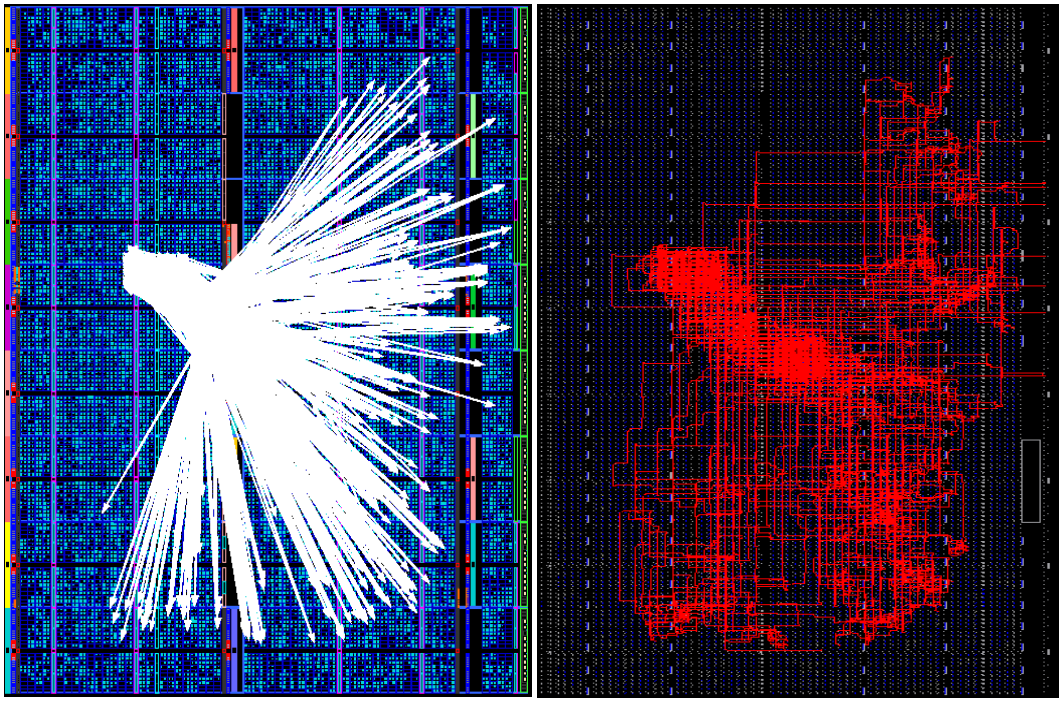

Figure 20. (left) Interconnections of the MUX (right) Routing of the MUX

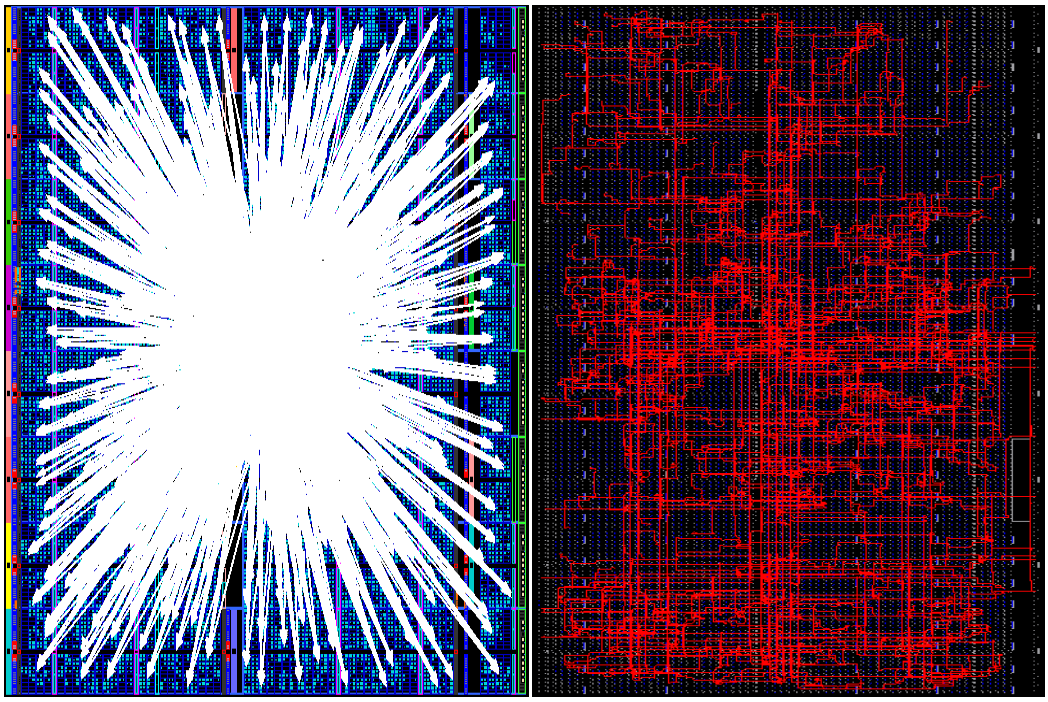

Figure 21. (left) Interconnections for input character from FIFO to ACCRs (right) Routing for input character from FIFO to ACCRs

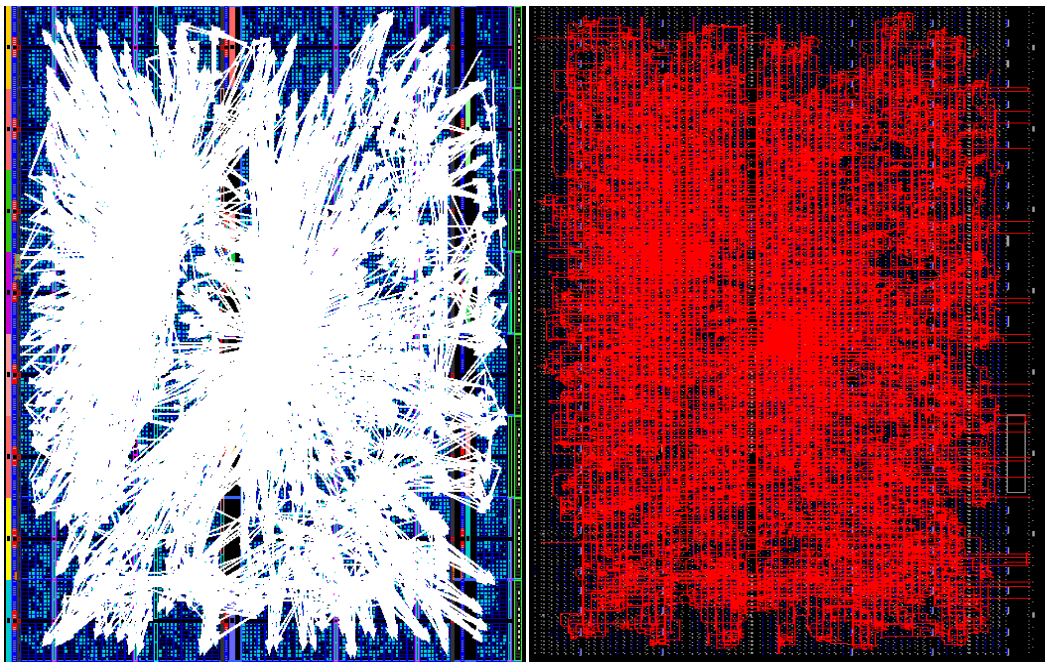

Figure 22. (left) Interconnections from ACCRs to their successors and the MUX (right) Routing from ACCRs to their successors and the MUX

