Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems

Hong-Jip Jung, Zachary K. Baker and Viktor K. Prasanna
University of Southern California, Los Angeles, CA, USA
hongjung, zbaker, prasanna@usc.edu

Abstract

The use of reconfigurable hardware for network security applications has recently made great strides as Field-Programmable Gate Array (FPGA) devices have provided larger and faster resources. The performance of an Intrusion Detection System is dependent on two metrics: throughput and the total number of patterns that can fit on a device.

In this paper, we consider the FPGA implementation details of the bit-split string-matching architecture. The bit-split algorithm allows large hardware state machines to be converted into a form with much higher memory efficiency. We extend the architecture to satisfy the requirements of the IDS state-of-the-art.

*We show that the architecture can be effectively optimized for FPGA implementation. We have optimized the pattern memory system parameters and developed new interface hardware for communicating with an external controller. The overall performance (bandwidth * number of patterns) is competitive with other memory-based string matching architectures implemented in FPGA.*

1 Introduction

The continued discovery of programming errors in network-attached software has driven the introduction of increasingly powerful and devastating attacks [11, 12]. Attacks can cause destruction of data, clogging of network links, and future breaches in security. In order to prevent, or at least mitigate, these attacks, a network administrator can place a firewall or Intrusion Detection System at a network choke-point such as a company's connection to a trunk line. A firewall's function is to filter at the header level; if a connection is attempted to a disallowed port, such as FTP, the connection is refused. This catches many obvious attacks, but in order to detect more subtle attacks, an Intrusion Detection System (IDS) is utilized. The IDS differs from a firewall in that it goes beyond the header, actually searching the packet contents for various patterns. Detecting these patterns in the input implies an attack is taking place, or that some disallowed content is being transferred across the network. In general, an IDS searches for a match from a set of rules that have been designed by a system administrator. These rules include information about the IP and TCP header required, and, often, a pattern that must be located in the stream. The patterns are some invariant section of the attack; this could be a decryption routine within an otherwise encrypted worm or a path to a script on a web server. Current IDS pattern databases reach into the thousands of patterns, providing for a difficult computational task.

In [18], a technique for reducing the out-degree of pattern-matching state machines is presented. This innovative technique allows state machines to be represented using significantly less state memory that would be required in a naïve implementation. Through the use of "bit-splitting," a single state machine is split into multiple machines that handle some fraction of the input bits. The best approach seems to be 4 smaller machines, each handling 2 bits of the input byte. Thus, instead of requiring 2^8 memory locations for each of the possible input combinations, only 2^2 locations are required per unit, for a total of $4*2^2=16$ locations over the four machines. Due to the disconnected nature of the multiple state machines, the final states must be reconnected using a "partial match vector" that ensures all of the machines are in an output state before the system will produce a result.

The earlier work did not provide many of the requirements for a realistic implementation. Our contribution is to adapt the basic architectural design from [18] to an FPGA implementation. This contribution is in several parts: first, the costs of logic and routing are included in our analysis and simulation, two, the problems of reporting results back to an external controller are addressed, and three, the architecture is modified to make the most efficient usage of the on-chip FPGA memory blocks. In Section 5 we show that through the use of a single FPGA device, our system archi-

[1] Supported by the United States National Science Foundation/ITR under award No. ACI-0325409 and in part by an equipment grant from the Xilinx and HP Corporations.

1-4244-0054-6/06/$20.00 ©2006 IEEE

tectures can support multi-Gigabit rates with 1000 or more patterns, while providing encoded attack identifiers.

Field Programmable Gate Arrays (FPGA) provide a fabric upon which applications can be built. FPGAs, in particular, SRAM based FPGAs from Xilinx [19] or Altera [2] are based on "slices" composed of look-up tables, flip-flops, and multiplexers. The values in the look-up tables can produce any combinational logic functionality necessary, the flip-flops provide integrated state elements, and the SRAM-controlled routing direct logic values into the appropriate paths to produce the desired architecture. Recently, reconfigurable logic has become a popular approach for network applications due to these characteristics.

This paper is structured as follows: We begin with a brief discussion of some of the prior work in string matching for Intrusion Detection, and the basic principles of the Aho-Corasick and Bit-split algorithms. We then present our work on the efficient design of the reconfigurable hardware implementation of the architecture described in [18]. This includes an analysis of appropriate memory sizes as well as additional hardware components required to make a feasible system. Finally, we will present some results from our experiments, showing that while the architecture is competitive, other memory-based architecture do have some performance advantages for databases of string literals.

