EFFICIENT FAILURE PROCESSING ARCHITECTURE IN REGULAR EXPRESSION PROCESSOR

SangKyun Yun

Department of Computer and Telecom. Engineering,
Yonsei University, Wonju, Korea
skyun@yonsei.ac.kr

ABSTRACT

Regular expression matching is a computational intensive task, used in applications such as intrusion detection and DNA sequence analysis. Many hardware-based regular expression matching architectures are proposed for high performance matching. In particular, regular expression matching processors such as ReCPU have been proposed to solve the problem that full hardware solutions require re-synthesis of hardware whenever the patterns are updated. However, ReCPU has inefficient failure processing due to data backtracking. In this paper, we propose an efficient failure processing architecture for regular expression processor. The proposed architecture uses the failure bit included in instruction format and provides efficient failure processing by removing unnecessary data backtracking.

KEYWORDS

String matching, Regular expression, Application Specific Processor, Intrusion detection

1. INTRODUCTION

Text pattern matching is a computational intensive task, exploited in several applications such as intrusion detection and DNA sequence analysis. A regular expression (RE) [1] is an expression that represents a set of strings. In many applications, text patterns are represented by regular expressions. Regular expression matching has become a bottleneck in software-based solutions of many applications. To achieve high-speed regular expression matching, full hardware based solutions have been proposed [2,3,4]. These solutions generate non-deterministic finite automata (NFA) based HDL description for given regular expressions and implements them on FPGA. However, these approaches require regeneration of the HDL description and re-synthesis of FPGA implementation whenever the patterns are updated.

To avoid the problem of full hardware solution, a processor-based approach such as ReCPU [5,6], SMPU [7], and REMP [8] has been proposed. This approach does not require re-synthesis of the hardware and guarantees the flexibility. ReCPU is a special-purpose processor for regular expression matching. In ReCPU, a regular expression is mapped into a sequence of instructions, which are stored in the instruction memory. When an instruction fails to match, the instruction

Natarajan Meghanathan et al. (Eds) : ACSIT, FCST, ITCA, CoNeCo - 2015
pp. 01–06, 2014. © CS & IT-CSCP 2015 DOI : 10.5121/csit.2015.51201

sequence is restarted from the next address of data where the first match occurred. If one or more instructions are matching and then matching fails, data should be backtracked, which leads to inefficient failure processing. SMPU is another regular expression processor and it does not address the inefficient failure processing problem although it proposes the concept of dual exit instructions for efficient pipelining. We should solve the inefficient failure processing problem due to excessive data backtracking.

In this paper, we propose an efficient failure processing architecture for regular expression processor. The proposed architecture provides efficient failure processing by removing unnecessary data backtracking.

2. RELATED WORKS

In this section, we review previous regular expression processors and present their inefficient failure processing problem. ReCPU [5] is a processor based regular expression matching hardware. The regular expression operators that have been implemented in ReCPU are as follows: · (concatenation), * (zero or more repetition), + (one or more repetition), | (alternative), and parenthesis. In ReCPU, regular expression operators and characters are mapped into instruction opcodes and operands, respectively. The instruction format of ReCPU has multi-character operand as shown in Figure 1(a) for parallel comparison and ReCPU can perform more than one character comparison per clock cycle. In addition, the multi-character operand in an instruction is simultaneously compared with several consecutive input data starting by shifted positions as shown in Figure 1(b). The operators like * and + correspond to loop style instructions. To use the nested parentheses, a open parenthesis '(' is treat as a function call and a close parenthesis ')' , which is usually combined with an operator such as ')*', as a return.

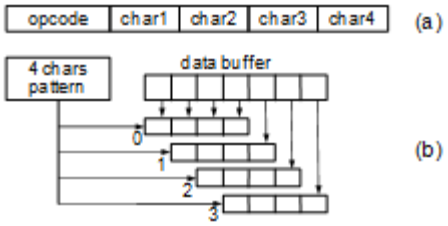

Figure 1. ReCPU (a) instruction format (b) comparator clusters

SMPU and REMP are regular expression processors improving the weakness of ReCPU. SMPU [7] proposes the concept of dual exit instructions for efficient pipelining and REMP [8] proposes an instruction set architecture for efficient repetitive operations.

Whenever one or more instruction are matching the input text and then the matching fails, ReCPU program is restarted from the next address of data where RE starts to match, as shown in Figure 2. Since data backtracking degrades the pattern matching performance, it is desirable to reduce unnecessary data backtracking.