The time spent on each step of the implementation process is reported in Table 10, from which we can see that the total implementation time of the FPGA end takes around 9 hours to finish. Although newer versions of the ISE software are able to utilize multiple cores to do the mapping, placing and routing works, it still requires hours of time to finish. The implementation time can become even longer if the design approaches the chip's capacity, or if the developer wants to push the performance to its limit.

Table 10. Implementation time of MME

Implementation Step	Run Time
Synthesis	19 minutes
Map	2 hours and 15 minutes
Place & Route	6 hours and 19 minutes

The time-consuming implementation process is inevitable for reconfiguration of traditional FPGA designs, where the HDL source codes need to be modified to reflect the design changes. However, this re-compilation of source code can be circumvented for MME due to its parameterized modular design, and it can be reconfigured to implement a new regexp by simply updating its parameter registers. This is a tremendous value for application scenarios where the functions to be implemented on FPGA regularly change, a category that content-based MIR systems fall into exactly.

3.5.2 Parameter Tuning

In this subsection we tune the previously defined variables, i.e., frame length l, average-pitch variation threshold k, and MME size t, to achieve the best tradeoff between retrieval accuracy, speed, and resource utilization. The retrieval accuracy is tested using ThinkIT's corpus [80] which includes 355 WAV files as user queries and 106 ground truth MIDI files. We further include 5463 MIDI files from the Essen [81] collection as noises, generating a database of 5569 MIDI files. The user sung queries may start anywhere in

their original melodies, not necessarily from the beginning. We used the top-10 hit rate as the accuracy metric, where the user query is said to be successfully retrieved if its original melody is in the reported top-10 most similar MIDI files, and the top-10 hit rate is counted as the percentage of successfully retrieved queries over all 355 user queries.

We first tune the average-pitch variation threshold *k*, with the frame length *l* set to 20 milliseconds (sufficiently small to capture most music information of the user query) and the MME size *t* set to include all 300 ACCR engines (sufficiently large to accommodate most user query regexps). The retrieval accuracy increases along with *k* as shown in Figure 23, due to the increasing regexp search space. However, the accuracy improvement becomes negligible when *k* is increased beyond 4, implying that the search space is saturated. Therefore, we determine the value of *k* to be 4, so that the search space is properly bounded while still guaranteeing high retrieval accuracy.

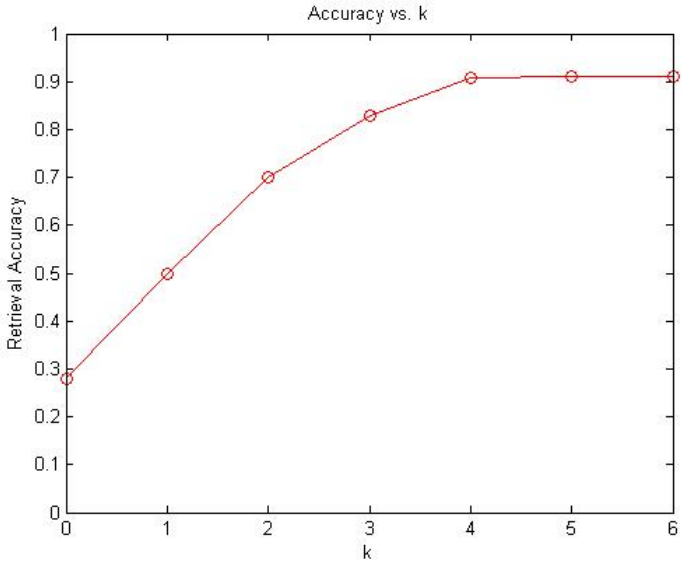

Figure 23. Average-pitch variation threshold vs. accuracy

We then conducted experiments to study how frame length parameter *l* affects the retrieval accuracy, and the results are shown in Figure 24. Generally speaking, a smaller frame length is more favorable than a bigger one, because the prior one is capable of capturing more musical details while the latter one may miss some essential musical information, especially for fast-rhythm melodies with many short notes. The downside of short frame length is the bigger resultant database size, and hence longer retrieval time. We can observe from Figure 24 that retrieval accuracy will not be further improved for frame lengths under 100 milliseconds, because they are beyond the shortest note distinguishable by the end user. Therefore, we chose 100 milliseconds as the frame length, so that we can minimize the database size to improve retrieval speed while at the same time retaining high retrieval accuracy.

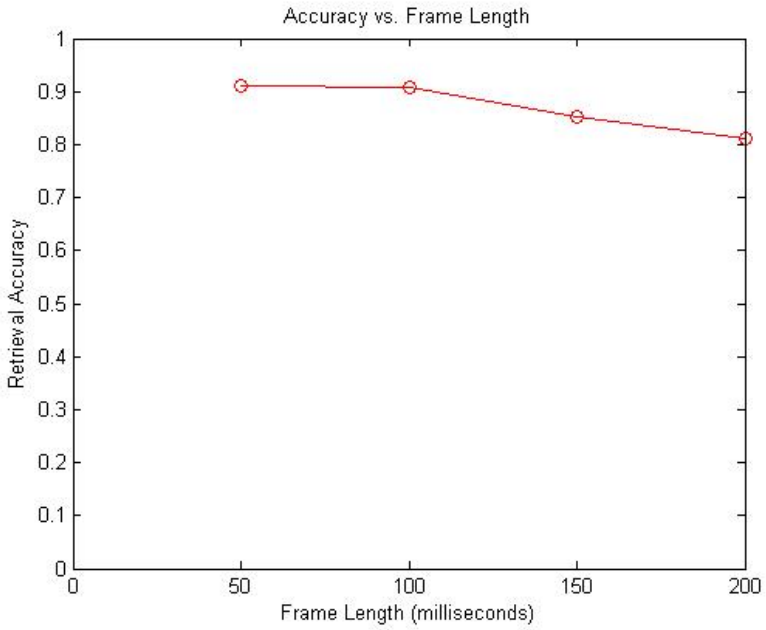

Figure 24. Frame length vs. accuracy

Finally, in most cases the user singing query is around 10 seconds including the silent parts. With the frame length set to 100 milliseconds, the resultant query regexp will contain around 100 ACCR terms, and therefore we set the MME size to 100. When the user query is too long to fit MME, it will be trimmed using the strategy discussed in subsection 3.4.1.2. This way, the 300 ACCR engines implemented on FPGA can be grouped into 3 MMEs running in parallel, each of which can implement one query regexp variant. With average-pitch variation threshold set to 4, all the 9 variants of the user query can be processed in 3 rounds.

