# Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching

**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

*In this paper we advocate the use of pre-decoding for CAM-based pattern matching. We implement an FPGA based sub-system for NIDS (Snort) pattern matching using a combination of techniques. First, we reduce the area cost of character matching using (i) character pre-decoding before they are compared in the CAM line, and (ii) efficient shift register implementation using the SRL16 Xilinx cell. Second we achieve high operating frequencies by (iii) using fine grain pipelining for faster circuits and (iv) decoupling the data distribution network from the processing components. Our results show that for matching more than 18,000 characters (the entire SNORT rule set) our implementation requires an area cost of less than 1.1 logic cells per matched character, achieving an operating frequency of about 375 MHz (3 Gbps) on a Virtex2 device. When using quad parallelism to increase the matching throughput, the area cost of a single matched character is reduced to less than one logic cell for a throughput of almost 10 Gbps.*

## 1. Introduction

High speed and always-on network access is becoming commonplace around the world, creating a demand for increased network security. Network Intrusion Detection Systems (NIDS) such as SNORT [12, 14] attempt to detect and prevent attacks from the network using pattern-matching rules in a way similar to anti-virus software. These systems must operate at line (wire) speed so that they do not become a bottleneck to the system's performance [4, 5]. Network Intrusion Detection Systems running in general purpose processors can only serve up to a few hundred Mbps throughput. For example, Antonatos et al. achieved no more than 350 Mbps for more than 14,000 bytes matching, using a Pentium 4 processor running at 1.7GHz running Linux [1]. Measurements on Snort show that 31% of total processing and 80% in the case of Web-intensive traffic is due to string matching [6]. Therefore, string matching can be considered as the most computational intensive part of an NIDS and in our implementation we focus on payload matching.

FPGA-based platforms can exploit the fact that the NIDS rules change relatively infrequently, and use reconfiguration to reduce implementation cost. In addition, FPGA-based systems can exploit parallelism in order to achieve satisfactory processing throughput.

Several architectures have been proposed for FPGA-based NIDS, using N/DFAs [13, 9, 10, 7, 3], CAM-based solutions [8], and CAM-based using discrete comparators [2, 15]. The most common approach is the use of regular expressions (NFAs/DFAs), which results in designs with low cost, but at a modest throughput. The basic idea is to generate regular expressions for every pattern or group of patterns, and implement them with N/DFA.

The use of parallelism (processing multiple bytes or characters per cycle) in general is difficult in finite-automata implementations that are built with the implicit assumption that the input is checked one byte at a time. One proposed solution to this problem is the usage of packet-level parallelism, where multiple pattern matching subsystems operating in parallel can process more than one packets [10]. Finally, finite automata are usually restricted in their operating frequency by the amount of combinational logic for state transitions. In many cases the equations are complex,

resulting in multilevel implementations even with FPGA 4-to-1 LUTs.

In a previous paper [15] we have shown that a CAM implemented using discrete comparators for pattern matching has several advantages: (i) it is simple and regular, (ii) it allows for fine grain pipelining and high operating frequencies, and (iii) it is straightforward to use multiple comparators in order to process multiple input bytes per cycle. The main drawback of this approach is its area cost which is around 4-5 logic cells per search pattern character including all overheads. To reduce the cost we have proposed sharing the result of comparators when the same character was searched for in two different patterns but at the same location. For example in search strings “AB” and “AC” we could use only one comparator for “A” instead of two (Figure 1).

In this paper we extend this observation and we combine it with decoding characters before they are matched by the comparators to form a Decoded CAM (DCAM). Decoding is implied in several finite automata implementations but was only recently explored in a short paper by Clark and Schimmel [3]. We extend Clark’s and our earlier ideas in the following ways: (i) we explore them in a discrete comparator CAM instead of finite automata, (ii) we use an efficient shift register implementation (the Xilinx SRL16 shift register) to replace our discrete flip-flop based shift registers, and (iii) we use a grouping algorithm to arrange search patterns in smaller groups so that the fanout within each group is reasonable, and the size of the decoders is smaller. The combination of these techniques offers operating speeds exceeding 335MHz, while the area cost is about one logic cell per search pattern character. We also explore the use of parallelism to further increase the throughput, and find that for processing $P$ bytes per cycle the area cost is about $P * 0.9$ logic cells per search pattern character, while the operating frequency does not drop below 300MHz.

The rest of the paper is organized as follows: In sections 2 and 3 we describe the architecture of the Decoded CAM and we present implementation results in terms of area cost and performance. Then in section 4 we present the related work and present some qualitative and quantitative comparison with our results. Finally, in section 5 we preset the conclusions of this work and discuss future extensions.

## 2. Architecture of the Pattern Matching System

The overall organization of a pattern matching systems is simple: a single input supplies the input stream of characters, and the output is an indication that a match did occur, plus the identifier of the matching rule. The details of this system (e.g. the encoder of matching signals, etc) are simple, we concentrate on the actual pattern matching block.

