Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System

Ioannis Sourdis and Dionisios Pnevmatikatos

Microprocessor and Hardware Laboratory,
Electronic and Computer Engineering Department,
Technical University of Crete, Chania, GR 73 100, Greece
{sourdis, pnevmati}@mhl.tuc.gr

Institute of Computer Science (ICS),
Foundation for Research and Technology-Hellas (FORTH),
Vasilika Vouton, Heraklion, GR 71110, Greece
pnevmati@ics.forth.gr

Abstract. Intrusion Detection Systems such as Snort scan incoming packets for evidence of security threats. The most computation-intensive part of these systems is a text search against hundreds of patterns, and must be performed at wire-speed. FPGAs are particularly well suited for this task and several such systems have been proposed. In this paper we expand on previous work, in order to achieve and exceed a processing bandwidth of 11Gbps. We employ a scalable, low-latency architecture, and use extensive fine-grain pipelining to tackle the fan-out, match, and encode bottlenecks and achieve operating frequencies in excess of 340MHz for fast Virtex devices. To increase throughput, we use multiple comparators and allow for parallel matching of multiple search strings. We evaluate the area and latency cost of our approach and find that the match cost per search pattern character is between 4 and 5 logic cells.

1 Introduction

The proliferation of Internet and networking applications, coupled with the widespread availability of system hacks and viruses have increased the need for network security. Firewalls have been used extensively to prevent access to systems from all but a few, well defined access points (ports), but firewalls cannot eliminate all security threats, nor can they detect attacks when they happen.

Network Intrusion Detection Systems (NIDS) attempt to detect such attempts by monitoring incoming traffic for suspicious contents. They use simple rules (or search patterns) to identify possible security threats, much like virus detection software, and report offending packets to the administrators for further actions. Snort is an open source NIDS that has been extensively used and studied in the literature [1–4]. Based on a rule database, Snort monitors network traffic and detect intrusion events. An example of a Snort rule is: `alert tcp any any`

`->192.168.1.0/24 111(content: "idc|3a3a|"; msg: "mountd access";)`
A rule contain fields that can specify a suspicious packet's protocol, IP address, Port, content and others. The "content" (idc|3a3a|) field contains the pattern that is to be matched, written in ascii, hex or mixed format, where hex parts are between vertical bar symbols "|". Patterns of Snort V1.9.x distribution contain between one and 107 characters.

NIDS rules may refer to the header as well as to the payload of a packet. Header rules check for equality (or range) in numerical fields and are straightforward to implement. More computation-intensive is the text search of the packet payload against hundreds of patterns that must be performed at wire-speed [3, 4]. FPGAs are very well suited for this task and many such systems have been proposed [5–7]. In this paper we expand on previous work, in order to achieve and exceed a processing bandwidth of 10Gbps, focusing on the string-matching module.

Most proposed FPGA-based NIDS systems use finite automata (either deterministic or non-deterministic) [8, 9, 6] to perform the text search. These approaches are employed mainly for their low cost, which is reported to be is between 1 and 1.5 logic elements per search pattern character. However, this cost increases when other system components are included. Also, the operation of finite automata is limited to one character per cycle operation. To achieve higher bandwidth researchers have proposed the use of packet-level parallelism [6], whereby multiple copies of the automata work on different packets at lower rate. This approach however may not work very well for IP networks due to variability of the packet size and the additional storage to hold these (possibly large) packets. Instead, in this work we employ full-width comparators for the search. Since all comparators work on the same input (one packet), it is straightforward to increase the processing bandwidth of the system by adding more resources (comparators) that operate in parallel. We evaluate the implementation cost of this approach and suggest ways to remedy the higher cost as compared to (N)DFA. We use extensive pipelining to achieve higher operating frequencies, and we address directly the fan-out of the packet to the multiple search engines, one of limiting factors reported in related work [2]. We employ a pipelined fan-out tree and achieve operating frequencies exceeding 245 MHz for VirtexE and 340 MHz for Virtex2 devices (post place & route results).

In the following section we present the architecture of our FPGA implementation, and in section 3 we evaluate the performance and cost of our proposed architecture. In section 4 we give an overview of FPGA-based string matching and compare our architecture against other proposed designs, and in section 5 we summarize our findings and present our conclusions.