3.5.3 Performance Comparison

We then tested the system performance using the parameters determined in the previous subsection. To summarize, the database of 5569 MIDI files are converted into strings stored in a plaintext file of 1.62 Megabytes, and each user query is parsed and average-pitch shifted into 9 regexp variants, each of which contains no more than 100 ACCR terms. On the FPGA side, 3 MMEs run in parallel, and it takes 3 rounds to finish processing of all 9 variants of the user query. Through the high speed Gigabit Ethernet, the database can be downloaded onto FPGA in less than 15 milliseconds, and the user query regexp parameters (several hundred bytes) can be transferred to the parameter buffer in almost real time. We then feed the query regexps of all 355 user sung WAV files into the MMEs for retrieval, and the total runtime for all queries as well as the average runtime for each query are listed in Table 11.

Table 11. Melody retrieval time

	Total	Average
runtime	19.4 seconds	54.6 milliseconds

Theoretically, for each user query the system needs to process a total of 1.62 × 3 = 4.86 Megabytes worth of data, which can be done in 48.6 milliseconds as the MME works at 100 MHz. This is very close to the actual performance shown in Table 11, which additionally includes the parameters downloading and results reporting phases.

We then compare the performance of the proposed system with related works selected from Music Information Retrieval Evaluation eXchange (MIREX) [82]. MIREX is a platform for researchers to present their state-of-the-art music retrieval related algorithms, and also a framework for formal evaluation of such algorithms. We selected several competitive algorithms from past sessions and compare their retrieval accuracy (top-10 hit rate) and runtime with MME in Table 12. Speedups of MME compared to these algorithms are also listed in the rightmost column.

Table 12. Performance comparison

Method	Platform	Runtime	Accuracy	Speedup
CSJ2 [83]	Dual Intel Xeon Quad Core @ 2.0 GHz with 24 GB memory	210 minutes	86%	649X
HAFR1 [84]	Dual AMD Opteron Quad Core @ 2.0 GHz with 32 GB memory	247 minutes	80.6%	764X
YF2 [84]	Intel Core 2 Quad Core @ 2.40GHz with 8GB memory	367 minutes	90.4%	1135X
MME	XUPV5 FPGA @ 100MHz with 256 MB memory	19.4 seconds	90.7%	1X

The most accurate solution reported in MIREX is proposed by Yeh and Fang [84] using a partial linear scaling method. It achieves retrieval accuracy of 90.4%, similar to that of MME but 1135X times slower. As can be seen from Table 12, all software based systems need tens of seconds for the retrieval of one user query, which is an unacceptable response time for real world MIR systems. In contrast, the proposed FPGA-based MME can return the retrieval result in just 50 milliseconds.

3.6 Summary

In this section we presented the CCR based regexp model for a melody matching system, its corresponding elastic matching algorithm for approximate matching, the MME architecture design, and experiments performed on a FPGA chip. The database MIDI files are modeled into strings and the user sung query is modeled as regexp composed of ACCR terms. We designed an approximate matching algorithm to calculate the edit distance between an ACCR-based regexp and an input string, where the ACCR terms can run in parallel. The algorithm is implemented in FPGA as a melody matching engine (MME) for hardware acceleration. MME is a modular architecture composed of ACCR engines, each of which implements the functionality of an ACCR term. This way, MME can implement a new user query by simple updating the ACCR engines' parameters, avoiding the time-consuming re-implementation of the source code. The relatively self-contained structure of MME makes it highly-scalable, and multiple FPGA can be easily clustered together to achieve more parallel processing power. Our experiments show that the proposed modeling can accurately capture the musical essence of both database melodies and user queries, and achieves a top-10 hit rate of 90.7%. Moreover, MME can process a user query in just 54.6 milliseconds, a significant speedup compared to software-based solutions. In conclusion, the proposed melody matching system is highly efficient, flexible, and scalable.

4. CONCLUSION

Design of pattern matching systems consists of modeling of patterns, matching algorithms, and scanner architectures, which need to be integrated properly to achieve best system performance. In this dissertation we explored all three components in design of pattern matching systems. More specifically, we target at matching problems of more complex patterns such as regular expressions and time series, and FPGA is selected as the scanner architecture to achieve high performance. In addition to design of the matching algorithm that fits the parallel architecture of FPGA, significant design efforts are made to remedy the notoriously time-consuming reconfiguration process that is inevitable for traditional FPGA based systems.

The proposed pattern matching system design includes pattern models, matching algorithms, and its proper mapping to the scanner architecture. As a result, the regexps (time series) are re-written into modular CCR (ACCR) structures that map well to the modular architecture of the FPGA based scanners, which runs the matching algorithm in parallel to achieve high speed and throughput. The matching algorithm is designed to maintain multiple active matching processes and resolve their ambiguities in the overlapped matching mode. The scanners are parameterized and memory-controlled, and therefore can be easily and rapidly reconfigured by simply writing to the on-chip memory. Combined together, the proposed models, algorithms, and scanner architectures compose a complete pattern matching system with hardware-level performance and software-level flexibility.

Following this design methodology, we first designed a CCR-based regexp matching system, named CES, for high speed network intrusion detection system. When implementing Snort [11], the counter based architecture of CCR can save up to 74% of memory bits compared to shift register based solutions. In addition to high performance, CES also features rapid deployment time (milliseconds) after the first-time implementation. It does not suffer from the time-consuming re-synthesis process when updating regexp patterns, which is inevitable for existing works and usually takes hours to complete. We then observed the similarity between the flexible matching capability of CCR and the elastic matching nature of music information retrieval problems, and designed a melody matching system. The music database and user humming are modeled into strings and ACCRs, respectively, and the problem is converted to a conventional pattern matching problem. Our results show that the ACCR structure can capture and model the elastic nature of user humming perfectly, leading to a high retrieval accuracy of 90.7%. The elastic matching algorithm is able to match ACCR based humming patterns approximately, and we designed the algorithm such way that it is easy to be parallelized onto FPGA architecture. The resultant melody matching engine outperforms its software competitors by up to 1135X, while updating of a new user query is kept extremely fast at less than a millisecond, thanks to the modular architecture of ACCR and the MME engine.