2 Related Work in Hardware IDS

Snort [16] and Hogwash [9] are current popular options for implementing intrusion detection in software. They are open-source, free tools that promiscuously tap the network and observe all packets. After TCP stream reassembly, the packets are sorted according to various characteristics and, if necessary, are string-matched against rule patterns.

System-level optimization has been attempted in software by SiliconDefense [10]. They have implemented a software tree-searching strategy that uses elements of the Boyer-Moore [14] and Aho-Corasick [1] algorithms to produce a more efficient search of matching rules in software, allowing more effective usage of resources by preventing redundant comparisons.

FPGA solutions attempt to provide a more powerful solution. In our previous work in regular expression matching [15], we presented a method for matching regular expressions using a Non-deterministic Finite Automaton, implemented on a FPGA.

In another of our previous works [4], we demonstrated an architecture based on the Knuth-Morris-Pratt algorithm. Using a maximum of two comparisons per cycle and a small buffer, the system can process at least one character per cycle. This approach is different from a general state machine because a general state machine, such as an Aho-Corasick tree machine, can require a large number of concurrent byte comparisons. The paper further proves an upper bound on the buffer size.

In [13], a multi-gigabyte pattern matching tool with full TCP/IP network support is described. The system demultiplexes a TCP/IP stream into several substreams and spreads the load over several parallel matching units using Deterministic Finite Automata pattern matchers.

The NFA concept is updated with predecoded inputs in [7]. The paper addresses the problem of poor frequency performance for a large number of patterns, a weakness of earlier work. By adding predecoded wide parallel inputs to a standard NFA implementations, excellent area and throughput performance is achieved.

A recent TCAM-based approach [20] utilizes a large number of tables and is dependent on having fast TCAM and SRAM memories available to a controller. Because the authors assume a 32 bit CAM word, patterns usually require a large number of individual lookups. However, through a probabilistic analysis of lookup behavior, the authors prove that far fewer lookups are actually required in practice that might be expected in a worst-case scenario. This allows a minimum of hardware resources to be expended.

3 Motivation for Bit-split Architecture and FPGA Implementation Issues

The Aho-Corasick [1] string matching algorithm allows multiple strings to be searched in parallel. A finite state machine is constructed from a set of keywords and is then used to process the text string in a single pass.

However, like other implementations of state machines that require one transition in each cycle, a huge amount of storage is required. This problem comes from the large number of edges, maximum 256, pointing to the potential next states. Reducing these edges is the contribution of [18] that this work is based on. The Aho-Corasick algorithm will be described in more detail in Section 3.1.1.

By splitting one Aho-Corasick state machine into a set of several state machines, the number of out-edges per state is significantly reduced. Each state machine is responsible for a subset of the input bits, causing proportionately more states to be active in the system but with far fewer next-states for any given machine. Because the bit-split algorithm removes most of the wasted edges, the total storage required is much smaller than that of the starting machine. A more detailed explanation is given in Section 3.2.

There are many advantages of the bit-split technique. First, the bit split machines maintain the ability of the Aho-Corasick machine to match strings in parallel. Second, the memory required for state transition storage reduces from 256 to 4 for each state. Third, the architecture is based on a runtime-programmed memory, thus allowing on-the-fly updates of rules without the cost of place-and-route (a problem encountered with hardwired-FPGA implementations [3, 5, 17]).

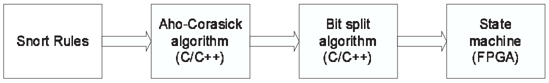

Figure 1. A procedure for converting Snort rules into an FPGA-based state machine

In [18] the bit-split algorithm is presented, but the details of the implementation and system are not considered. We are interested in FPGA implementation issues such as efficient use of block RAM and reducing routing delays. This paper makes contributions by developing efficient solutions to these issues.

3.1 Bit-split Aho-Corasick Algorithms

This section describes the behavior of the Aho-Corasick string matching machine and the conversion of this state machine to a bit-split machine. The description will be shown with a different example from [18]. This conversion is done by software, external to the hardware device. The software yields the state tables for the bit-split machine, and the tables are loaded into the block RAM of FPGA at run time. Figure 1 shows this procedure.