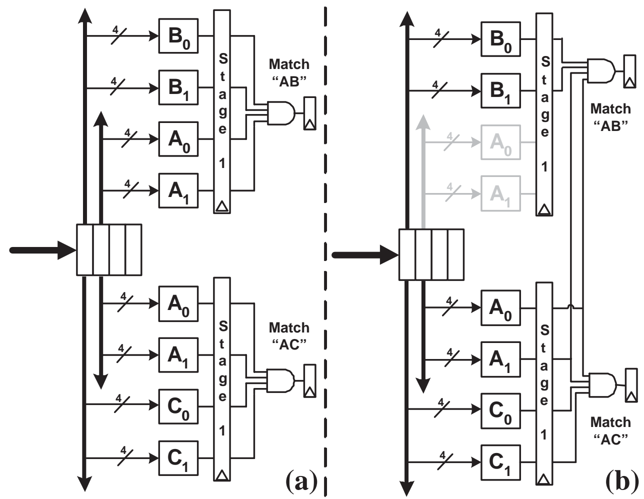


**Figure 1. Basic CAM Comparator structure and optimization. Part (a) depicts the straightforward implementation where a shift register holds the last $N$ characters of the input stream. Each character is compared against the desired value (in two nibbles to fit in FPGA LUTs) and all the partial matches are combined with an AND gate to produce the final match result. Part (b) depicts an optimization where the match “A” signals are shared across the two search strings “AB” and “AC” to save area.**

In our previous work [15], we assumed the simple organization depicted in Figure 1(a). The input stream is inserted in a shift register, and the individual entries are fan-out to the pattern comparators. So, to search for strings “AB” and “AC”, we have two comparators fed from the first two position of the shift register. Figure 1(a) reflects the FPGA implementation where each 8-bit comparator is broken down to two 4-bit comparators each of which fits in one LUT. This implementation is simple and regular, and with proper use of pipelining can achieve very high operating frequencies. Its drawback is the high area cost. To remedy this cost, in our previous work we had suggested *sharing* the character comparators for strings with “similarities”. This is shown in Figure 1(b) where the result of a single comparator for character A is shared between the two search strings “AB” and “AC”. Our preliminary results at the time indicated an area improvement of at least 30%.

The DCAM architecture builds on this idea extending it further by the following observation: instead of keeping a window of input characters in the shift register each of which is compared against search patterns, we can first test for equality of the input for the desired characters, and then

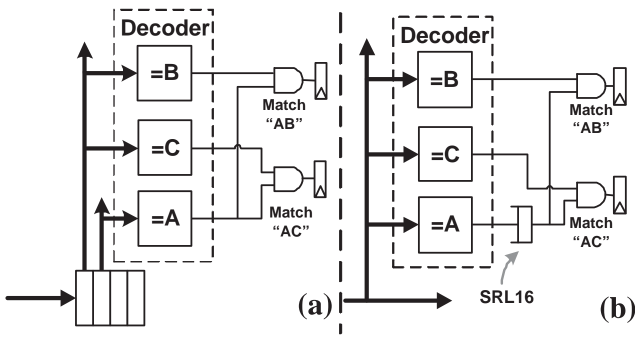


**Figure 2. Comparator Optimization: starting from the shared comparator implementation of part (a) we can move the comparators *before* the shift register, and delay the matching signals properly to achieve the correct result. Note that the shift register is 8-bit wide in part (a), and 1-bit wide part (b).**

delay the partial matching signals. These two approaches are compared in Figure 2. Part (a) corresponds to our earlier design with the LUT details abstracted away in the equality boxes. Part (b) shows how we can first test for equality of the three distinct characters of interest and then delay the matching of character A to obtain the complete match for strings "AB" and "AC". This approach achieves not only the sharing of the equality logic for character A, but also transforms the 8-bit wide shift register used in part (a) into possibly multiple single bit shift register for the equality result(s). Hence, if we can exploit this advantage, the potential for area savings is significant.

One of the possible shortfalls of the DCAM architecture is that the number one single bit shift registers is proportional to the length of search patterns. Figure 3 illustrates this point: to match a string of length four characters, we (i) need to test equality for these four characters (in the dashed "decoder" block), and to delay the matching of the first character by three cycles, the matching of the second character by two cycles, and so on, for the width of the search pattern. In total, the number of storage elements required in this approach is $L*(L-1)/2$ for a string of length L. For many and long search patterns, this number can exceed the number of bits in the character shift register used in the original CAM design. To our advantage though is the fact that these shift registers are true FIFOs with one input and one output, in contrast with the shift registers in the simple design in which each entry in the shift register is fan-out to comparators.

To tackle this possible obstacle, we use two techniques. First, we reduce the number of shift registers by sharing their outputs whenever the same character is used in the same position in multiple search patterns. This technique is similar to the comparator sharing depicted in figure 1(b). Second, we use the SRL16 optimized implementation of shift register (described in more detail in the following subsection) that is available in recent Xilinx devices that uses a single logic cell for a shift register of any width up to 16. Together these two optimizations lead to significant area savings as we will show in the evaluation section. In the following subsections we describe the techniques we used to achieve an efficient implementation of the DCAM architecture.

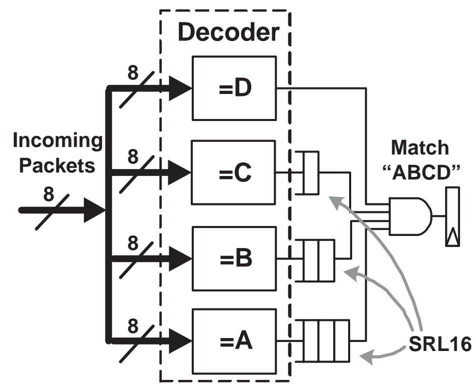


**Figure 3. Details of Pre-decoded CAM matching: four comparators provide the equality signals for characters A, B, C, and D. To match the string "ABCD" we have to remember the matching of character A 3 cycles ago, the matching of B two cycles ago, etc, until the final character is matched in the current cycle. This is achieved with the shift registers of width 3, 2, ... at the proper match lines.**

## 2.1. Xilinx SRL16 shift register

The Xilinx SRL16 cell is a shift register with a programmable width up to 16-bit. It uses the $16\times1$ storage space that implements the Lookup Table, and as a result is implemented with a single Logic Cell. The four inputs that usually are the LUT's inputs are used to determine the width of the shift register. While the SRL16 provides synchronous output, for timing reasons it is beneficial to add a discrete flip-flop to its output. This configuration provides a better timing solution and simplifies the design [16]. In addition it allows a single logic cell to implement a shift register with width of up to 17 bits. Figure 4 shows the detailed block diagram of a logic cell configured as a SRL16 shift register.