In the future, we plan to study the applicability of CCR model in other problem domains such as bioinformatics and DNA sequencing. The rich expressiveness offered by CCR and its corresponding matching algorithm may need to be carefully tuned to

meet application needs. For example, in DNA sequencing the character class can be reduced to only include the ‘A’, ‘C’, ‘G’, and ‘T’ symbols, and it may be desirable to track the alignment path for a reported match. We would also like to explore the relationship and difference between the elastic matching algorithm and the dynamic time warping (DTW) algorithm which is a traditional matching algorithm for general time series patterns and is notorious for its slow runtime. By studying how the two algorithms achieve elasticity differently, we may obtain insights on how to improve the performance of the DTW algorithm. Finally, we observed that CCR may be capable of modeling patterns in a statistical way, by assigning probabilities into character classes. This is similar to the application of the hidden Markov model in speech recognition problems, and we would like to explore whether CCR can be extended to a probabilistic model.

REFERENCES

[1] Regular expressions, http://en.wikipedia.org/wiki/Regular_expression, accessed August 2013.

[2] Time series, http://en.wikipedia.org/wiki/Time_series, accessed August 2013.

[3] D. Knuth, J. H. Morris, and V. Pratt, "Fast Pattern Matching in Strings", *SIAM Journal on Computing*, Volume 6, Issue 2, pp. 323–350, 1977.

[4] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (2nd Edition), Addison-Wesley, Boston, 2000.

[5] J. v. Neumann, “First Draft of a Report on the EDVAC”, 1945.

[6] Graphics processing unit, http://en.wikipedia.org/wiki/Graphics_processing_unit, accessed August 2013.

[7] Field programmable gate array, http://en.wikipedia.org/wiki/Field-programmable_gate_array, accessed August 2013.

[8] Single instruction multiple threads, http://semipublic.comp-arch.net/wiki/Single_Instruction_Multiple_Threads_(SIMT), accessed August 2013.

[9] CUDA, http://www.nvidia.com/object/cuda_home_new.html, accessed August 2013.

[10] OpenCL, http://www.khronos.org/opencl/, accessed August 2013.

[11] Snort, http://www.snort.org/, accessed September 2009.

[12] Bro, http://www.bro-ids.org/, accessed August 2013.

[13] Application Layer Packet Classifier for Linux, http://l7-

filter.sourceforge.net/, accessed August 2013.

[14] SpamAssassin, http://www.spamassassin.org/, accessed September 2009.

[15] I. Sourdis, J. Bispo, J. M.P. Cardoso and S. Vassiliadis, "Regular Expression Matching in Reconfigurable Hardware", *International Journal of Signal Processing Systems for Signal, Image, and Video Technology*, Volume 51, Issue 1, pp. 99-121, 2007.

[16] S. Pu, C.-C. Tan, and J.-C. Liu, "SA2PX: A Tool to Translate SpamAssassin Regular Expression Rules to POSIX", *Proceedings of 6th Conferences on Email and Anti-Spam*, 2009.

[17] Z. K. Baker, H.-J. Jung, and V. K. Prasanna, "Regular Expression Software Deceleration For Intrusion Detection Systems", *International Conference on Field Programmable Logic and Applications*, 2006.

[18] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithm to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection", *Proceedings of ACM Special Interest Group on Data Communication*, 2006.

[19] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall", *IEEE Symposium on Field-Programming Custom Computing Machines*, 2003.

[20] E. Berk and C. Ananian, "JLex: A Lexical Analyzer Generator for Java", http://www.cs.princeton.edu/~appel/modern/java/JLex/, accessed August 2013.

[21] D. Pao, "A NFA-based Programmable Regular Expression Match Engine", *ACM/IEEE Symposium on Architectures for Networking and Communications*

Systems, 2009.

[22] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata", *Proceedings of ACM Special Interest Group on Data Communication*, 2008.

[23] C. L. Hayes and Y. Luo, "DPICO: A High Speed Deep Inspection Engine Using Compact Finite Automata", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2007.

[24] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture for High-Throughput Regular Expression Pattern Matching", *ACM/IEEE International Symposium on Computer Architecture*, 2006.

[25] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, R.H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2006.

[26] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. D. Pietro, "An Improved DFA for Fast Regular Expression Matching", *ACM SIGCOMM Computer Communications Review*, Volume 38, Issue 5, pp. 29-40, 2008.

[27] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2007.

[28] M. Becchi and S. Cadambi, "Memory-Efficient Regular Expression Search

Using State Merging", *Proceedings of IEEE Conference on Computer Communications*, 2007.

[29] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for Acceleration SNORT IDS", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2007.

[30] I. Bonesana, M. Paolieri, and M. D. Santambrogio, "An Adaptable FPGA-based System for Regular Expression Matching", *Design, Automation and Test in Europe*, 2008.

[31] R.Sidhu and V. K. Prasanna, "Fast Regular Expression Using FPGAs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[32] R. Franklin, D. Carver, and B. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[33] M. Faezipour and M. Nourani, "Constraint Repetition Inspection for Regular Expression on FPGA", *IEEE Symposium on High Performance Interconnects*, 2008.

[34] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of Regular Expression Pattern Matching Circuits on FPGA", *Design, Automation and Test in Europe*, 2006.

[35] Y.-H. E. Yang and V. Prasanna, "Automatic Construction of Large-Scale Regular Expression Matching Engines on FPGA", *International Conference on Reconfigurable Computing and FPGAs*, 2008.

[36] C. R. Clark and D. E. Schimmel, “Scalable Pattern Matching for High Speed Networks”, *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.

[37] J. Moscola, Y. H. Cho, and J. W. Lockwood, “A Scalable Hybrid Regular Expression Pattern Matcher”, *IEEE Symposium on Field-Programmable Custom Computing Machines,* 2006.

[38] M. Becchi and P. Crowley, “Efficient Regular Expression Evaluation: Theory to Practice”, *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[39] Perl compatible regular expression, http://www.pcre.org/, accessed August 2013.

[40] POSIX regular expression, http://www.regular-expressions.info/posix.html, accessed August 2013.

[41] H. Wang, S. Pu, G. Kneze, and J.-C. Liu, “A Modular NFA Architecture for Regular Expression Matching”, *International Symposium on Field Programmable Gate Arrays*, 2010.