2 Architecture of Pattern Matching Subsystem

The architecture of an FPGA-based NIDS system includes blocks that match header fields rules, and blocks that perform text match against the entire packet payload. Of the two, the computationally expensive module is the text match.

In this work we assume that it it relatively straightforward to implement the first module(s) at high speed since they involve a comparison of a few numerical fields only, and focus in making the pattern match module as fast as possible.

If the text match operates at one (input) character per cycle, the total throughput is limited by the operating frequency. To alleviate this bottleneck suggested using packet parallelism where multiple copies of the match module scan concurrently different packet data. However, due to the variable size of the IP packets, this approach may not offer the guaranteed processing bandwidth. Instead, we use discrete comparators to implement a CAM-like functionality. Since each of these comparators is independent, we can use multiple instances to search for a pattern in a wider datapath. A similar approach has been used in [7].

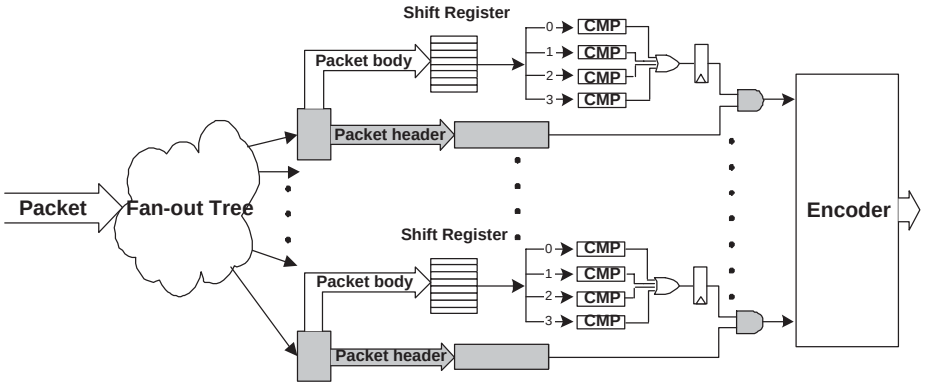

Fig. 1. Envisioned FPGA NIDS system: Packets arrive and are fan-out to the matching engines. N parallel comparators process N characters per cycle (four in this case), and the matching results are encoded to determine the action for this packet. Shaded is the header matching logic that involves numerical field matching.

The results of the system are (i) an indication that there was indeed a match, and (ii) the number of the rule that did match. Our architecture uses fine grain pipeline for all sub-modules: fan-out of packet data to comparators, the comparators themselves, and for the encoder of the matching rule. Furthermore to achieve higher processing throughput, we utilize N parallel comparators per search rule, so as to process N packet bytes at the same time. In the rest of this section we expand on our design in each of these sub-modules. The overall architecture we assume is depicted in Figure 1. In the rest of the paper we concentrate on the text match portion of the architecture, and omit the shaded part that performs the header numerical field matching. We believe that previous work in the literature have fully covered the efficient implementation of such functions [6, 7]. Next we describe the details of the three main sub-systems: the comparators, the encoder and the fan-out tree.

2.1 Pipelined Comparator

Our pipelined comparator is based on the observation that the minimum amount of logic in each pipeline stage can fit in a 4-input LUT and its corresponding register. This decision was made based on the structure of Xilinx CLBs, but the structure of recent Altera devices is very similar so our design should be applicable to Altera devices as well. In the resulting pipeline, the clock period is the sum of wire delay (routing) plus the delay of a single logic cell (one 4-input LUT + 1 flip-flop). The area overhead cost of this pipeline is zero since each logic cell used for combinational logic also includes a flip-flop. The only drawback of this deep pipeline is a longer total delay (in clock cycles) of the result. However, since the correct operation of NIDS systems does not depend heavily on the actual latency of the results, this is not a crucial restriction for our system architecture. In section 3 we evaluate the latency of our pipelines to show that indeed they are within reasonable limits.