3.1.1 Aho-Corasick Algorithm

The objective of the Aho-Corasick algorithm is to find all substrings of a given *input string* that matches against some set of previously defined strings.

These previously defined strings are called $patterns$ or $keywords$. The pattern matching machine consists of a set of states that the machine moves through as it reads one character symbol from the input string in each cycle. The movement of the machine is controlled by three types of state transitions: normal transitions (successful character matches), error transitions (when the machine attempts to realign to the next-longest potential match), and acceptance (successful matching of a full string). The operations of this algorithm are implemented as a Finite Automata. Figure 2 shows these three operations derived from the keywords {cat, et=, cmdd, net} sampled from Snort rule set.

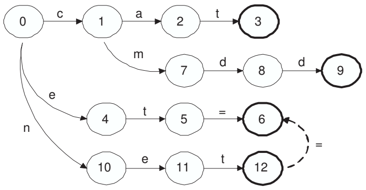

Figure 2. Pattern matching machine

The normal transition maps a current state to a next state according to the input character. For example, if the current state is 0 and the machine reads ‘c’ as the next input, then the next state will be 1. This operation is indicated as a line labeled with a corresponding character in Figure 2. The absence of this line indicates an error. When the state machine cannot make a successful forward match, it follows an error transition. Most error transitions end in state 0. However in case of state 12, if the machine sees ‘=’ as a next input, then it follows the error transition (the dotted line), and the next state is determined to be state 6. There are 255 arrows outgoing from state 12, all ending in the root node 0. Even though we do not draw these arrows, storage is required for these transitions. By using the error transition, we can store the information about substrings shared between keywords. For instance, “bookkeeper” and “keepsake” share the substring “keep”. Thus, the input string “bookkeepsake” would cause the state machine to reach the ‘p’ character in the “bookkeep” branch and then switch to the “keepsake” branch when the ‘s’ is detected. Finally, if the state machine reaches an accepting state (bold circles), it means that a keyword was matched by an input string. Let us see the behavior of this machine when it sees an input string “net=xc” by an example below. This example indicates the state transitions made by the Aho-Corasick state machine in processing the input string.

```
(0)n(10)e(11)t(12)=(6)x(0)c(1)
```

The number between parentheses is a state. Initially, the current state is state 0. The machine moves through the various states as it reads character ‘n’, ‘e’, and ‘t’. Then it reaches state 12, an accepting state, and outputs a result for the matched keyword “net”. On reading input character ‘=’, the machine makes error transition, going to state 6. Here the machine outputs a matched keyword “et=” because state 6 is also an accepting state. When it sees ‘x’, it makes an error transition to state 0 because there is no better transition possible. The machine starts again from initial state 0 and then goes to state 1 when it reads ‘c’.

3.2 Construction of Bit-split Finite State Machines

The architecture of the string matching machine of [18] is shown in Figure 5. This figure is based on a rule module containing 16 keywords. The state transition table, generated by the algorithm explained in this section, fills the memory of each tile.

From the state machine AC constructed by Aho-Corasick algorithm, eight 1-bit state machines are generated. Let B_0, B_1, $\ldots$, B_7 be binary machines corresponding to each 1-bit of 8-bit ASCII character. We will indicate state i of AC as state $AC - i$. To build B_i, the construction is started from $AC - 0$ and create $B_i - 0$. $B_i - 0$ contains only $AC - 0$. We look at the ith bit of input character and separate the procedure into two cases, i.e. whether the ith bit is 0 or 1. If $B_i - 0$, containing only $AC - 0$, reaches some next states in the Aho-Corasick state machine because the ith bit is 0, and if the set of those states are not included in B_i machine, then we create a new state ($B_i - 1$) and add it to B_i. Also, if state 0 of $B_i - 0$ state reaches some next states by seeing the ith bit is 1 and the set of those states are not existing in B_i, then we also create a new state ($B_i - 2$). This procedure also considers the error transition: if a state m can reach a state n through an error transition line by reading the ith bit of input character, then state n is put into new bit-split state. From these newly generated states of B_i machine ($B_i - 1$ and $B_i - 2$), we do the above procedure repeatedly until no more new states are generated. Note that in AC, there is only one reachable next state by reading input character (this separates string matching from the more elaborate regular expression matching). But in B_i, there can be multiple reachable states by reading one bit of the input character. A resulting state in B_i is an accepting state if at least one of its corresponding states of AC are accepting states. And the partial match vector, indicating which of the strings might be matched at that point, is maintained for the states of B_i.

