A Dynamically Reconfigurable Pattern Matcher for Regular Expressions on FPGA

Tom DAVIDSON [a] and Mattias MERLIER [a] and Karel BRUNEEL [a]
Dirk STROOBANDT [a]
[a] *Ghent University, Elis department, CSL, HES team, Sint-Pietersnieuwstraat, 41, 9000 Gent, Belgium. [name.surname]@ugent.be*

Abstract. In this article we describe how to expand a partially dynamic reconfigurable pattern matcher for regular expressions presented in previous work by Divyasree and Rajashekar [2]. The resulting, extended, pattern matcher is fully dynamically reconfigurable. First, the design is adapted for use with parameterisable configurations, a method for Dynamic Circuit Specialization. Using parameterisable configurations allows us to achieve the same area gains as the hand crafted reconfigurable design, with the benefit that parameterisable configurations can be applied automatically. This results in a design that is more easily adaptable to specific applications and allows for an easier design exploration. Additionally, the parameterisable configuration implementation is also generated automatically, which greatly reduces the design overhead of using dynamic reconfiguration. Secondly, we propose a number of expansions to the original design to overcome several limitations in the original design that constrain the dynamic reconfigurability of the pattern matcher. We propose two different solutions to dynamically change the character that is matched in a certain block. The resulting pattern matcher, after these changes, is fully dynamically reconfigurable, all aspects of the implemented regular expression can be changed at run-time.

Keywords. Regular Expression Matching, FPGA, Run-Time reconfiguration, Dynamic Circuit Specialization

Introduction

Regular expressions are widely used in the domain of Computer Sciences. They describe patterns in a confined and standardized way. Table 1 shows an example syntax. Applications that use regular expressions range from scripting languages (eg. PHP) to Network Intrusion Detection Systems (NIDS) such as Snort [1]. Regular expression matching is a computationally intensive problem. Furthermore, the patterns that have to be matched can vary frequently. Because of this need for flexibility, pattern matching is usually done in software, which, in general, is flexible enough to be able to quickly change the regular expressions that are being matched. Snort [1] is such a pure software-based NIDS and is widely used. The payload of network packets has to be matched with a database of regular expressions. Based on this matching, malicious packets will be dropped. As network rates are increasing to higher data-rates, so is the need for high performance sys-

tems. However the inherent sequential nature of software limits the performance of such a regular expression matcher, especially for large sets of expressions. Hardware solutions are inherently parallel and should be able to provide better performance. But network security applications also require frequent updates of the regular expression database. To support this, reconfigurable hardware like FPGAs can be used for to implement the NIDS. The normal update process for FPGAs includes adapting the HDL to the current requirements, synthesis, place and route (PAR) and the complete reconfiguration of the FPGA. This whole process is called the toolflow for configuring FPGAs. This generation process might take up to a few hours, especially for large sets of patterns. Having to delay SNORT updates for a few hours can leave the system vulnerable. The solution proposed in this article uses parameterisable configurations[3,4]. With this new technique, an FPGA design can be made reconfigurable automatically. All NP-hard problems in the FPGA toolflow are solved off line. Specializing the design is done at run-time by evaluating closed-form boolean function. Once these boolean functions are evaluated, specific parts of the FPGA are reconfigured. This process, evaluating the boolean functions and reconfiguring parts of the FPGA, takes in the order of milliseconds, a much more acceptable down-time.

Table 1. Supported regular expression syntax

Syntax	**Description**
a	All ASCII alphanumerics
`^$*+?|()[{.-\`	Metacharacter, each of these has a special meaning
.	Dot: matches any character other than newline
`\?`	Backslash combined with a metacharacter reduces it to its literal meaning
`[abc]`	Character class, matches each character within brackets
`[^abc]`	Reversed character class, matches any character other than these between brackets
`[a-zA-E]`	Character class with range
RegExp1RegExp2	Concatenation: Regular expression 1 followed by regular expression 2.
RegExp1|RegExp2	Union: Regular expression 1 OR regular expression 2.
RegExp∗	Kleene Star: Matches zero or more occurrences of the regular expression
RegExp+	Plus: Matches one or more occurrences of the regular expression
RegExp?	Question Mark: Matches zero or one occurrences of the regular expression
RegExp{M,N}	Constraint Repetition. Matches between M and N occurrences of the regular expression
^RegExp	Caret. Only matches the regular expression at the beginning of the string.
RegExp$	Dollar. Only matches the regular expression at the end of the string.
Flags	**Description**
i	Case sensitivity
s	Dot (`.`) matches any character, including new line
m	^ en $ match at each newline