In our design, we use one SRL16 cell at the output of each equality test (i.e. for each distinct character) and for each location (offset) where this character appears in a

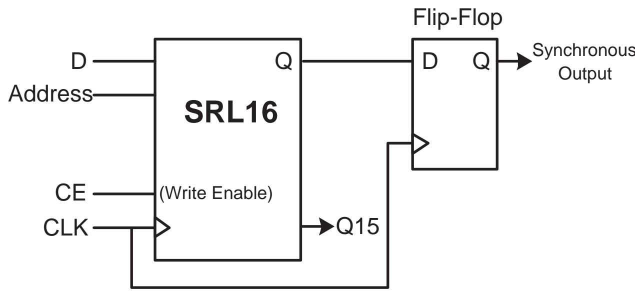


**Figure 4. Xilinx Logic Cell SRL16 structure. Fully synchronous Shift Register.**

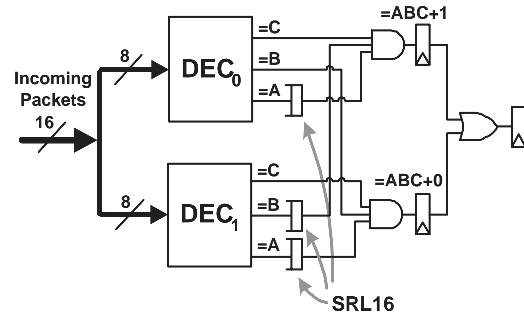


**Figure 5. Decoded CAM processing 2 characters per cycle: Two sets of comparators provide matching information for the two character positions. Their results have to be properly delayed to ensure matching of the string ABC starting at an offset of either 0 or 1 within the 16-bit input word.**

search pattern. However, we share the output of the SRL16 cells for search pattern characters that appear in the same position in multiple search strings. To avoid fan-out problems we replicate SRL16 cells so that the fanout does not exceed 16. This is based on an experimental evaluation we performed on Xilinx devices that showed that when the fanout exceeds 16 the operating frequency drops significantly.

## 2.2 Techniques to Increase Performance

In order to achieve better performance we used techniques to improve the operating speed, as well as the throughput of our DCAM implementation. To achieve high operating frequency, we use extensive fine grain pipeline in a manner similar to our earlier work. In fact, each of our pipeline stages consists of a single processing LUT and a pipeline register in its output. In this way the operating frequency is limited by the latency of a single logic cell and the interconnection wires. To keep interconnection wires short, we addressed the long data distribution wires that usually have large fan-out by providing a pipelined fan-out tree. More details on these two techniques can be found in [15].

As alluded earlier, to increase the processing throughput of a DCAM we can use parallelism. Similar to our previous work we can widen the distribution paths by a factor of $P$ providing $P$ copies of comparators(decoders) and the corresponding matching gates. Figure 5 illustrates this point for $P = 2$. The single string ABC is searched for starting at offset 0 or 1 within the 2-byte wide input stream, and the two partial results are OR-ed to provide the final match signal. This technique can be used for any value of $P$, not restricted to powers of two. Note also that the decoders provide the equality signals *only* for the distinct characters in the $N$ search patterns. Therefore we can reduce the required area (and the fanout of the input lines) if the patterns are "similar". In the next subsection we exploit this behavior to further reduce the area cost of DCAMs.

## 2.3. Search Pattern Partitioning

In the DCAM implementation we use partitioning to achieve better performance and area density. In terms of performance, a limiting factor to the scaling of an implementation to a large number of search patterns is the fanout and the length of the interconnections. For example, if we consider a set of search patterns with 10,000 uniformly distributed characters, we have an average fanout of 40 for each of the decoders outputs. Furthermore, the distance between all the decoders outputs and the equality checking AND gates will be significant.

If we partition the entire set of search patterns in smaller groups, we can implement the entire fanout-decode-match logic for each of these groups in a much smaller area, reducing the average length of the wires. This reduction in the wire length though comes at the cost of multiple decoders. With grouping, we need to decode a character for each of the group in which they appear, increasing the area cost. On the other hand, the smaller groups may require smaller decoders, if the number of distinct characters in the group is small. Hence, if we group together search patterns with more similarities we can reclaim some of the multi-decoder overhead.

In the partitioned design, each of the partitions will have a structure similar to the one depicted in Figure 6. The multiple groups will be fed data through a fanout tree, and all the individual matching results will be combined to produce the final matching output.

Each of the partitions will be relatively small, and hence can operate at a high frequency. However, for large designs, the fanout of the input stream much traverse long distances.