Let us understand this with a simple example. Because we have already provided the general description of how to construct 1-bit state machine and the specific example of these machines is given in [18], we will show the construction of 2-bit state machines in this paper, in particular bits 5 and 4, or the B_{54} machine. The reason we deal with 2-bit state machine is that an optimal number of bit-split state machine is 4 (2-bit state machine), not 8 (1-bit state machine) as previously shown in [18]. Table 1 shows the ASCII code of characters used in this explanation. The resulting graph is shown in Figure 3.

	7	6	5	4	3	2	1	0
=	0	0	1	1	1	1	0	1
a	0	1	1	0	0	0	0	1
c	0	1	1	0	0	0	1	1
d	0	1	1	0	0	1	0	0
e	0	1	1	0	0	1	0	1
m	0	1	1	0	1	1	0	1
n	0	1	1	0	1	1	1	0
t	0	1	1	1	0	1	0	0

Table 1. ASCII code of character ‘=’, ‘a’, ‘c’, ‘d’, ‘e’, ‘m’, ‘n’ and ‘t’

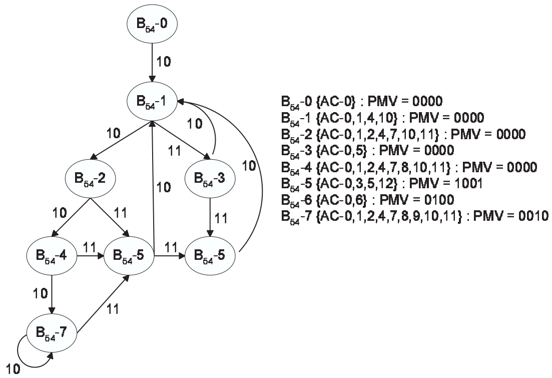
B54-0
10
B54-1
10
10
11
10
B54-2
B54-3
10
10
11
11
B54-4
11
B54-5
11
B54-5
10
11
B54-7
10
B54-0 {AC-0} : PMV = 0000
B54-1 {AC-0,1,4,10} : PMV = 0000
B54-2 {AC-0,1,2,4,7,10,11} : PMV = 0000
B54-3 {AC-0,5} : PMV = 0000
B54-4 {AC-0,1,2,4,7,8,10,11} : PMV = 0000
B54-5 {AC-0,3,5,12} : PMV = 1001
B54-6 {AC-0,6} : PMV = 0100
B54-7 {AC-0,1,2,4,7,8,9,10,11} : PMV = 0010

Figure 3. Sequence of state transitions

Starting from AC-0, we construct a B_{54}-0 state. The B_{54}-0 state has only $\{AC\text{-}0\}$. The state machine B_{54} has 4 outgoing edges which can be named as 00-edge, 01-edge, 10-edge, and 11-edge. Table 1 shows us that there are no 2-bit codes in 5th and 4th bit corresponding to outgoing 00-edge and 01-edge from AC machine. This means that a AC-i goes to AC-0 when it reads 00 and 01. Hence, we only have to handle the 10-edge and 11-edge. When the state machine AC sees the input character ‘c’, ‘e’, and ‘n’ the corresponding 5th and 4th code is 10, so the next state of AC-0 is AC-0,1,4,10 as shown in Figure 2. Note that AC-0 is also a reachable state by reading the input code 10, because there are many 8-bit characters which are not ‘c’, ‘e’, and ‘n’ but still have ‘10’ as their 5th and 4th bits. At this point we check whether the set $\{AC\text{-}0,1,4,10\}$ is already included in B_{54} state machine or not. Since we only have B_{54}-0 until now and B_{54}-0 has only $\{AC\text{-}0\}$ as its corresponding Aho-Corasick machine’s state, we create B_{54}-1 $\{AC\text{-}0,1,4,10\}$ and connect this to B_{54}-0 with 10-edge. Then this B_{54}-1 is put into a queue. We have processed all the works in B_{54}-1, since all the outgoing edges from AC-0 have only 10 code.