1. Parameterisable Configurations

The technique of parameterisable configurations is a state-of-the-art method for designing dynamic reconfigurable hardware for Dynamic Circuit Specialization on the *Register Transfer Level* (RTL) [5]. In this technique, we take the slowly varying inputs, called the parameters, and specialize the circuit for actual parameter values. Such a specialized circuit is both smaller and faster than the original generic circuit [3,4,6]. If a parameter value changes, a new specialized circuit is generated on line for that specific value.

For an in depth explanation of the parameterisable configuration technique, we refer to [4]. We limit ourselves here to the fact that the FPGA toolflow is adapted to make use

of this technique. We start with a normal VHDL description, in which the parameters are selected using annotations. These parameters are a set of slowly varying inputs to the original VHDL design. Next, the adapted FPGA toolflow generates both a master configuration (independent of the parameters) and a method for specializing the circuit for actual parameter values. In the current proposal, we will only change the truth tables of certain LUTs, the routing and placement stay fixed. Hence, the master configuration contains the place and route information for all the LUTs in the circuit. To specialize a circuit for a specific parameter value, we change the truth tables of some of the LUTs in the master configuration. The new truth table values are calculated on line and are based on evaluating closed-form boolean expressions, something that can be done very quickly.

It is important to realize that the only decision the designer has to make is the selection of the parameters, everything else is generated automatically by the toolflow. Since in the case of a NIDS, such as SNORT, choosing the parameters is straightforward, the design cost of using Dynamic Circuit Specialization is very low. And, as we will show later, it succeeds in getting similar results as hand-optimized run-time-reconfigurable designs.

2. Previous work

Most of the work done on implementing regular expressions in hardware requires the complete regeneration of the hardware when the regular expression changes. These designs aren't suitable for applications where fast dynamic reconfiguration is required. However there are a few exceptions that allow some form of run-time reconfiguration.

The first approach is the ASIC implementation of Brodie et al. [7]. The second is the micro controller implementation of Baker et al. on FPGA [8]. Both of these implementations use memory to describe the regular expressions. Overwriting this memory can be used to change these regular expressions at run-time. However, both of these solutions feature a more sequential implementations. Our interest is in developing a massively parallel regular expression matcher.

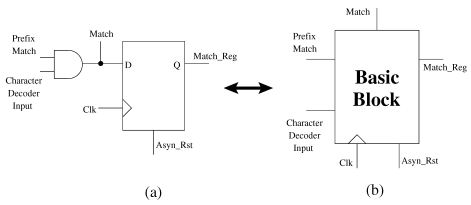

Figure 1. Structure of a basic block

Lastly Divyasree and Rajashekar propose a pattern matcher built by cascading generic blocks [2]. This pattern matcher has a limited run-time reconfigurability, which we will discuss in more detail later. The generic blocks are able to implement regular expressions and are dynamically reconfigurable to a certain, limited, degree. These generic blocks will be used as the starting point for our contribution, so we will discuss them in detail.