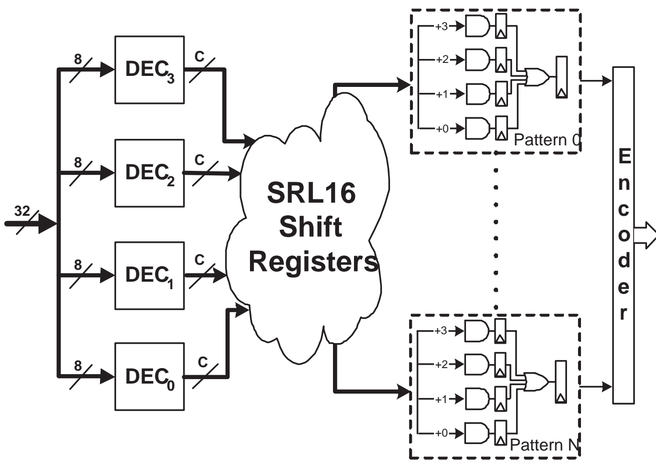


**Figure 6. The structure of an $N$-search pattern module with parallelism $P = 4$. Each of the $P$ copies of the decoder generates the equality signals for $C$ characters, where $C$ is the number of distinct characters that appear in the $N$ search strings. A shared network of SRL16 shift registers provides the results in the desired timing, and $P$ AND gates provide the match signals for each search pattern.**

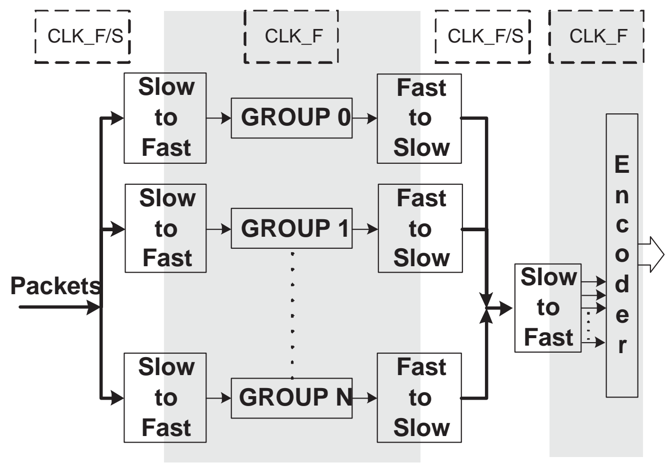


**Figure 7. DCAM with Multiple Clock Domains. Slow but wide busses distribute input data over large distances to the multiple search matching groups. These groups operate at higher clock rates to produce results faster.**

In our designs we have found that these long wires limit the frequency for the entire design. To tackle this bottleneck we used multiple clocks: one slow clock to distribute the data across long distances over wide busses, and a fast clock for the smaller and faster partitioned matching function. The idea is shown in Figure 7.

Experimenting with various partition sizes and slow-to-fast clock speed ratios we found that reasonable sizes for groups is between 64 and 256 search patterns, while a slow clock of twice the period is slow enough for our designs.

## 2.4. Pattern Partitioning Algorithm

To identify which search patterns should be included in a group we have to determine the relative cost of the various different possible groupings. The goal of the partitioning algorithm is (i) to minimize the total number of distinct characters that need to be decoded for each group, and (ii) to maximize the number of characters that appear in the same position in multiple of search patterns of the group (in order to share the shift registers). For this work we have implemented a simple, greedy algorithm that partitions iteratively the set of search strings according to the following steps:

1. First we create an array with one entry for each search pattern. Each array entry contains the set of distinct characters in the search string.

2. Starting with a number of empty partitions or groups, we first perform a step of initial assignment of search patterns to obtain a "seed" pattern in each group of the different groups.

3. Then we use an iterative method: for each group we select an unassigned search pattern so that the cost of adding it to the group is the least among the unassigned patterns. The cost is computed by finding the set difference between the set of characters used already by the group and the set of characters in the search pattern under consideration. We iterate among all groups and all search patterns until all the patterns have been assigned to a group.

Our algorithm implements a simple heuristic and does not guarantee an optimal partitioning of the search patterns. However, we have compared it with a straightforward approach of just sorting the search patterns, and we found that using the group identified by our algorithm the area cost was about 5% smaller and 5% faster than the one using partitioning based on sorted search patterns. Our algorithm is more efficient in minimizing the number of shift registers requiring 9% fewer shift registers than the sorting the search patterns. For the entire SNORT rule set and using 24 groups, our algorithm produced groups that contain an average of 54 distinct search characters each. Therefore each of the decoders is significantly smaller that a full 8-to-256 decoder.

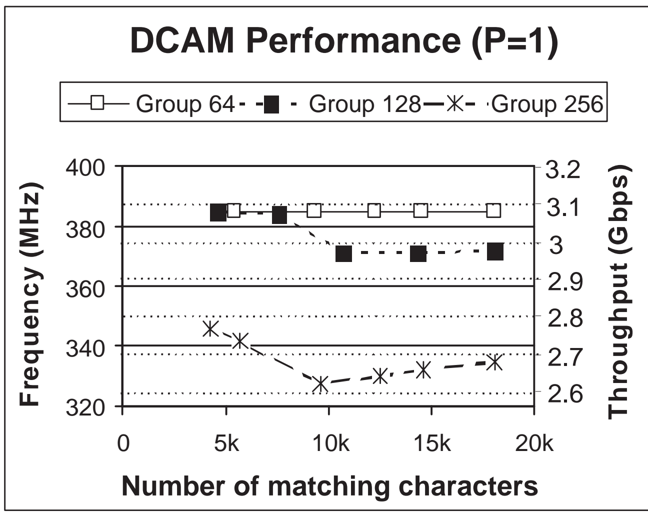


**Figure 8. DCAM Performance in terms of operating frequency and throughput for the group sizes of 64, 128, and 256 rules.**