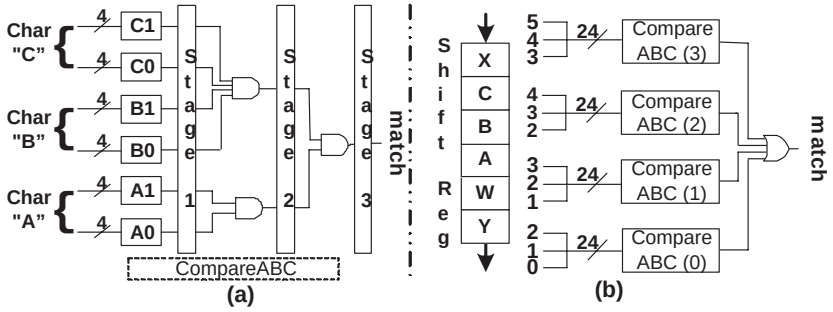

Fig. 2. (a) Pipelined comparator, which matches pattern "ABC". (b) Pipelined comparator, which matches pattern "ABC" starting at four different offsets.

Figure 2(a) shows a pipelined comparator that matches the pattern "ABC". In the first stage comparator matches the 6 half bytes of the incoming packet data, using six 4-input LUTs. In the following two stages the partial matches are AND-ed to produce the overall match signal. Figure 2(b) depicts the connection of four comparators that match the same pattern shifted by zero, one, two and three characters (indicated by the numerical suffix in the comparator label). Comparator comparator_ABC(0) checks bytes 0 to 2, comparator_ABC(1) checks bytes 1 to 3 and so on. Notice that the choice of four comparators is only indicative; in general we can use N comparators, allowing the processing of N bytes per cycle.

2.2 Pipelined Encoder

After the individual matches have been determined, the matching rule has to be encoded and reported to the rest of the system (most likely software). We

use a hierarchical pipelined encoder. In every stage, the combinational logic is described by at most 4-input, 1-output logic functions, which is permitted in our architecture.

The described encoder assumes that at most one match will occur in order to operate correctly (i.e. it is not a priority encoder). While in general multiple matches can occur in a single cycle, in practice we can determine by examining the search strings whether this situation can occur in practice. If all the search patterns have distinct suffixes, then we are ensured that we will not have multiple matches in a single cycle. However, this guarantee becomes more difficult as we increase the number of concurrent comparators. To this end we are currently working on a pipelined version of a priority encoder, which will be able to correctly handle any search string combination.

2.3 Packet data Fan-out

The fan-out delay is major slow-down factor that designers must take into account. While it involves no logic, signals must traverse long distances and potentially suffer significant latencies. To address this bottleneck we created a register tree to "feed" the comparators with the incoming data. The leaves of this tree are the shift registers that feed the comparators, while the intermediate nodes of the tree serve as buffers and pipeline registers at the same time. To determine the best fan-out factor for the tree, we experimented with the Xilinx tools, and we determined that for best results, the optimal fan-out factor changes from level to level. In our design we used small fan-out for the first tree levels and increase the fan-out in the later levels of the tree up to 15 in the last tree level. Intuitively, that is because the first levels of the tree feed large blocks and the distance between the fed nodes is much larger than in last levels. We also experimented and found that the optimal fan-out from the shift-registers is 16 (15 wires to feed comparators and 1 to the next register of shift register).

2.4 VHDL Generator

Deriving a VHDL representation starting from a Snort rule is very tedious; to handle tens of hundred of rules is not only tedious but extremely error prone. Since the architecture of our system is very regular, we developed a C program that automatically generates the desired VHDL representation directly from Snort pattern matching expressions, and we used a simple PERL script to extract all the patterns from a Snort rule file.

3 Performance Evaluation

The quality of an FPGA-based NIDS can be measured mainly using performance and area metrics. We measure performance in terms of operating frequency (to indicate the efficiency of our fine grain pipelining) and total throughput that can

be serviced, and we measure total area, as well as area cost per search pattern character.

We evaluate our proposed architecture we used three sets of rules. The first is an artificial set that cannot be optimized (i.e. at every position all search characters are distinct), that contains 10 rules matching 10 characters each. We also used the "web-attacks.rules" from the Snort distribution, a set of 47 rules to show performance and cost for a medium size rule set, and we used the entire set of web rules (a total of 210 rules) to test the scalability of our approach for large rule sets. The average search pattern length for these sets was 10.4 and 11.7 characters for the Web-attack and all the Web rules respectively.