[42] Y.-H. Yang, W. Jiang, V. K. Prasanna, “Compact Architecture for High-throughput Regular Expression Matching on FPGA”, *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[43] K. Eguro, “SIRC: An Extensible Reconfigurable Computing Communication API”, *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 2010.

[44] Wireshark, http://www.wireshark.org/, accessed August 2013.

[45] Query by humming, http://en.wikipedia.org/wiki/Query_by_humming, accessed August 2013.

[46] M. A. Casey, R. Veltkamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney, "Content-Based Music Information Retrieval: Current Directions and Future Challenges", *Proceedings of the IEEE*, Volume 96, Issue 4, pp. 668-696, 2008.

[47] R. Typke, F. Wiering, and R. C. Veltkamp, "A Survey of Music Information Retrieval Systems", *International Symposium on Music Information Retrieval,* 2005.

[48] N. Orio, "Music Retrieval: A Tutorial and Review", *Foundations and Trends® in Information Retrieval*, Volume 1, Issue 1, pp. 1-90, 2006.

[49] J. Pickens, "Feature Selection for Polyphonic Music Retrieval", *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,* 2001.

[50] J. Pickens, "Harmonic Models for Polyphonic Music Retrieval", *Proceedings of the 11th International Conference on Information and Knowledge Management,* 2002.

[51] S. Doraisamy and S. M. R üger, "An Approach Towards a Polyphonic Music Retrieval System", *International Symposium on Music Information Retrieval*, 2001.

[52] MIDI, http://en.wikipedia.org/wiki/MIDI, accessed August 2013.

[53] L. R. Rabiner, “On the Use of Autocorrelation Analysis for Pitch Detection”, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Volume 25, Issue 1, pp. 24-33, 1977.

[54] A. d. Cheveigne´ and H. Kawahara, “YIN, a Fundamental Frequency Estimator for Speech and Music”, *The Journal of the Acoustical Society of America*, Volume 111, Issue 4, pp. 1917-1930, 2002.

[55] P. McLeod and G. Wyvill, “A Smarter Way to Find Pitch”, *International Computer Music Conference,* 2005.

[56] A. M. Noll, “Cepstrum Pitch Determination,” *Journal of the Acoustical Society of America*, Volume 41, Issue 2, pp. 293-309, 1967.

[57] A. Ghias, J. Logan, D. Chamberlin, and B. Smith, “Query By Humming: Musical Information Retrieval In an Audio Database”, *Proceedings of ACM Multimedia*, 1995.

[58] Y.-H. Tseng, “Content-Based Retrieval for Music Collections”, *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1999.

[59] S. Downie and M. Nelson, “Evaluation of a Simple and Effective Music Information Retrieval System”, *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2000.

[60] B. Cui, H. V. Jagadish, B. C. Ooi, and K.-L. Tan, “Compacting Music Signatures for Efficient Music Retrieval”, *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, 2008.

[61] J. Shifrin, B. Pardo, C. Meek, and W. Birmingham, “HMM-Based Musical Query Retrieval”, *Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital Libraries*, 2002.

[62] E. Unal, S. S. Narayanan, and E. Chew, “A Statistical Approach to Retrieval under User-dependent Uncertainty in Query-by-humming Systems”, *Proceedings of the 6th ACM SIGMM International Workshop on Multimedia Information Retrieval*, 2004.

[63] C.-C. Liu, J.-L. Hsu, and A. L.P. Chen, “An Approximate String Matching Algorithm for Content-Based Music Data Retrieval”, *IEEE International Conference on Multimedia Computing and Systems,* 1999.

[64] K. Lemström, “String Matching Techniques for Music Retrieval”, PhD thesis, University of Helsinki, Helsinki, 2000.

[65] N. H. Adams, M. A. Bartsch, and G. H. Wakefield, “Note Segmentation and Quantization for Music Information Retrieval*”, IEEE Transactions on Audio, Speech, and Language Processing*, Volume 14, Issue 1, pp. 131-141, 2006.

[66] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals". *Soviet Physics Doklady*, Volume 10, Issue 8, pp. 707–710, 1966.

[67] X. Wu, M. Li, J. Liu, J. Yang, and Y. Yan, "A Top-down Approach to Melody Match in Pitch Contour for Query by Humming", *Proceedings of International Conference of Chinese Spoken Language Processing*, 2006.

[68] G. P. Nam, T. T. T. Luong, H. H. Nam, K. R. Park, and S.-J. Park, "Intelligent Query by Humming System Based on Score Level Fusion of Multiple Classifiers", *EURASIP Journal on Advances in Signal Processing*, Volume 2011, Issue 1, pp. 1-11, 2011.

[69] J.-S. Jang and H.-R. Lee, "Hierarchical Filtering Method for Content-based Music Retrieval via Acoustic Input", *Proceedings of the 9th ACM International Conference on Multimedia*, 2001.

[70] L. Wang, S. Huang, S. Hu, J. Liang, and B. Xu, "An Effective and Efficient Method for Query by Humming System Based on Multi-Similarity Measurement Fusion", *International Conference on Audio, Language and Image Processing*, 2008.

[71] R. B. Dannenberg, W. P. Birmingham, G. Tzanetakis, C. Meek, N. Hu, and B. Pardo, "The MUSART Testbed for Query-by-humming Evaluation", *International Symposium on Music Information Retrieval*, 2003.

[72] N. Hu and R. B. Dannenberg, "A Comparison of Melodic Database Retrieval Techniques Using Sung Queries", *Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital Libraries*, 2002.

[73] J.-S. Jang, H.-R. Lee, and M.-Y. Kao, “Content-based Music Retrieval Using Linear Scaling and Branch-and-bound Tree Search”, *IEEE International Conference on Multimedia and Expo*, 2001.

[74] P. Senin, “Dynamic Time Warping Algorithm Review”, University of Hawaii at Manoa, Technical Report, Honolulu, 2008.

[75] Y. Zhu and D. Shasha, “Warping Indexes with Envelope Transforms for Query by Humming”, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003.

[76] MIDICSV, http://www.fourmilab.ch/webtools/midicsv/, accessed May 2012.

[77] Musical score for “happy birthday to you”, http://www.music-for-music-teachers.com/happy-birthday.html, accessed August 2013.

[78] YIN source code, http://audition.ens.fr/adc/sw/yin.zip, accessed May 2012.

[79] Altera white paper, “Understanding Metastability in FPGAs”, http://www.altera.com/literature/wp/wp-01082-quartus-ii-metastability.pdf, accessed May 2012.