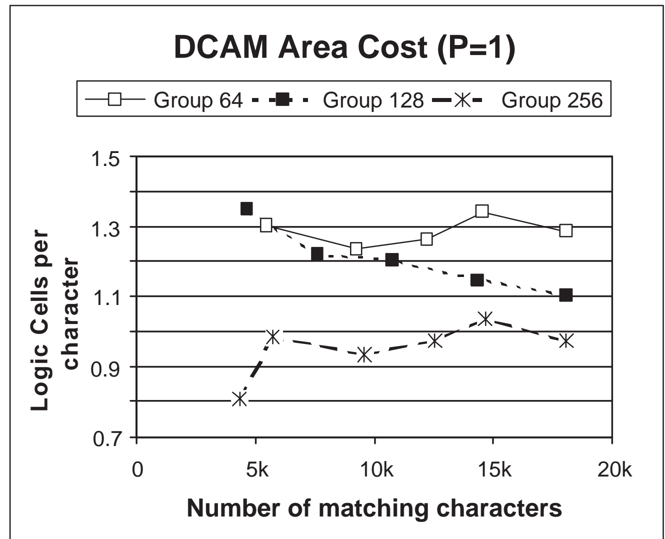


**Figure 9. DCAM Area cost in terms of operating frequency and throughput for the group sizes of 64, 128, and 256 rules.**

## 3. Evaluation

We evaluate the efficiency of our DCAM architecture and implementation using two main metrics: performance in terms of operating frequency and processing throughput, and area cost in terms of required FPGA logic cells and slices. We implemented the DCAM architecture on Xilinx Virtex2 devices at -6 speed grade, and varied the exact device model to keep the device utilization above 70%. We generated the VHDL description for the implementation automatically from the SNORT rule set according to the results of our partitioning algorithm. To evaluate the impact of partitioning on our proposed architecture, we considered three different group sizes: 64, 128, and 256 rules per group. Experimentally we have found that groups smaller than 64 or larger than 256 rules are inefficient and that the range 64-256 is sufficient to explore grouping efficiency. We used the official SNORT rule set [12] which consists of a total of about 1,500 rules and a corresponding 18,000 characters. Finally, we also considered the use of parallelism to increase throughput and we implemented a DCAM the process 4 bytes per cycle ($P = 4$).

### 3.1. DCAM Performance and Area Evaluation

Our first step is to evaluate the basic performance and cost of DCAMs. Figure 8 plots the performance both in terms of operating frequency, as well as in processing throughput (Gbps) for the three group sizes (64, 128, 256 rules per group) and for rule sets with sizes between 4,000 and 18,000 total characters. We can see that all the different designs achieve operating frequencies between 335 and 385MHz. This corresponds to a processing bandwidth between 2.7 and 3.1 Gbps. From our results we can draw two general trends for group size. The first is that smaller group sizes are more insensitive to the total design size (the plot for group size of 64 rules is almost flat). The second is that when the group size approaches 256 the performance deteriorates, indicating that optimal group sizes will be in the 64-128 range.

We measured area cost and plot the number of logic cells needed for each search pattern character in Figure 9. Unlike performance, the effect of group size on the area cost is more pronounced. As expected, larger group sizes result in smaller area cost due to the smaller replication of comparators in the different groups. Similar to performance, the area cost sensitivity to total rule set size increases with group size. In all, the area cost for the entire SNORT rule set is about 1.28, 1.1 and 0.97 logic cells per search pattern character for group sizes of 64, 128 and 256 rules respectively. This cost includes all overhead of fan-out of the input data, as well as of the output encoder and the slow-to-fast and fast-to-slow converters, and is comparable to (or even better than) area costs of designs based on finite automata. While smaller group sizes offer the best performance, it appears that if we also take into account the area cost, our medium group size (128) becomes more attractive.

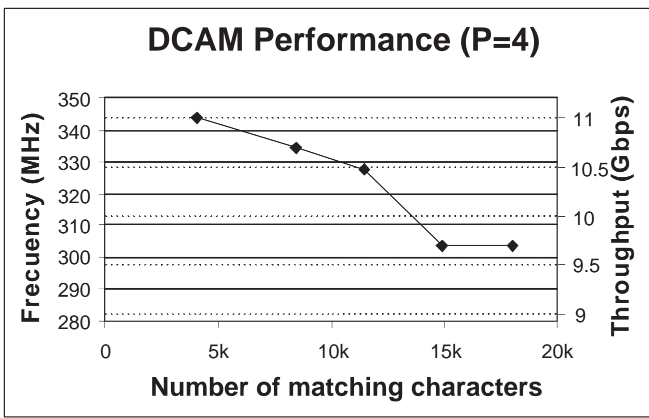


**Figure 10. DCAM Performance (operating frequency and throughput) for 4-byte per cycle processing (P = 4) and group size of 64 rules.**

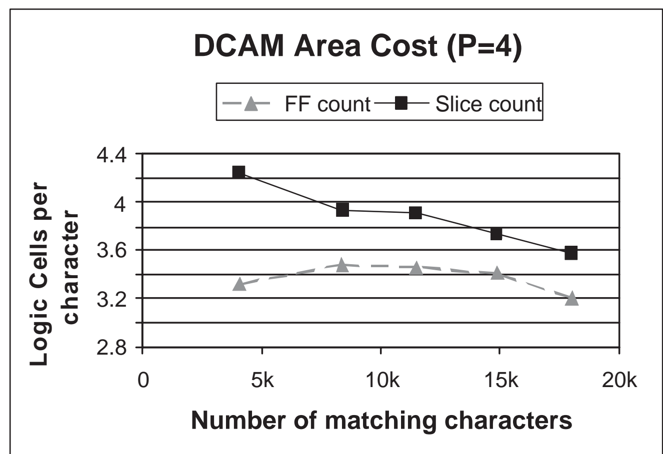


**Figure 11. DCAM Area cost for 4-byte per cycle processing (P = 4) and group size of 64 rules. The dashed line is computed counting flip-flop instead of slice utilization.**