We synthesized each of these rule sets using the Xilinx tools (ISE 4.2i) for several devices (the -*N* suffix indicates speed grade): Virtex 1000-6, VirtexE 1000-8, Virtex2 1000-5, VirtexE 2600-8 and Virtex2 6000-5. The structure of these devices is similar and the area cost of our design is expected (and turns out) to be almost identical for all devices, with the main difference in the performance.

The top portion of Table 1 summarizes our performance results. It lists the number of bits processed per cycle, the device, the achieved frequency and the corresponding throughput (in Gbps). It also lists the total area and the area required per search pattern character (in logic cells) of rules, the corresponding device utilization, as well as the dimensions of the rule set (number of rules and average size of the search patterns). For brevity we only list results for four parallel comparators, i.e. for processing 32 bits of data per cycle. The reported operating frequency gives a lower bound on the performance using a single (or fewer) comparators.

In the top portion of the table we can see that for our synthetic rule set (labeled 10x10) we are able to achieve throughput in excess of 6 Gbps for the simplest devices and over 12 Gbps for the advanced devices. For the actual Web attack rule set (labeled 47x10.4), we are able to sustain over 5 Gbps for the simplest Virtex 1000 device (at 171 MHz), and about 11 Gbps for a Virtex2 device (at 345 MHz). The performance with a VirtexE device is almost 8 Gbps at 245 MHz. Since the architecture allows a single logic cell at each pipeling stage, and the percentage of the wire delay in the critical path is around 50%, it is unlikely that these results can be improved significantly.

However the results for larger rule sets are more conservative. The complete set of web rules (labeled 210x11.7) operates at 204MHz and achieve a throughput of 6.5 Gbps on a VirtexE, and at 252MHz having 8 Gbps throughput on a Virtex2 device. Since the entire design is larger, the wiring latency contribution to the critical path has increased to 70% of the cycle time. The total throughput is still substantial, and can be improved by using more parallel comparators, or possibly by splitting the design in sub-modules that can be placed and routed in smaller area, minimizing the wire distances and hence latency.

In terms of implementation cost of our proposed architecture, we see that each of the search pattern characters costs between 15 and 20 logic cells depending on the rule set. However, this cost includes the four parallel comparators, so the

actual cost of each search pattern character is roughly 4-5 logic cells multiplied by N for N times larger throughput.

Table 1. Detailed comparison of string matching FPGA designs

Description	Input Bits/ c.c.	Device	Freq. MHz	Throughput (Gbps)	Logic Cells[1]	Logic Cells /char	Utilization	#Patterns × #Characters
Sourdis-Pnevmatikatos Discrete Comparators	32	Virtex 1000	193	6.176	1,728	17.28	7%	10 × 10
			171	5.472	8,132	16.64	33%	47 × 10.4
		VirtexE 1000	272	8.707	1,728	17.28	7%	10 × 10
			245	7.840	7,982	16.33	33%	47 × 10.4
		Virtex2 1000	396	12.672	1,728	16.86	16%	10 × 10
			344	11.008	8,132	16.64	80%	47 × 10.4
		VirtexE 2600	204	6.524	47,686	19.40	94%	210 × 11.7
		Virtex2 6000	252	8.064	47,686	19.40	71%	210 × 11.7
Sidhu et al.[9] NFAs/Reg. Expression	8	Virtex 100	93.5	0.748	280	∼31	11%	(1×) 9[4]
			57.5	0.460	1,920	∼66	80%	(1×) 29[4]
Franklin et al.[8] Regular Expressions	8	Virtex 1000	31	0.248	20,618	2.57	83%	8,003[5]
			99	0.792	314	3.17	1%	99
			63.5	0.508	1726	3.41	7%	506
		VirtexE 2000	50	0.400	20,618	2.57	53%	8,003
			127	1.008	314	3.17	1%	99
			86	0.686	1726	3.41	4%	506
Lockwood[6] DFAs 4 Parallel FSMs on different Packets	32	VirtexE 2000	37	1.184	4,067[2]	16.27	22%[2]	34 × 8[3]
Lockwood[10] FSM+counter	32	VirtexE 1000	119	3.808	98	8.9	0.4%	1 x 11
Gokhale et al.[5] Discrete Comparators	32	VirtexE 1000	68	2.176	9,722	15.2	39%	32 × 20
Young Cho et al.[7] Discrete Comparators	32	Altera EP20K	90	2.880	N/A	10	N/A	N/A

We compute the latency of our design taking into account the three components of our pipeline: fan-out, match, encode. Since the branching factor is not fixed in the fan-out tree, we cannot offer a closed form for the number of stages.