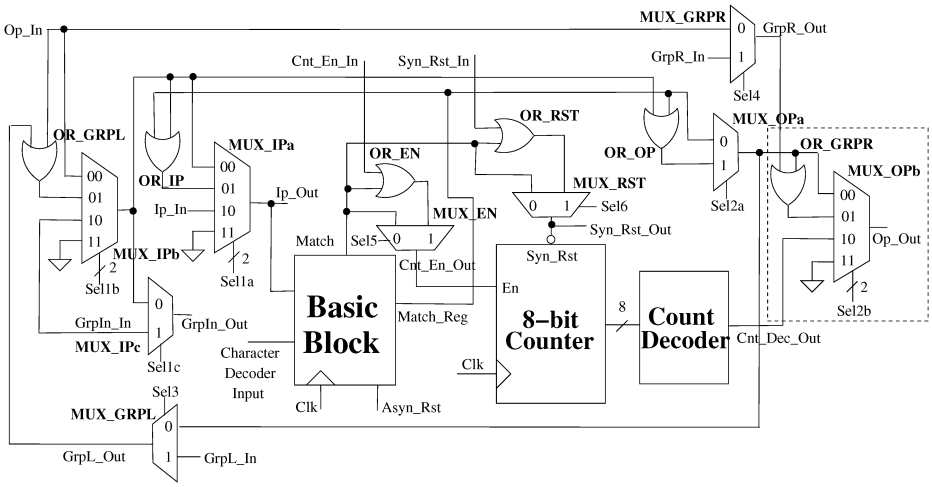

Figure 2. Structure of a full generic block

Figure 2 shows the structure of one such generic block. This block is capable of matching one character and applying regular expression operators to it, such as '*', '+' and '?'. The basic block featured here is shown in more detail in Figure 1. This basic block gives a latched match signal when the matching character is detected by the global decoder. Using the counter and decoder the constrained repetition functionality is implemented. This allows the generic block to match, for example, 'a{5,10}'. This expression matches if 'a' is repeated between 5 and 10 times. One of the most important features of this generic block however is that all these regular expression operators can also be applied to cascades of generic blocks. This means that, for example, '(ab*c?){2,7}' can also be matched. Figure 3 shows how these generic blocks need to be connected to implement this functionality.

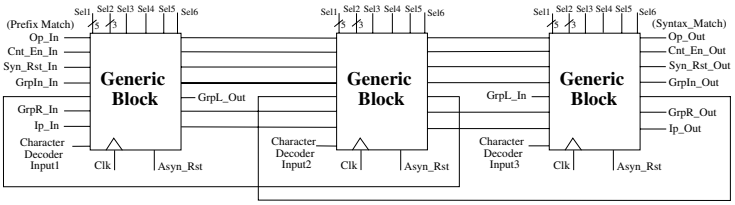

Figure 3. Cascade of generic blocks

The architecture of the full regular expression matcher from [2] also features a global decoder that decodes an 8-bit character to one of the 256 bit lines. How these bit lines are connected determines which characters are matched in the generic blocks.

2.1. Limitations

The proposed architecture for a dynamic pattern matcher by Divyasree and Rajashekar has some serious limitations and shortcomings if we need a fully dynamic pattern

matcher. In the next section we will describe our proposed solutions to overcome these limitations.

1. First of all the proposed implementation is optimized by designing on a low level. When small changes have to be made, these changes also have to be made on this low level, thus greatly increasing the design cost. For example, Divyasree and Rajashekar describe in [2] how they optimized the counter and the decoder. They first designed the counter and decoder on the LUT level, to make sure a minimal amount of LUTs are used. Next, for the placement they add relative design constraints on the location of these LUTs, to guarantee optimized placement. If, for example, we have a regular expression application that needs a larger counter range, all this work needs to be redone.
2. Secondly, dynamic character changes are not possible. For example the regular expression $a + b * c?$ can be changed dynamically to $a * b\{5\}c+$, but not to $d * b\{5\}c+$. This problem is a result of the used architecture: a global decoder receives the incoming character and sends a one-bit matching signal to each generic block that matches that character. The routing in the design is fixed and the decoder is static. A character change on a generic block thus requires the rerun of the router.
3. The last limitation involves the use of character classes, special characters and flags. These functionalities are possible with the proposed architecture but they are not dynamically adaptable.

3. Our Proposed Solutions and Results

To overcome the limitations described in Section 2.1, we have re-implemented the design of [2] using our parameterizable configuration tool flow. This increases the abstraction level of the design and makes the design process faster and easier while allowing easy design space exploration (Section 3.1). To remedy the limits in character sets supported, we propose extensions to either the generic block or the global decoder (Section 3.2).