[80] IOACAS data set, http://mirlab.org/dataSet/public/IOACAS_QBH_Coprus1.rar, accessed May 2012.

[81] Essen data set, http://www.esac-data.org/, accessed August 2013.

[82] MIREX, http://www.music-ir.org/mirex/wiki/MIREX_HOME, accessed August 2013.

[83] MIREX 2009 query by singing results, http://www.music-ir.org/mirex/wiki/2009:Query-by-Singing/Humming_Results, accessed August 2013.

[84] MIREX 2010 query by singing results, http://www.music-ir.org/mirex/wiki/2010:Query-by-Singing/Humming_Results, accessed August 2013.

APPENDIX A. PROOF FOR MIN-MAX ALGORITHM

A.1: Proof of Lemma 1

Lemma 1. *CN_1 always holds. For $1 < i \leq n$, when premises PR_i.(a) and PR_i.(b) hold, CN_i holds.*

Proof:

CCR_1 is permanently active. Its state changes from polling to busy at starting of B_k, and back to polling at starting *of* G_k, when MT_k advances out of CCR_1 and (MIN_1, MAX_1) are reset. Then, at starting of B_{k+1}, MF_{k+1} is generated and MB_{k+1} can use (MIN_1, MAX_1) without causing loss of any on-going matching processes, i.e., collision cannot occur at CCR_1. For its simplicity, we will not discuss the collision condition in CCR_1 in the rest of this appendix.

Next we consider CCR_i and CCR_x ($1 \leq x < i$) as stated in Lemma 1. Let us consider two consecutive matching bursts MB_k and MB_{k+1}, where MB_{k+1} is started at cycle p. The following cases show all possible positions of MT_k^p:

Case 1. MT_k^p is located at CCR_j, $j > i$.

Case 2. MT_k^p is located at CCR_j, $x < j \leq i$.

Case 3. MT_k^p is located at CCR_j, $j \leq x$.

In all three cases, the matching front of MB_{k+1} has to pass CCR_x first before it can advance to CCR_i. Next, we show that when PR_i.(a) and PR_i.(b) hold, MF_{k+1} can never catch up with MT_k .

In Case-1, the matching tail of MB_k has already passed CCR_i at cycle p and it can never move back. Therefore, for any cycle q later than p, MF_{k+1}^q will never catch up with MT_k^q at CCR_i and hence CN_i holds.

In Case-2, let q denote the cycle when MF_{k+1}^q reached CCR_x (i.e., CCR_x is activated by CCR_{x-1} and enters polling state). Before the matching front can pass through CCR_x and advances to its successor, at least b_x^L characters must have been matched by CCR_x according to IR-2. By PR_i.(b), these characters are not acceptable to any CCR between CCR_{x+1} and CCR_i (inclusive). By IR-4, all these CCRs will be reset and thus the matching tail of MB_k will advance beyond CCR_i. Therefore, the matching front of MB_{k+1} can never catch up with the matching tail of MB_k at CCR_i, i.e., CN_i holds.

In Case-3, let us also assume that MF_{k+1}^q reached CCR_x at cycle q. By PR_i.(a), MF_{k+1} cannot catch up with MT_k at any CCR between CCR_1 and CCR_x, and thus at cycle q MT_k^q must have advanced beyond CCR_x. MT_k^q is either beyond CCR_i, or between CCR_{x+1} and CCR_i (inclusive), we can show that CN_i holds based on an analysis process similar to that of Case 1 or 2. ■

A.2: Proof of Lemma 2

Lemma 2. *For $1 < i \leq n$, when PR_i.(a) holds and PR_i.(c) holds, CN_i holds.*

Proof:

The following three conditions in PR_i.(c) need to be considered.

Condition 1: $b_i^U = 1$.

Condition 2: $b_i^L = b_i^U$.

Condition 3: $b_{i-1}^{L} \geq b_{i}^{U}$.

Condition-1 essentially states that no collision could occur at CCR_i, because the matching tail can stay for only one cycle. That is, when the matching tail of a matching burst advances to CCR_i, MIN_i increments to 1. But at the next cycle, (MIN_i, MAX_i) will be reset to 0 per IR-4 because $MIN_i = 1 = b_i^U$, and the matching tail will pass down to a successor CCR. No resource contention can occur in the duration of one cycle.

In Condition-2, let $b_i^L = b_i^U = m$. By definition of constraint repetition, CCR_i can be rewritten as a concatenation of m identical CCRs, $CC_i\{1,1\}$. According to condition (1), all of these CCRs, and thus CCR_i as a whole, is collision free.

In Condition-3, a collision could occur at CCR_i, only when MT_k is at CCR_i, and then MF_{k+1} in CCR_{i-1} catches up with MT_k (in CCR_i). For an arbitrary input character z, if z belongs to CC_{i-1} but not CC_i, (or z belongs to CC_i, but not CC_{i-1}) the state of CCR_i (or CCR_{i-1}) is changed to idle/polling, and therefore no collision could occur between MT_k and MF_{k+1}. When $z \in CC_{i-1} \cap CC_i$, both CCRs update the values of their counters. Given that $b_{i-1}^{L} \geq b_{i}^{U}$, per IR-4 and CR-3, MT_k will always advance beyond CCR_i no later than the cycle at which MAX_{i-1} reaches the value b_{i-1}^{L} for CCR_{i-1} to assert AS_{i-1}. That is, no collision could occur at CCR_i. ■

A.3: Proof of Lemma 3

Lemma 3. *For $1 < i \leq n$, when PRi.(d) holds, CN_i' holds even if CN_i does not hold.*

Proof:

When the matching front of MB_k reaches CCR_i, it will leave only upon a non-acceptable character. In this case, MIN_i is no longer needed because it is designed to determine whether or not the minimum number of possibly matched characters has exceeded the upper bound, in this case ∞. When MF_{k+1} reaches CCR_i at cycle *y* and a collision occurs, all characters arrived up to cycle *y* must be acceptable to CCR_i because MT_k did not leave CCR_i. Then, MF_{k+1} is essentially merged with MT_k to form a larger matching burst starting at CCR_i. Resetting a non-zero MIN_i does not lose any matching information, because it is not needed for CCR_i to function correctly. Hence, no false detection will happen. ■

A.4: Proof of Theorem 1 and 2

Theorem 1. *If R_n is collision-free and S_n is acceptable to R_n, MIN-MAX will report a match. (Sufficient condition)*

Theorem 2. *If R_n is collision-free and MIN-MAX reports a match, there must have been a string S_n that has been matched by R_n. (Necessary condition)*

To prove these two theorems, we will first derive two lemmas for detailed analysis of low level signal transitions of a CCR, and then we will use high level induction of interactions among CCRs to show the correctness of the two theorems. To ease the discussion without loss of generality, we only consider regexps composed of concatenated CCRs. The regexps discussed in this section are assumed to be collision free unless otherwise specified.