Now, the queue is not yet empty, so the bit-split algorithm retrieves the first element from the queue. In our example it should be B_{54}-1. With this B_{54}-1, we do the same procedure as above. The bit-split algorithm considers all the possible outgoing edges from all the elements of B_{54}-1, i.e. it finds all the reachable states in AC machine from AC-0,1,4,10. If $AC - 0$ sees 10 as its input code it can reach AC-0,1,4,10. Similar to this, $AC - 1$ can reach $AC - 2, 7$, and so on. Thus B_{54}-1 finds $\{AC\text{-}0,1,2,4,7,10,11\}$ as its all the reachable states when it reads 10 code on 5th and 4th bit. Since $\{AC\text{-}0,1,2,4,7,10,11\}$ do not exist, we create B_{54}-2 $\{AC\text{-}0,1,2,4,7,10,11\}$ and connect this to B_{54}-1 with 10-edge. Likewise, we create B_{54}-3 $\{AC\text{-}0,5\}$ and connect this to B_{54}-1 with 11-edge. This procedure is repeated until there are no elements in the queue. In this procedure the error transition should be considered, as in the case of constructing an Aho-Corasick machine where AC-12 can move to AC-6 by error transition when it reads ‘=’. In the procedure of constructing B_{54}, this situation occurs when B_{54}-5 $\{AC\text{-}0,3,5,12\}$ has a reachable state. Here, AC-5 finds reachable state AC-6 and this does not depend on the error transition. The error transition of AC-12 is also AC-6. Thus, $AC - 12$ cannot find any new reachable state.

The final step is to find an accepting state (partial match vector) for each state of B_{54}. This is very simple. For instance, $B_{54} - 5$ has $\{AC\text{-}0,3,5,12\}$. Among these states,

AC-3 and AC-12 are accepting states in AC. From the output function of Aho-Corasick algorithm we know that AC-3 state stands for “cat” and the AC-12 state stands for “net”. Because “cat” is the first keyword and “net” is the fourth, the partial match vector is 1001 if we assume that we are using only four keywords in this example.

The sample state transition table for B_{54} is given in Table 2 and the state transition table for B_{76}, B_{32}, and B_{10} can be made by the same procedure.

	00	01	10	11	PMV
0	0	0	1	0	0000
1	0	0	2	3	0000
2	0	0	4	5	0000
3	0	0	1	6	0000
4	0	0	7	5	0000
5	0	0	1	6	1001
6	0	0	1	0	0100
7	0	0	7	5	0010

Table 2. State transition table of B_{54}

4 Architectural Advances and Innovations

The architecture of the string matching machine of [18] is shown in Figure 5. This figure is based on the number of keywords is 16 in one rule module. The state transition table, generated by the algorithm explained in Section 3.2, fills the memory of each tile. For example, Table 2 is for tile 1 (5th and 4th bit) and four tiles constitute one rule module.

Each character of input string is divided into four 2-bit vectors and distributed to the corresponding tile of the rule module. Each tile reads this 2-bit input and selects its next state among the four states through a 4:1 MUX as illustrated in Figure 5. The next state becomes active in the next clock cycle. We do the same procedure with each tile and corresponding 2-bit input. Each memory access includes the next state pointers as well as a partial match vector for each tile. The bitwise-AND of four partial match vector yields a full match vector. If at least one bit of this full match vector is 1, it means that a keyword match has occurred. If a input string is “=net” the sequence of 5th and 4th bit is 11, 10, 10, and 11. For this input sequence, the state transitions made by B_{54} are shown below.

$$\begin{array}{llllll}
\text{state transition} & : 0 & \rightarrow 0 & \rightarrow 1 & \rightarrow 2 & \rightarrow 5 \\
\text{input sequence} & : & 11 & \rightarrow 10 & \rightarrow 10 & \rightarrow 11
\end{array}$$

When the state transition reaches state 5, tile 1 outputs a partial match vector 1001. The other three tiles will also output partial match vectors. In this case the full match vector will be 0001. By this FMV we know that the keyword “net” has been matched.