3.1. RTL implementation

With the use of parametrisable configurations the design can be done on the RT level, without having to optimise anything by hand. For example, in the solution described by [2], certain design constraint have to be added to the VHDL code to guarantee optimised placement. This is not the case in our design. Additionally, in [2], they have to determine by hand how the LUT truth tables will have to change. Changing some small design detail, such as the counter width, can force the designers to redo all this work. This is also done automatically by the parametrisable configurations toolflow.

Our approach thus, results in a more efficient development of generic blocks and also allows for an easier design exploration. Now, the designer can more easily optimize the blocks for specific regular expression applications. To show that our approach is not only faster and easier but also finds similar results, we have made an implementation with the same general assumptions as in [2]. This means we used the same counter length and use SNORT [1] as the targeted application when faced with design choices. However, in

our design, it is very easy to change these assumptions and still obtain optimized results without much effort.

To be clear, we implemented a fully reconfigurable pattern matcher on a Xilinx Virtex II Pro XC2VP30 FF896. All the numbers below are results from our experiments on this design.

Our experiments have shown that using parameterizable configurations to design the generic block results in a circuit with the same area size as the manually optimized design (see Table 2). These results are based on correctly working generic block implementations on a Virtex II Pro FPGA. Given the clear advantages of parameterisable configurations, every approach suggested in this paper uses parameterisable configurations wherever this is possible.

Table 2. The resource usage of the different generic block implementations

	Unoptimized	Hand optimized	Parameterisable configurations
LUTs	48	24	24

The overhead introduced by parameterisable configuration, the specialisation overhead, can be split up in two parts. The time needed to calculate the new values, called the evaluation overhead, and the time needed to reconfigure the FPGA with these new values, called the reconfiguration overhead.

The following measurements were taken from the Virtex II Pro implementation. The specialisation overhead for one generic block is 26,75 μseconds. The evaluation overhead is 24,26 μseconds and the reconfiguration overhead is 2,49 μseconds. The reconfiguration of several generic blocks can be done in parallel. This means that the reconfiguration overhead will increase much slower than linear with additional generic blocks. The evaluation time is linear with the amount of generic blocks that need to be reconfigured.

3.2. Dynamic character changes

To support dynamic character changes, two methods are proposed. The first one extends the current generic block (Section 3.2.1) by adding extra logic. The second solution is memory-based and modifies the global decoder (Section 3.2.2).

In most cases the second approach is preferable, since it has a fixed cost, regardless of the number of special characters or character classes the application requires. In some cases, for example, when implementing only a single regular expression or when there are not enough BRAM available on the FPGA, the first approach could be more efficient. The first approach only adds to the LUT cost of the generic block, whereas the second approach requires the usage of either BRAMs, or a large number of LUTs, used as distributed RAM.

3.2.1. LUT-based implementation

In the first approach, we omit the global decoder and extend the generic block. Each generic block now receives 8 character bits instead of one bit. This means each extended generic block has its own character decoding block. There are several options for the complexity and size of this block.

We start with a character decoding block that only matches single characters. Using parameterisable configurations, with the character input as the parameter, the size of this block is reduced from 6 to 3 LUTs. These are results from our actual implementation on a Virtex II Pro FPGA. However, this block has some limitations: there is no support for special characters and character classes.

To add special characters and character classes extra area is needed. Each extra special character and character class requires a character decoder block with extra LUTs. In this case however, case-sensitivity ($/i$) and negation ($\sim$) can be added without an extra LUT cost. In Table 3 we show an overview of the number of LUTs needed for special characters and for character classes. This table shows the cost of implementing the standard 3-LUT character decoder, with the addition of the special characters and character classes listed. The most flexible implementation would require 14 LUTs.

Table 3. The total resource usage (in LUTs) of the different character decoding blocks

Notation	Description	LUT cost, after applying parameterisable configurations
/d, [0-9]	Numbers	5
/s	Whitspace	6
$[A - Z]$	Capital letters	6
$[a - z]$	Lower-case letters	6
/w	Combines the previous three	10
	A combination of all 4 groupings	14

With this extension, each generic block can be configured to match any character, a special character, a character class or a combination of one or more. The set of possible character classes and special characters is limited but can be tuned for specific applications. As shown above, adding the basic character decoder functionality, will cost 3 additional LUTs. This 3-LUT solution includes normal characters, case sensitivity ($/i$), matching any character $/s$ and negation ($\sim$). Including a small set of character classes would require extra LUTs, as shown in Table 3, but this increases the flexibility of this solution significantly.