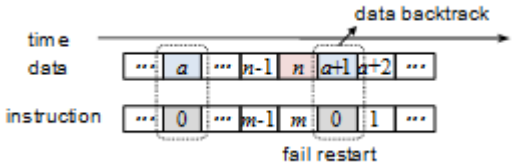

Figure 2. Restart operation of ReCPU

REMP [8] proposes an idea that a failure bit is included in the instruction format to solve data backtracking problem as shown in Figure 3. However, it does not propose the detailed implementation method of failure bit.

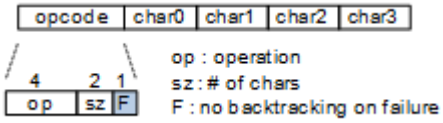

Figure 3. Instruction Format of REMP.

In this paper, therefore, we propose an efficient failure processing architecture and the implementation method utilizing the failure bit included in an instruction.

3. PROPOSED ARCHITECTURE

A regular expression may represent a set of strings. In a regular expression processor, regular expressions are mapped into a sequence of instructions. Each instruction in the instruction sequence is associated with a prefix sub-pattern of a regular expression. If an instruction succeeds to match current input data, it means that the input text is matching the corresponding prefix pattern of a regular expression.

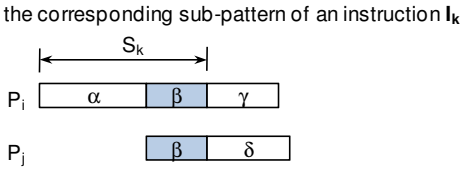

Figure 4. An instruction with failure bit F=1

Consider a regular expression $P_1 | P_2 | \ldots | P_n$. Let the corresponding sub-pattern of an instruction I_k be S_k. If there is a pattern P_j such that a suffix of S_k is a prefix of P_j , this means that matching operations of two patterns are overlapped, as shown in Figure 4. Otherwise, there is no overlapped matching. If an instruction has no overlapped matching, data backtracking is not necessary when the following instruction fails to match. Otherwise, data backtracking is required.

We can use this feature to reduce data backtracking as follows. For a instruction I_k, if there is no pattern P_j such that a suffix of S_k is a prefix of P_j , or there is no overlapped matching, a *failure bit* F is set to 1. Otherwise, F is set to 0. Setting a failure bit of an instruction should be performed by a compiler.

In the proposed regular expression processor, the next address of data where the first match occurred is stored as a backtracked data address (*bk_addr*). Without the failure bit information, when an instruction fails to match, data should be always backtracked to address *bk_addr*. However, we can use failure bit information in determining whether backtracking is required and adjusting the backtracked data address in order to reduce unnecessary data backtracking.

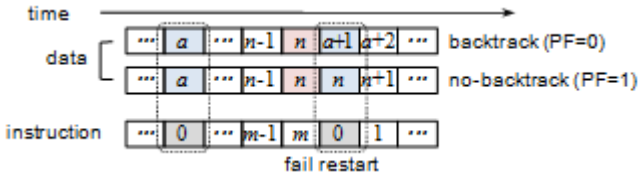

Figure 5. Restart operation of the proposed architecture

If an instruction succeeds to match, its failure bit F is stored as previous failure bit (PF). When an instruction fails to match and the instruction sequence is restarted, the data backtracking is determined according to PF value. If PF is 1, data backtracking is not required; If PF is 0, data backtracking is required. Figure 5 shows the restart operation of the proposed architecture. Thus, using failure bit information, we can remove unnecessary data backtracking.

If an instruction with F=1 succeeds to match, *bk_addr* is adjusted to the next data address since data backtracking is not required at current location. Adjusting the backtracked data address reduces the backtracking distance of data.

Example: Figure 6 shows a REMP [8] program for two patterns P_1 and P_2. It also shows the corresponding sub-pattern of each instruction. Multiple patterns are combined into one REMP program by using OR (for short patterns) or ORX (for long patterns) instructions. If ORX succeeds to match, the instruction sequence goes to the next instruction. Otherwise, the instruction sequence jumps to the instruction for an alternative pattern (in this example, CMP efxy), whose location is specified by a relative address. STAR, PLUS, and OPT instructions perform *, +, and ? operations for a short pattern, respectively. Figure 6 also shows failure bit values of instructions. Only two instructions in address 1 and 3 have F=0.