As we can see from Figure 5, we need a memory with a size of 256x48 for each tile. To implement this memory in FPGA, we must choose the most appropriate memory configuration. Slice-based RAM is far too expensive in terms of area to implement many 256x48 blocks. At one slice per 32 bits, four 256x48 RAM blocks would require 1,536 slices per 16 patterns. This is not competitive with other approaches. However, on-board RAM, in particular, Xilinx block RAM, is an appropriate choice as they do not consume logic resources. Unfortunately, the closest fit in the Xilinx Virtex family of FPGA is a 512x36 SRAM block. We have no choice but to use two 512x36 blocks for the architecture of Figure 5. Using two 512x36 block RAMs causes a loss of at least ((256x48)/(512x36))*100 = 66.7% memory space. The restriction of a block RAM size mentioned above suggests a re-thinking of the optimal number of keywords in one rule module.

	16	18	20	22	24	26	28	30	32
MAX	246	283	289	330	319	359	381	386	392
AVG	138	155	171	188	204	220	236	251	268

Table 3. Comparing the number of keywords and required states

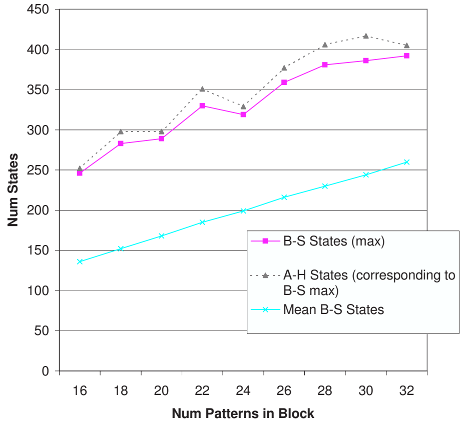

Figure 4. A relationship between the number of keywords and required states

Table 3 and Figure 4 shows the relationship between the number of keywords and the number of required states in a bit-split machine. Using the keywords from the “web-cgi” Snort rule set, Table 3 and Figure 4 illustrates the maximum number of states among all the tiles (The maximum is always from Tile 3 because the least-significant bits have the least degree of similarity). The total number of string literals in our sample Snort ruleset is 860, providing a rough idea of the overall IDS database behavior. However, there is a high volume of memory space wasted per tile if one rule module deals with 16 keywords as in [18]. If we only think about the average, 138x48 = 6624 bits are required for the case of 16 keywords. The total available memory space is 2x512x36 = 36864 bits, Therefore 82.0% of block RAM per tile are wasted. Table 4 shows the block RAM size used by each number of keywords and corresponding wasted memory ratio per tile. Contrary to the case of 16 keywords which uses 8 bits to find a next state, the number of keywords from 18 to 32 requires 9 bits, since the maximum number of states is larger than 256.

From Table 4, if we only consider area efficiency, 32 is the optimal number of keywords per rule module. But we should also take into account the frequency performance. Optimizing frequency performance complicates the problem, as considered in the next section.

	Wasted Ratio (%)
16	82.0
18	77.3
20	74.0
22	70.4
24	66.8
26	63.0
28	59.0
30	55.1
32	50.1

Table 4. The wasted ratio of block RAM corresponding to each number of keywords. All use the same two 512x36 block RAMs.

4.0.1 Full Match Vector Sizing

Before we show the performance of each case discussed in Section 4, there is one thing to be changed in old architecture shown in Figure 5. Let us examine the problem of this architecture first.

The architecture is very simple. Due to the local arrangement of small state machine blocks, a few rule modules on a device does not impact performance significantly. However, routing delays become important as the number of modules scales up. For our speed optimization, the minimization of routing delays between rule modules is more important than the optimization of rule module itself. Since many rule modules can be put into one FPGA, the arrangement of rule module is critical to the overall performance.

In our implementation, the critical path is the production of the encoded output, derived from the full match vectors (FMV) of each module. If we think of Figure 5 as an RTL level, we can find that the FMV lines, 16 per rule module, are bundled at the end of string matching detector. In order to reduce the total number of outgoing lines from the device, we use a priority encoder as in Figure 6. The priority encoder chooses one rule module if it matches one or more keywords. Multiple matches in a single module can be separated in software as a multiple match implies that multiple pattern strings have overlapped on one branch of the Aho-Corasick tree. However, for potential overlaps in multiple modules, the patterns must be arranged so that the longest pattern is in the highest priority module. This allows for shorter patterns that are a substring of the longer, matching pattern, to be extracted in software.