[1] Two *Logic Cells* form one *Slice*, and two Slices form one *CLB* (4 Logic Cells).

[2] These results does not includes the cost/area of infrastructure and protocol wrappers

[3] 34 regular expressions,with 8 characters on average, (about 250 character)

[4] One regular Expression of the form (a | b)*a(a | b)^k for k = 8 and 28. Because of the * operator the regular expression can match more than 9 or 29 characters.

[5] Sizes refer to Non-meta characters and are roughly equivalent to 800, 10, and 50 patterns of 10 characters each.

The pipeline depths for the designs we have implemented are: $3 + 5 + 4 = 12$ for the Synth10 rule set, $3 + 6 + 5 = 14$ for the Web Attacks rule set, and $5 + 7 + 7 = 19$ for the Web-all rule set. For 1,000 patterns and pattern lengths of 128 characters, we estimate the total delay of the system to be between 20 and 25 clock cycles.

We also evaluated resource sharing to reduce the implementation cost. We sorted the 47 web attack rules, and we allowed two adjacent patterns to share comparator i if their i^{th} characters were the same, and found that the number of logic cells required to implement the system was reduced by about 30%. Due to space limitations, we do not expand on this option in detail. However, it is a very promising approach to reduce the implementation cost, and allow even more rules to be packed in a given device.

4 Comparison with previous work

In this section we attempt a fair comparison with previous reported research. While we have done our best to report these results with the most objective way, we caution the reader that this task is difficult since each system has its own assumptions and parameters, occasionally in ways that are hard to quantify.

One of the first attempts in string matching using FPGAs, presented in 1993 by Pryor, Thistle and Shirazi [11]. Their algorithm, implemented on Splash 2, succeeded to perform a dictionary search, without case sensitivity patterns, that consisted of English alphabet characters (26 characters). Pryor et al managed to achieve great performance and perform a low overhead AND-reduction of the match indicators using hashing.

Sidhu and Prassanna [9] used Regular Expressions and Nondeterministic Finite Automata (NFAs) for finding matches to a given regular expression. They focused in minimizing the space -$O(n^2)$- required to perform the matching, and their automata matched 1 text character per clock cycle. For a single regular expression, the constructed NFAs and FPGA circuit was able to process each text character in 17.42-10.70ns (57.5-93.5 MHz) using a Virtex XCV100 FPGA.

Franklin, Carver and Hutchings [8] also used regular expressions to describe patterns. The operating frequency of the synthesized modules was about 30-100 MHz on a Virtex XCV1000 and 50-127 MHz on a Virtex XCV2000E, and in the order of 63.5 MHz and 86 MHz respectively on XCV1000 and XCV2000E for a few tens of rules.

Lockwood used the Field Programmable Port Extender (FPX) platform, to perform string matching. They used regular expressions (DFAs) and were able to achieve operation at 37 MHz on a Virtex XCV2000E [6]. Lockwood also implemented a sample application on FPX using a single regular expression and were able to achieve operation at 119 MHz on a Virtex V1000E-7 device [10].

Gokhale, et al [5] using CAM to implement Snort rules NIDS on a Virtex XCV1000E. Their hardware runs at 68MHz with 32-bit data every clock cycle, giving a throughput of 2.2 Gbps, and reported a 25-fold improvement on the

speed of Snort v1.8 on a 733MHz PIII and an almost 9-fold improvement on a 1 GHz PowerPC G4.

Closer to our work is the recent work by Cho, Navab and Mangione-Smith [7]. They designed a deep packet filtering firewall on a FPGA and automatically translated each pattern-matching component into structural VHDL. The content pattern match unit micro-architecture used 4 parallel comparators for every pattern so that the system advances 4 bytes of input packet every clock cycle. The design implemented in an Altera EP20K device runs at 90MHz, achieving 2.88 Gbps throughput. They require about 10 logic cells per search pattern character. However, they do not include the fan-out logic that we have, and do not encode the matching rule. Instead they just OR all the match signals to indicate that some rule matched.