Recall that $S_n = \{s_1, s_2, \ldots, s_n\}$ denotes a set of consecutive string segments acceptable to $R_n = CCR_1 \bullet CCR_2 \bullet \ldots \bullet CCR_n$, where s_i is acceptable to CCR_i and the length

of s_i is denoted as L_i, $1 \leq i \leq n$. A string s_i is acceptable to CCR_i if and only if each character of s_i is acceptable to CC_i, and L_i falls in the range of $[b_i^L, b_i^U]$. To facilitate discussion, let t_i denote the value of MAX_i at the beginning of s_i.

Lemma 4. *When PR_i holds and input s_i is acceptable to CCR_i, CN_i also holds.*

Here, PR_i denotes the premise "CCR_i is active and $MIN_i = 0$", and CN_i denotes the conclusion "CCR_i asserts AS_i to its successor CCR_{i+1}", respectively. When CN_i holds, it implies that PR_{i+1} also holds according to IR-3. In other words, the inductive relationship between (PR_i, CN_i) and (PR_{i+1},CN_{i+1}) also holds. This will provide a basis for the inductive analysis between behaviors of adjacent CCRs, and therefore proof of Theorem 1.

Proof of Lemma 4:

If PR_i holds and an acceptable string s_i is fed in, L_i falls between b_i^L and b_i^U, and we can conclude from CR-2 and CR-3 that

$$MIN_i \leq 0 + L_i \leq b_i^U. \quad (1)$$

CR-1 will lead to the following equation

$$MAX_i = t_i + L_i \geq b_i^L. \quad (2)$$

Based on (1) and (2), CN_i will hold because IR-2 is satisfied. ■

Proof of Theorem 1.

If $S_n = \{s_1, \ldots, s_n\}$ is acceptable to $R_n = CCR_1 \bullet \ldots \bullet CCR_n$ with s_i matching CCR_i, respectively, we can perform an inductive, step by step analysis to prove correctness of

Theorem 1. For CCR_1, according to IR-1, $ACTIVE_1$ is always 1, and MIN_1 is always 0, i.e., PR_1 always holds. Therefore by Lemma 1, an acceptable substring s_1 will lead to CN_1 and thus PR_2. By induction, we can conclude that for CCR_n, the acceptable substring s_n will lead to CN_n. According to IR-5, it is guaranteed that CCR_n will generate a match signal under MIN-MAX. ■

Lemma 5. *When PR_i holds, CN_i also holds.*

Here, PR_i denotes the premise "CCR_i asserts AS_i to its successor CCR_{i+1}", and CN_i denotes the conclusion "CCR_i has just matched an acceptable string s_i", respectively. We aim to establish an inductive relationship between (PR_i, CN_i) and (PR_{i-1}, CN_{i-1}), so that Theorem 2 can be proved on the basis of Lemma 5.

Lemma 5 represents a "look back" scenario which states that AS_i can be asserted only if CCR_i has matched s_i. The goal of this proof process is to show two points. The first one is that when AS_i is asserted, we can always take a *look back search* from the current input character to locate an acceptable substring s_i, whose acceptance is defined by the condition that $b_i^L \le L_i \le b_i^U$, i.e., when PR_i holds, CN_i holds. The second point is that for the derived L_i value, PR_{i-1} holds before matching of s_i starts. That is,

$$CN_i \rightarrow PR_{i-1}. \tag{3}$$

Proof of Lemma 5:

For CCR_i, we will prove the correctness of Lemma 5 in the following two cases, where the 'B' prefix reflects the backward calculation direction for matching lengths.

B-Case 1. $b_i^L \le MAX_i \le b_i^U$ at assertion of AS_i.

B-Case 2. $MAX_i > b_i^U$ at assertion of AS_i.

The two cases are classified based on values of the MAX register when AS_i is asserted. In B-Case 1, there are MAX_i successive acceptable characters that have been matched by CCR_i according to CR-1. We can choose these MAX_i successive characters to be s_i, i.e., select L_i to be MAX_i. L_i falls between b_i^L and b_i^U and therefore s_i is acceptable to CCR_i, i.e., CN_i holds.

According to CR-1, CCR_i can start matching of s_i, i.e., MAX_i can begin incrementing, if and only if it is active, which implies that CCR_{i-1} was asserting AS_{i-1} to CCR_i before the beginning of s_i, and therefore the inductive relationship (3) also holds.

In B-Case 2, there are also MAX_i successive acceptable characters that have been matched by CCR_i according to CR-1. Since MAX_i exceeds the upper bound, we can choose b_i^U successive characters to be s_i, i.e., select L_i to be b_i^U. As such, CN_i holds. Note that the inequality

$$MIN_i \leq b_i^U \tag{4}$$

must hold. Otherwise, CCR_i would be inactive according to IR-4, and it would not have asserted AS_i, contradicting the condition under analysis. From CR-2, CR-3 and (4) we can infer that MIN_i is 0 before matching of s_i, which implies that CCR_{i-1} was asserting AS_{i-1} at that point, and therefore (3) holds.

In summary, we have shown in the two cases listed above that when PR_i holds, s_i can always be recovered for CCR_i such that CN_i holds, and the inductive relationship in (3): $CN_i \rightarrow PR_{i-1}$ also holds. ■

Proof of Theorem 2.

When CCR_n of $R_n = CCR_1 \bullet CCR_2 \bullet \ldots \bullet CCR_n$ asserts its AS_n, PR_n holds by IR-5. According to Lemma 5, we can retrospect s_n by using the rules for selection of the L_i value as suggested in the two B-cases such that CN_n and PR_{n-1} holds. We can then apply this retrospection procedure recursively to identify a sequence of matched substrings s_n, s_{n-1}, …, and s_1, which composes a matched input string $S_n = \{s_1, s_2, \ldots s_n\}$ with matching lengths of $\{L_1, L_2, \ldots L_n\}$. In other words, the MIN-MAX algorithm is guaranteed to produce the matching lengths when a match is made. ■

APPENDIX B. CONTEXT-DEPENDENT FEATURES

B.1: Zero-Width Patterns

Zero-width patterns are look-around assertions which match a specific pattern without consuming any input characters. A positive assertion is matched when the input is acceptable to the pattern, otherwise a negative assertion is matched. Look-behind assertions match the input string up to the current matching position, and look-ahead assertions match the input string following the current matching position. For example, the regexp "[a-zA-Z](?=[0-9])" contains a positive look-ahead zero-width pattern "(?=[0-9])", and this regexp states that it can match an alphabet letter followed by a digital number in the input string, but the digital number should not be consumed, i.e., it can still be used to match a subsequent CCR term. Activation signals tailored to implement zero-width patterns are discussed next.