3.2.2. RAM-based implementation

In the second approach we keep the generic block and extend the decoder to provide dynamic character changes. This extended design is RAM-based. Basically, this decoder is a very wide RAM memory that takes the character inputs as its address and outputs a match signal to all the generic blocks that need to be activated.

When a new character is read, the global decoder will activate the match inputs of all generic blocks that should match the character. It does this by reading the appropriate outputs from the RAM memory. For each 8-bit character a string of bit values is stored, one bit value for each generic block. These bit values are stored in a RAM memory, so they can be changed when a generic block needs to match a different character or character class. This change only involves writing the corrected bit values to the RAM memory and can be done easily.

On an FPGA there are several ways to implement this RAM-memory. One option is using LUTs as distributed RAM, another is using the available Block RAMs. For the distributed RAM implementation we use the method Xilinx tools provides us, this results

in 21 LUTs, used as RAMs, per generic block. For the BRAM implementation, we need 256 bits per generic block. With a quick calculation we can see that the BRAM to LUT ratio of most Xilinx Virtex and Spartan 6 FPGAs is high enough to allow us to implement the BRAM based design. One normal generic block requires 24 LUTs, and for each generic block 256 BRAM bits are needed, resulting in a need for 10.66 BRAM bits per LUT for the FPGAs.

One of the main advantages of this solution over the previous one is that it allows the implementation of dynamic character reconfiguration without adding to the LUT-area cost of the design. This means a larger number of generic blocks can be implemented on the FPGA. Another big advantage is the flexibility of this approach. There is no need to analyse the exact application, as all special characters and character classes can be implemented without adding any logic. Off course, as soon as there is a limit to the number of BRAMs available or if there is no need to implement a large number of generic blocks, it could be better to use the first solution.

4. Conclusion

We have shown that the parameterisable configurations technique can achieve the same specialised circuits as hand crafted specialised circuits, but with a much lower design overhead. Additionally we extended and optimized a previous, partially reconfigurable, pattern matcher design. This results in a fully dynamic pattern matcher for regular expressions.

With these features this regular expression matcher should be seen as a run-time reconfigurable platform that can implement any regular expression and can switch between implemented regular expressions in milliseconds. There are still limitations: the regular expressions must have less characters than the total amount of available generic blocks and the structure is such that only single level groupings are possible.

References

[1] Martin Roesch, "Snort - Lightweight Intrusion Detection for Networks", in *LISA 1999: 13th System Administration Conference*, pp. 229–238, 1999.

[2] J. Divyasree, H. Rajashekar, and K. Varghese, "Dynamically reconfigurable regular expression matching architecture," in *Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, (Washington, DC, USA), pp. 120–125, IEEE Computer Society, 2008.

[3] Karel Bruneel and Dirk Stroobandt, "Reconfigurability-Aware Structural Mapping for LUT-Based FPGAs", in *ReConFig 2008*,pp. 223–228, 2008.

[4] Karel Bruneel and Dirk Stroobandt,"Automatic generation of run-time parameterizable configurations," in *in 18th International Conference on Field Programmable Logic and Applications (FPL 08)*, pp. 361–366, 2008.

[5] K. Bruneel, P. Bertels, and D. Stroobandt, "A method for fast hardware specialization at run-time," in *FPL* (K. Bertels, W. A. Najjar, A. J. van Genderen, and S. Vassiliadis, eds.), pp. 35–40, IEEE, 2007.

[6] Karel Bruneel and Fatma Abouelella and Dirk Stroobandt, "Automatically mapping applications to a self-reconfiguring platform", in *DATE 2009*, pp. 964–969, 2009.

[7] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *SIGARCH Comput. Archit. News*, vol. 34, pp. 191–202, May 2006.

[8] Z. K. Baker, H. jip Jung, and V. K. Prasanna, "Regular expression software deceleration for intrusion detection systems," in *in 16th International Conference on Field Programmable Logic and Applications (FPL'06)*, pp. 28–30, 2006.