The results of these works are summarized in the bottom portion of Table 1, and we can see that most previous works implement a few tens of rules at most, and achieve throughput less than 4 Gbps. Our architecture on the same or equivalent devices achieves roughly twice the operating frequency and throughput. In terms of best performance, we achieve 3.3 times better processing throughput compared with the fastest published design which implements a single search pattern. Our 210-rule implementation achieves at least a 70% improvement in throughput compared to the fastest existing implementation.

5 Conclusions and Future Work

We have presented an architecture for Snort rule match in FPGAs. We propose the use of extensive fine grain pipelining in order to achieve high operating frequencies, and parallel comparators to increase the processing throughput. This combination proves very successful, and the throughput of our design exceeded 11 Gbps for about 50 Snort rules. These results offer a distinct step forward compared to previously published research. If latency is not critical to the application, fine grain pipelining is very attractive in FPGA-based designs: every logic cell contains one LUT and one Flip-Flop, hence the pipeline area overhead is zero. The current collection of Snort rules contains less than 1500 patterns, with an average size of 12.6 characters. Using the area cost as computed earlier, we need about 3 devices of 120,000 logic cells to include the entire Snort pattern matching, and about 4 devices to include the entire snort rule set including header matching. These calculations do not include area optimizations, which can lead to further significant improvements.

Throughout this paper we used four parallel comparators. However, a different level of parallelism can also be used depending on the bandwidth demands. Reducing the processing width leads to a smaller, possibly higher frequency design, while increasing the processing width leads to a bigger and probably lower frequency design. Throughput depends on both frequency and processing width, so we need to seek for the cost effective tradeoff of these two factors.

Despite the significant body of research in this area, there are still improvements that we can use to seek better solutions. In our immediate goals is to

use the hierarchical decomposition of large rule set designs, and attempt to use the multiple clock domains. The idea is to use a slow clock to drive long wide busses to distribute data and a fast clock for local processing that only uses local wiring. The target would be to retain the frequency advantage of our medium-sized design (47 rules) for a much larger rule set. All the devices we used already support multiple clock domains, and with proper placement and routing tool support this approach will also be quicker to implement: each of the modules can be placed and routed locally one after the other, reducing the memory and processing requirements for placement and routing.

Furthermore, future devices offer significant speed improvements, and it would be interesting to see whether the fine grain pipelining will be as effective for these devices (such as the Virtex 2 Pro) as it was for the devices we used in this work.

Acknowledgments

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union.

References

1. SNORT official web site: (http://www.snort.org)
2. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of LISA'99: 13th Administration Conference. (1999) Seattle Washington, USA.
3. Desai, N.: Increasing performance in high speed NIDS. In: www.linuxsecurity.com. (2002)
4. Coit, C.J., Staniford, S., McAlerney, J.: Towards faster string matching for intrusion detection or exceeding the speed of snort. In: DISCEXII, DAPRA Information Survivability conference and Exposition. (2001) Anaheim, California, USA.
5. Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S., Hogsett, V.: Granidt: Towards gigabit rate network intrusion detection technology. In: Proceedings of 12th International Conference on Field Programmable Logic and Applications. (2002) France.
6. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines. (2003) Napa, CA, USA.
7. Young H. Cho, S.N., Mangione-Smith, W.: Specialized hardware for deep network packet filtering. In: Proceedings of 12th International Conference on Field Programmable Logic and Applications. (2002) France.
8. Franklin, R., Carver, D., Hutchings, B.: Assisting network intrusion detection with reconfigurable hardware. In: IEEE Symposium on Field-Programmable Custom Computing Machines. (2002)
9. Sidhu, R., Prasanna, V.K.: Fast regular expression matching using fpgas. In: IEEE Symposium on Field-Programmable Custom Computing Machines. (2001) Rohnert Park, CA, USA.
10. Lockwood, J.W.: An open platform for development of network processing modules in reconfigurable hardware. In: IEC DesignCon '01. (2001) Santa Clara, CA, USA.
11. Pryor, D.V., Thistle, M.R., Shirazi, N.: Text searching on splash 2. In: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines. (1993) 172–177