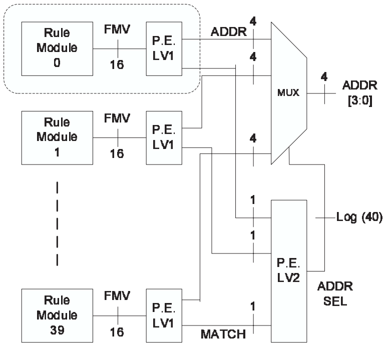

Figure 6. Priority encoder blocks for reporting results outside of device

Number of Patterns per Module	Results for 5 rule modules	
	Frequency (Mhz)	Area (slices)
16	199.6	1761
18	199.9	1800
20	189.2	1894
22	203.4	1982
24	198.2	2070
26	214.5	2158
28	226.2	2246
30	221.4	2334
32	190.3	2422

Table 5. Effect of the number of patterns on frequency and area

5 Performance Results and Comparisons

Table 5 compares the performances of all the keywords that we showed in 4. We synthesized only 5 rule modules for comparison to provide a rough approximation of the performance of a scaled-up system. The relative area and time performance between the various numbers of keywords per module should remain similar as the number of modules increases. The fastest case is for 28 keywords per module. It is faster than the case of 32 keywords by 15%. But in the view of area efficiency, it is worse than that of 32 keywords by 9%. Hence, the 28 keywords per module provides the highest frequency performance while maintaining a high number of patterns simultaneously matched.

The synthesis tool for the VHDL designs is Synplicity Synplify Pro 7.2 and the place-and-route tool is Xilinx ISE 6.2. The target device is the Virtex4 fx100 with speed grade -12. The FX series device provides a much better RAM/logic ratio compared to the other devices in the Virtex IV series. Because the architecture is constrained only by the amount of block RAM and not the logic, it is best to

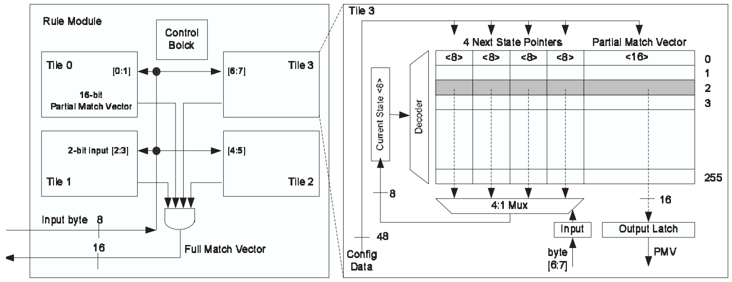

Figure 5. Bit-split Architecture of [18]

find the device with the largest amount of block RAM. The results are based on the placed-and-routed design. The Virtex 4 device supports 376 block RAMs, and allowing 47 rule modules. With 47 rule modules at 28 patterns per module, a single device can handle 1,316 patterns. At 200 MHz and one byte per cycle, the system has a throughput of 1.6 Gbps. The comparisons against other implementations are shown in Table 6. We only consider architectures providing on-the-fly memory-based pattern recognition capabilities, excluding hardwired reconfiguration-based architectures.

Table 6 has several columns each providing a metric of the behavior of the various designs. The first column, bandwidth, gives a measure of the throughput provided by the design. In hardware architectures, this is largely the frequency of the device multiplied by the number of bits consumed in each cycle. Memory requirements are measured in bytes per character. This measure is most relevant when the memories are on-board block RAMs, as they are limited and do not have the vast address space possible with external memories. Logic cells per character is a measure of how much reconfigurable logic is required to implement the various architectures. An architecture like the Bit-split architecture in which a single hardware module is responsible for many patterns will have a lower number than an architecture like the KMP design that has state machine for each pattern. The final column is the total number of characters that can be placed on a device. For the bit-split implementation, it is the total number of characters from the lexically sorted Snort database subset that could fit in 47 modules. For the KMP architecture of [4] it is the maximum number of characters (32) per module times the number of modules. The other results are taken from their respective source publications [6, 8].

Table 6 only has the comparison of results makes clear that the bit-split algorithm is fairly efficient in its memory consumption compared to the original Aho-Corasick tree. However, it is not particularly efficient compared to many of the other competing hardware approaches. That said, while the memory per character numbers in Table 6 do seem to make the bit-split architecture fare poorly in comparison to the other architectures, it must be remembered that the memories are not being fully utilized. For the 28 keyword case, almost 60% of the RAM is empty. This is entirely due to the FPGA BRAM default sizing – while a 256 entry RAM would be more appropriate even for a large number of patterns per module, the extra space is essentially free. Thus, it might be more appropriate to have the Mem (bytes/char) metric as 23 bytes per character instead of 46.