### 3.2. Designs with parallelism

As we described earlier, we can utilize parallelism to increase the processing throughput of a DCAM. In this subsection we evaluate the performance and cost of a DCAM of four parallel matching structures, i.e. a DCAM that processes 4 input bytes per cycle. We used a group set of 64 rules based on the results of Figure 8, and the observation that since each group include four comparators, it would be roughly equivalent in size to a group of size 256 of the single-byte processing equivalent design. Figure 10 plots the performance in terms of operating frequency and corresponding processing throughput (Gbps) for rule set sizes ranging from 4,000 to 18,000 search pattern characters. As expected the performance drops as the total design size increases, and the operating frequency is lower than for DCAMs processing a single character per cycle. For medium sized designs, a DCAM can operate at around 330MHz, while for our largest rule set (the entire SNORT rule set) the DCAM operates at 300MHz. These frequencies correspond to processing throughput of 10.5 and 9.7 Gbps respectively.

The area cost per search pattern character is shown in Figure 11 (solid line). Depending on the design size, the number of required logic cells per search pattern character is between 3.9 and 3.6. Since each search pattern character is searched for in four different locations (within the 4-byte input word), the actual area cost of matching one character at one location is less that one for the entire SNORT rule set, smaller than for our earlier reported cost.

The area cost report of the Xilinx tools report area cost (device utilization) in terms of occupied slices. However, one slice contains 2 logic cells, and is reported to be occupied even if one of the two is in use. Hence it is possible that the exact cost will actually be smaller if the unused logic cells in slices can be used for other logic. Our DCAM design always uses a flip-flop after each LUT, and it uses flip-flops without corresponding logic for signal fan-out. Hence, it is possible to measure the exact DCAM cost by counting flip-flops, and fortunately the P&R tools report the flip-flop utilization. We used this number to compute the area cost in terms of flip-flops, and plotted the results in Figure 11 with a dashed line. Using this metric and for the entire SNORT rule set, the cost per search pattern character is around 3.2 flip-flops per character, compared to the 3.6 logic cells per character reported by the P&R tools.

### 3.3. Comparison of DCAM and Discrete Comparator CAM

To get a better feeling for the improvement of the DCAM architecture compared to our earlier discrete comparator CAM design, we implemented the rule sets we used in our previous article in the DCAM architecture. These rule sets were smaller for two reasons: first the area cost was higher, and fewer rules could fit in a given device. Second, our earlier work focused mainly on high performance, and gave optimal results for a few hundred of rules. In our earlier work we reported performance and area cost for processing 4 bytes per cycle (i.e. $P = 4$). Hence to obtain results that would be directly comparable, we used the same parallelism setting and implemented a 4-byte per cycle DCAM. We also refrain from using partitioning in the DCAM both to remain closer to the previous design but also because the number of rules is small.

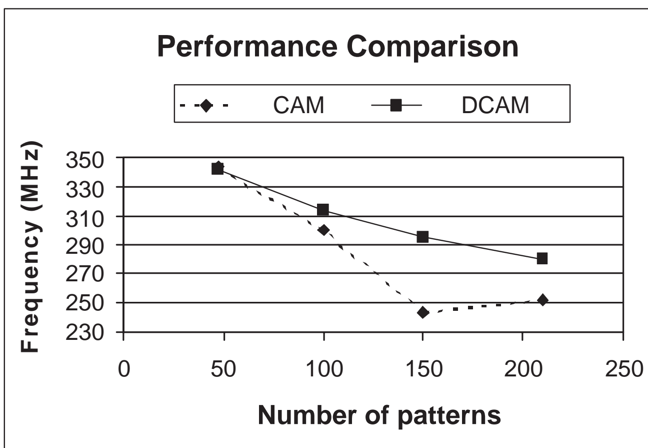


**Figure 12. Performance comparison between the Discrete Comparator CAM and the DCAM architectures.**

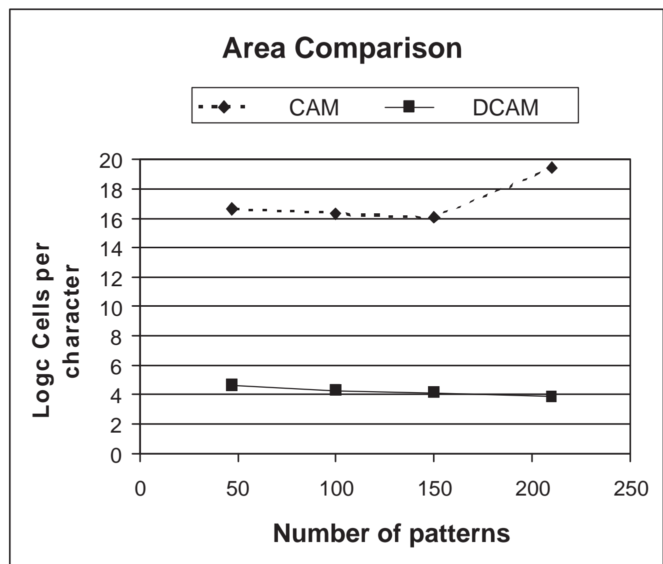


**Figure 13. Area cost comparison between the Discrete Comparator CAM and the DCAM architectures.**

Figure 12 plots the operating frequency for our earlier architecture (tagged CAM) and our proposed DCAM architecture for a number of patterns ranging from 50 to 210 rules. The results show that while for the smallest rule set both implementations operate at 340 MHz, when the rule set size increases, the scalability of the DCAM approach is better, and for 210 rules achieves about 12% better frequency.