patterns : P_1 = abc(ef)*(st)+xyabz?a, P_2 = efxyzw		
program	**sub-pattern**	**F**
0 ORX abc, +7	**ab**c	1
1 STAR ef	abc(**ef**)*	0
2 PLUS st	abc(ef)*(st)+	1
3 CMP xyab	abc(ef)*(st)+xy**ab**	0
4 OPT z	abc(ef)*(st)+xyabz?	1
5 CMP k	abc(ef)*(st)+xyabz?k	1
6 MATCH 1	(match P1)	-
7 CMP efxy	**ef**xy	1
8 CMP zw	efxyzw	1
9 MATCH 2	(match P2)	-

Figure 6. REMP program and corresponding subpatterns

For an input string "gabcefstxyabzpabcef…", the REMP program executes as shown in Figure 7. When the instruction at address 5 fails to match and the instruction sequence is restarted, data is not backtracked and the instruction sequence is restarted from the current data since PF is 1. The start instruction of an instruction sequence compares four shifted data in parallel and non-start instructions match one of four shifted data specified by previous instruction.

input string: gabc efst xyab zpab cef …		
instr. sequence	**input text**	**PF / match result**
0 ORX abc, +7	gab/**abc**/bce/cef	0 / success
1 STAR ef	ef	1 / success
1 STAR ef	st	0 / fail – try alternative
2 PLUS st	st	0 / success
2 PLUS st	xy	1 / fail – try alternative
3 CMP xyab	xyab	1 / success
4 OPT z	z	0 / success
5 CMP k	p	**1** / fail - restart, **no backtrack**
0 ORX abc, +7	zpa/pab/**abc**/bce	1 / success
…	…	…

Figure 7. Instruction Execution Sequence and PF snapshot

4. EVALUATION

Table 1 shows advantages of the proposed architecture in comparison to previous regular expression processors such as ReCPU and SMPU. The proposed architecture using failure bit information reduces data backtracking. However, in ReCPU and SMPU, a data backtracking is always required whenever one or more instructions are matching and then matching fails. Moreover, data backtracking requires additional clock cycles since double word data should be fetched for instruction execution. The proposed architecture provides more efficient failure processing performance than previous processors by removing unnecessary data backtracking.

Table 1. Comparison between proposed architecture and previous processors

	previous processors (ReCPU …)	**proposed architecture**
data backtracking	always	in necessary cases
backward jump address	the next address of first match data	adjust it forward if needed

5. CONCLUSIONS

Regular expression matching is a computational intensive task, exploited in several applications such as intrusion detection and DNA sequence analysis. Regular expression matching processors such as ReCPU have been proposed to solve the problem that full hardware solutions require re-synthesis of hardware whenever the patterns are updated. However, ReCPU has inefficient failure processing due to excessive data backtracking. In this paper, we proposed an efficient failure processing architecture using the failure bit included in instruction format for regular expression processor. The proposed architecture provides efficient failure processing by removing unnecessary data backtracking and reducing data backtracking distance.

ACKNOWLEDGEMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0025467).

REFERENCES

[1] J. Friedl, Mastering Regular Expressions, 3rd ed., O'Reilly Media, August 2006..
[2] R. Sidhu and V. Prasanna, "Fast regular expression matching using FPGAs," in IEEE Symp. Field-Programmable Custom Computing Machines (FCCM'01), 2001.
[3] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of regular expression pattern matching circuits on FPGA," in Proc conf. Design, automation and test in Europe (DATE '06), 2006.
[4] J. C. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in IEEE Int. Conf Field Programmable Technology (FPT'06), 2006.
[5] M. Paolieri, I. Bonesana, M.Santambrogio, "ReCPU: a Parallel and Pipelined Architecture for Regular Expression Matching," in Proc. IFIP Int. Conf. VLSI-SoC, 2007.
[6] I. Bonesana, M. Paolieri, and M.Santambrogio, "An adaptable FPGA-based system for regular expression matching." In Proc. conf. Design, Automation and Test in Europe, (DATE'08), 2008.
[7] Q. Li, J. Li, J.Wang, B. Zhao, and Y. Qu, "A pipelined processor architecture for regular expression string matching," Microprocessors and Microsystems," vol. 36, no. 6, pp. 520–526, Aug. 2012
[8] B. Ahn, K. Lee, and S.K. Yun, "Regular expression matching processor supporting efficient repetive operations," Journal of KIISE: Computing Practices and Letters, vol. 19, no. 11, pp. 553–558, Nov. 2013 (in Korean).

AUTHORS

SangKyun Yun received the BS degree in electronics engineering from Seoul National University, Korea and the MS and Ph.D degrees in electrical engineering from KAIST, Korea. He is a professor in the Department of Computer and Telecom. Engineering, Yonsei University, Wonju, Korea.