To simplify the discussion, we only consider the case of zero-width pattern of a single CCR. For an example of the zero-width pattern "$CCR_i(?=CCR_{i+1})CCR_{i+2}$", which is a positive look-ahead assertion, we can add an activation signal path from CCR_i to CCR_{i+2} so that CCR_{i+1} and CCR_{i+2} can perform matching in tandem. Furthermore, CCR_{i+1}'s matching outcome is latched in a flip-flop (FF). The match signal output by the final CCR cannot directly indicate a match as in standard CCR designs. Instead, it should trigger reading of all the previous FFs' contents, to be ANDed all together to decide whether or not the rule is matched. This way, if the regexp is matched without satisfying all positive look-ahead zero-width patterns, the matching engine still reports no match. Figure. 25 illustrates the implementation of the flowing regexp R_1

R_1: $CCR_1(?=CCR_2)CCR_3(?=CCR_4)CCR_5$

The zero-width pattern "$CCR_i(?!CCR_{i+1})CCR_{i+2}$", i.e., a negative look-ahead assertion, can be solved in a similar fashion, except that the output of the FF is inverted. This way, if the regexp gets matched with any of the zero-width CCRs also matched (i.e., violation of negative assertion), the final output will still be no match.

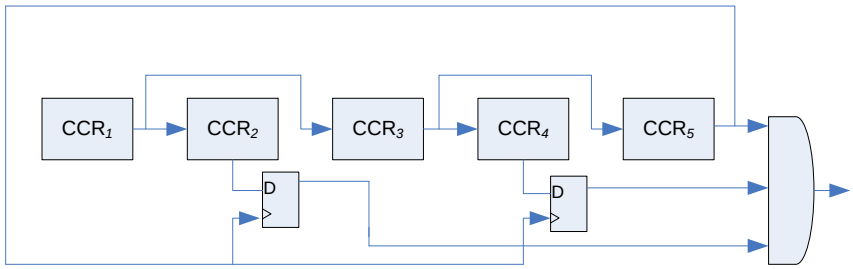

Figure 25. Implementation of R_1

Look-behind assertions need to be handled differently. "(?<=[0-9]) [a-zA-Z]" is a positive look-behind zero-width pattern and it matches an alphabet following a digital number without consuming the input digital number when this term is active. In the example of "$CCR_{i-1}(?<=CCR_i)CCR_{i+1}$", the positive look-behind pattern CCR_i needs to start checking against the input string simultaneously with CCR_{i-1}, and CCR_{i+1} gets activated if and only if both CCR_{i-1} and CCR_i produce a matching signal. This way, CCR_i consumes no input string and the subsequent CCRs can start matching only if

CCR_i gets matched (i.e., the positive look-behind assertion is satisfied). Figure. 26 depicts the implementation detail of the example regexp

$$R_2: CCR_1(?<=CCR_2)CCR_3(?<=CCR_4)CCR_5$$

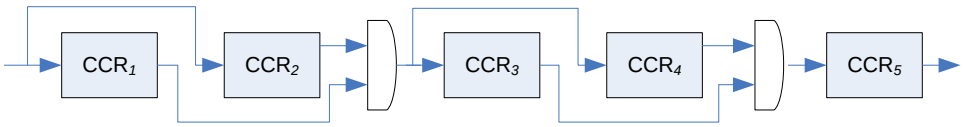

Figure 26. Implementation of R_2

The zero-width pattern "$CCR_{i-1}(?<!CCR_i)CCR_{i+1}$" (negative look-behind assertion) can be solved by a similar design, where an inverter is added to the output of the zero-width CCR. This way, subsequent CCRs can start matching only if the zero-width CCR outputs a no match, i.e., the negative look-behind assertion is satisfied.

B.2: Back-Reference

When a back-reference is made to a sub-regexp surrounded by parentheses, the input(s) accepted by the sub-regexp would need to be used to replace the back-reference term. When the matching process proceeds to the back-referenced term, it needs to match exactly what has been matched before. Back-references have the format of "*i*", where *i* is an integer index of the sub-regexps. For example, the regexp "(a|b)cd\1" will match a string whose first character is '*a*' or '*b*', followed by "*cd*", and then followed by

the exact character that sub-regexp of "(a|b)" has matched. Therefore, input strings "*acda*" or "*bcdb*" will be accepted, but "*acdb*" or "*bcda*" will not.

Back-reference is supported in [29] by storing local data in BRAM. Now that BRAM is used for storage of character classes in CES, we propose to support back-reference by adding a Distributed RAM (D-RAM) synthesized from a slice of LUT to each CCR as a memory to store the matched substring. A multiplexer-based memory access scheme would allow each CCR of a regexp to read all of their allocated D-RAM, because any of these D-RAMs may be referred to for matching of a subsequent back-reference CCR. Figure. 27 shows the implementation for the following example of R_3:

$$R_3: (CCR_1)\backslash 1(CCR_3)\backslash 2.$$

The '\1' and '\2' terms are implemented as CCR_2 and CCR_4, respectively. D-RAMs of CCR_2 and CCR_4, and multiplexers of CCR_1 and CCR_3 are omitted in this figure to make the figure more readable.

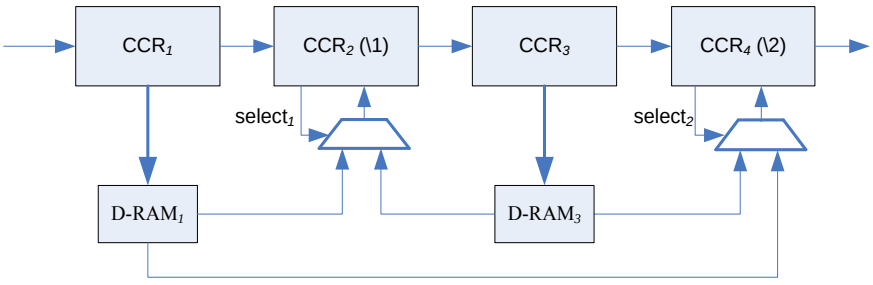

Figure 27. Implementation of R_3

A modular architecture to support these features needs to consider all possible compounded sub-modules and their interconnections. As a result, in CES we only consider the condition that terms being used for a back-reference are in the simple form, i.e., a single CCR. For the high cost of D-RAMs and their routing resources, back-reference should be supported by a dedicated CES, not as a generic feature to be included in every CCR engine.