Figure 13 plots the cost of the designs again in terms of logic cells per search pattern character. In is clear that the DCAM architecture results in drastically smaller designs: for the largest rule set, the DCAM area cost is about 4 logic cells per character, while the cost of our earlier design is almost 20 logic cells per character. All in all, and for these rule sets, the DCAM architecture offers 12% better performance at an area cost of about one fifth as compared to our discrete comparator CAM design.

## 4. Related Work

Pryor, Thistle and Shirazi in 1993 made one of the first attempts in string matching using FPGAs [11]. Since then many others researchers have worked on FPGA-based string match systems.

Most researchers designed there pattern matching architectures using regular expressions and either non-deterministic or deterministic finite automata. Sidhu and Prassanna presented a method for finding matches to a given regular expression text using NFAs, and focused mainly in minimizing the space required to perform the matching[13]. Franklin,Carver and Hutchings also used regular expressions to store patterns extracted from SNORT database[7]. They managed to include 16,000 characters[1] requiring 2.5 logic cells per matching character. Moscola, Lockwood, Loui and Pachos constructed regular expressions and converted them into DFAs. To increase their design's throughput, they use multiple parallel content scanners and exploited in this way packet-level parallelism [10]. Lockwood also implemented a sample application on FPX board using a single regular expression[9]. Clark and Schimmel [3] developed a pattern matching coprocessor that supports the entire SNORT rule-set using NFAs. In order to reduce design's area they used centralized decoders instead of character comparators for the NFA state transitions. Their design processes one character per cycle, and implemented over 1,500 patterns (17,537 characters), requiring about 1.1 logic cells per search pattern character. They achieved an operating frequency of 100 MHz for a total throughput 0.8 Gbps using a Virtex-1000 device.

While most proposed designs use N/DFAs, researchers have proposed architectures based on CAMs and discrete comparators. Gokhale, et al [8] used a CAM to implement Snort rules. Their hardware can serve total throughput of 2 Gbps in a VirtexE device. Cho, Navab and Mangione-Smith [2] presented an architecture of pattern matching using discrete comparators.They were the first to use 4 parallel comparators for every pattern in order to exploit parallelism and process 4 bytes of packet data every clock cycle. They used an Altera EP20K device and achieved a frequency of

[1] non-Meta characters, size of regular expression

**Table 1. Detailed comparison of string matching FPGA designs**

| Description | Input Bits/ c.c. | Device | Freq. MHz | Throu-ghput (Gbps) | Logic Cells[2] | Logic Cells /char | #Patterns × #Characters |
|---|---|---|---|---|---|---|---|
| **Sourdis-Pnevmatikatos** | 32 | Virtex2-6000 | 303 | 9.708 | 64,268 | 3.56 | |
| | | Spartan3-5000 | 154 | 4.913 | 66,556 | 3.69 | |
| | 8 | Virtex2-3000 | 385[9] | 3.080 | 23,228 | 1.28 | |
| Pre-decoded | | | 372[10] | 2.975 | 19,854 | 1.10 | |
| CAMs | | | 335[11] | 2.678 | 17,538 | 0.97 | 1,466×12.3 |
| | | Spartan3-1500 | 261[9] | 2.086 | 26,620 | 1.47 | |
| | | | 263[10] | 2.107 | 23,100 | 1.28 | |
| | | | 250[11] | 2.000 | 19,902 | 1.10 | |
| | 32 | Spartan3-5000 | 186 | 5.941 | 65,466 | 4.38 | 1286×11.6 |
| **Sourdis-Pnevmatikatos**[15] | 32 | Virtex-1000 | 171 | 5.472 | 8,132 | 16.64 | |
| | | VirtexE-1000 | 245 | 7.840 | 7,982 | 16.33 | 47 × 10.4 |
| Discrete | | Virtex2-1000 | 344 | 11.008 | 8,132 | 16.64 | |
| Comparators | | VirtexE-2600 | 204 | 6.524 | 47,686 | 19.40 | 210 × 11.7 |
| | | Virtex2-6000 | 252 | 8.064 | 47,686 | 19.40 | |
| **Sidhu et al.**[13]NFAs | 8 | Virtex-100 | 57.5 | 0.460 | 1,920 | ~66 | (1×) 29[5] |
| **Franklin et al.**[7] | 8 | Virtex-1000 | 31 | 0.248 | 20,618 | 2.57 | 8,003[6] |
| Regular | | VirtexE-2000 | 50 | 0.400 | 20,618 | 2.57 | 8,003[6] |
| Expressions | | | 49.5 | 0.396 | 40,232 | 2.52 | 16,028[6] |
| **Moscola et al.**[10]DFAs[3] | 32 | VirtexE-2000 | 37 | 1.184 | 8,134[3] | 19.4 | 21 × 20[4] |
| **Lockwood**[9]FSM+counter | 32 | VirtexE-1000 | 119 | 3.808 | 98 | 8.9 | 1 x 11 |
| **Gokhale et al.**[8]Dis. Comparators | 32 | VirtexE-1000 | 68 | 2.176 | 9,722 | 15.2 | 32 × 20 |
| **Cho et al.**[2]Dis. Comparators | 32 | Altera EP20K | 90 | 2.880 | ~17,000 | 10.55 | 105 × 15.3 |
| **Clark et al.**[3]NFAs-Shared Decoders | 8 | Virtex-1000 | 100 | 0.800 | ~19,660 | ~1.1 | 1,500×11,7[7] |

90MHz, achieving 2.88 Gbps throughput.

Our previous research [15] showed that architecture with discrete comparators, using fine-grain pipeline and parallelism, can achieve roughly twice the operating frequency and throughput on the same or equivalent devices compared to other architectures. However the area cost was 4-5 logic cells per matching character (multiplied by $N$ for designs that process $N$ characters per cycle), when other designs need betwen 1.1 and 2.5 logic cells per matching character.

Summarizing, DCAM achieves similar and for large number of patterns better performance compared to our earlier work. The area cost also decreases at 0.97-1.28 logic cells per character for designs without parallelism, and less than one logic cell for designs that process four characters per cycle. Table 1 summarizes a detailed comparison between related work and DCAM designs. Compared to Clark and Schimmel [3], DCAMs have similar or lower area cost. A direct performace comparison is not possible because the SRL16 cell is not supported by the Virtex devices used by Clark and Schimmel. Based on our earlier results [15], we can estimate that a design operates in a Virtex2 two times faster than in a Virtex. Using this general rule, we can roughly estimate that our architecture (335-385MHz on a Virtex2) is at least 70% faster than Clark's et al. architecture (100MHz on a Virtex). Finally, on Table 1 some initial results for Spartan3 devices are presented. These results show that Spartan3 has about 50% to 100% worst performance. Considering that is about 10 times cheaper, Spartan3 can constitute a very cost-effective solution.

[2]Two *Logic Cells* form one *Slice* and 2 or 4 Slices form one *CLB* in Virtex-VirtexE and Virtex2-Virtex2 Pro devices respectively.

[3]These results do not include the cost/area of infrastructure and protocol wrappers.

[4]21 regular expressions,with 20 characters on average, (420 character).

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

[6]Sizes refer to Non-meta characters and are roughly equivalent to 1600, and 800 patterns of 10 characters each.

[7]over 1,500 patterns that contain 17,537 characters.

[8]4 Parallel FSMs on different Packets.

[9]Design with group size 64.

[10]Design with group size128.

[11]Design with group size 256.

## 5. Conclusions and Future Work

In this paper we expanded on our previous work and proposed the Decoded CAM architecture for content matching for Snort NIDS. DCAMs offer simplicity and regularity while at the same time can operate at high frequency and require a modest area cost for their implementation. We used fine-grain pipeline to obtain high operating frequencies and we can use parallelism to increase the processing width and hence the processing throughput.

Based on our DCAM architecture we can produce implementations that include the entire payload processing part of the SNORT rule-set in a single device (a Virtex2-3000 for $P = 1$ and a Virtex2-6000 for $P = 4$), having total throughput of at least 2.7 and 9.7 Gbps respectively. DCAM's performance and area cost results showed that it is a better solution compared to our previous work, and that can achieve at least equal area cost compared to the best published.

There are still improvements that can be used to make DCAM architecture better. One parameter we have not explored in detail is the width of pre-decoded incoming data. In this work we used 8 bits. However, we could consider different widths such as 4 bits (a single nibble fitting in a LUT), or 12 bits, etc. Narrower decoders may prove beneficial since they may increase the degree of sharing of decoder terms even further but require wider AND gates to determine the pattern match. A comparison of the effect of this parameter on the performance and cost of DCAMs would be very interesting.

Finally, an evaluation of the partitioning alternatives would also be very interesting. Our partitioning algorithm is greedy, and hence may leave room for further improvements. A more sophisticated algorithm could take into account the exact location of the similarities between search patterns (in order to increase the degree of shift register sharing), and would use a global instead of local approach to cost minimization. We plan to investigate these issues further in order to further improve the DCAM architecture.

## Acknowledgments

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union. We are greatful to E. Markatos and S. Paschalakis for their constructive comments, and A. Meletioy for his support.

## References

[1] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, and M. Polychronakis. Performance analysis of content matching intrusion detection systems. In *Proceedings of the International Symposium on Applications and the Internet (SAINT2004)*, Tokyo, Japan, January 2004.

[2] Y. H. Cho, S. Navab, and W. Mangione-Smith. Specialized hardware for deep network packet filtering. In *Proceedings of 12th International Conference on Field Programmable Logic and Applications*, France, 2002.

[3] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns. In *Proceedings of 13th International Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, 2003.

[4] C. J. Coit, S. Staniford, and J. McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of snort. In *DISCEXII, DAPRA Information Survivability conference and Exposition*, Anaheim, California, USA, June 2001.

[5] N. Desai. Increasing performance in high speed NIDS. In *www.linuxsecurity.com*, March 15 2002.

[6] M. Fisk and G. Varghese. An analysis of fast string matching applied to content-based forwarding and intrusion detection. In *Techical Report CS2001-0670 (updated version)*, University of California - San Diego, 2002.

[7] R. Franklin, D. Carver, and B. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[8] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In *Proceedings of 12th International Conference on Field Programmable Logic and Applications*, France, 2002.

[9] J. W. Lockwood. An open platform for development of network processing modules in reconfigurable hardware. In *IEC DesignCon '01*, Santa Clara, CA, USA, January 2001.

[10] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, USA, April 2003.

[11] D. V. Pryor, M. R. Thistle, and N. Shirazi. Text searching on splash 2. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–177, April 1993.

[12] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Administration Conference*, Seattle Washington, USA, November 1999.

[13] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Rohnert Park, CA, USA, April 2001.

[14] SNORT official web site. http://www.snort.org.

[15] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a network intrusion detection system. In *Proceedings of 13th International Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, 2003.

[16] Xilinx. Virtex-II Platform FPGAs: Detailed description. http://direct.xilinx.com/bvdocs/publications/ds031-2.pdf, October 2003.