Although we get worse results than the original bit-split ASIC work [18] in some aspects of our experiments, that is largely due to the very different assumptions made in the earlier paper. The earlier results in [18] were based only on an estimate of the bit-split memory speed and size. As well, the issues of routing and outputting the results of the state machines were not considered. The authors of the earlier paper compared their performance estimates for single memory units on custom VLSI against placed-and-routed FPGA system implementations. These are not reasonable comparisons, as the expense and performance of custom ASICs make them impossible to judge against FPGA architectures. As well, the earlier results were for single units. Adding the other necessary elements of a system architecture could only have a detrimental effect on the system performance, as clearly demonstrated by our results. By building and simulating the complete architecture, we have contributed a better understanding of the performance that can be expected from the bit-split algorithm on FPGA. Although our architecture is not tested on a real network, we expect similar results in those environments because we would assume most TCP reassembly, etc. would be handled off of the FPGA.

6 Conclusion

With the growing importance of Intrusion Detection Systems, the performance and efficiency of the string matching architecture is essential. In [18], the bit-split algorithm provides a good architecture for high speed string matching. We have considered FPGA implementation details such as memory efficiency and pin count not addressed in the original paper [18]. Hence, details such as the number of keywords per rule module were not optimized in the earlier

	Device	BW (Gbps)	Mem (bytes/char)	Logic Cells/char	Characters Total
USC Bitsplit	Virtex 4 fx100	1.6	46	0.27	16715
USC KMP Arch	Virtex II Pro	1.8	4	3.2	3200
UCLA ROM Filter	Virtex 4	2.2	5.72	0.209	32168

Table 6. Results comparisons against various other memory-based on-the-fly reconfigurable implementations in reconfigurable hardware

work. Because the original paper did not simulate the details of their architecture, it did not consider the very real problems of routing delay and pin count. In this paper, we have implemented and tested bit-split algorithm to determine the most memory efficient number of keywords per module. Also, we have developed a new architecture utilizing a priority encoder to reduce the number of external IO pins. Our approach is not just a minor change for finding better fit for Xilinx BRAM components. We have laid foundations in this work for extending the bit-split algorithm to a realistic, deployable implementation.

In our future work, we plan to show that the bit-split architecture is flexible enough that arbitrary DFAs (including regular expression pattern matching algorithms) can be adapted, while maintaining competitive data rates in an FPGA implementation.

References

[1] A. V. Aho and M. J. Corasick. Efficient String Matching: an Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.

[2] Altera, Inc. `http://www.altera.com`.

[3] Z. K. Baker and V. K. Prasanna. A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.

[4] Z. K. Baker and V. K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *The Twelfth Annual ACM International Symposium on Field-Programmable Gate Arrays (FPGA '04)*, 2004.

[5] Y. Cho and W. H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. In *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.

[6] Y. Cho and W. H. Mangione-Smith. Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network . In *Proceedings of the Thirteenth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2005 (FCCM '05)*, 2004.

[7] C. R. Clark and D. E. Schimmel. Scalable Parallel Pattern Matching on High Speed Networks. In *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.

[8] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Implementation of a Deep Packet Inspection Circuit using Parallel Bloom Filters in Reconfigurable Hardware. In *Proceedings of the Eleventh Annual IEEE Symposium on High Performance Interconnects (HOTi03)*, 2003.

[9] Hogwash Intrusion Detection System, 2004. `http://hogwash.sourceforge.net/`.

[10] C. Joit, S. Staniford, and J. McAlerney. Towards Faster String Matching for Intrusion Detection. `http://www.silicondefense.com`, 2003.

[11] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security & Privacy Magazine*, 1(4), July-Aug 2003.

[12] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-propagating Code. *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, April 2003.

[13] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of the Eleventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, 2003.

[14] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):762–772, Oct. 1977.

[15] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *Proceedings of the Ninth Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, 2001.

[16] Sourcefire. Snort: The Open Source Network Intrusion Detection System. `http://www.snort.org`, 2003.

[17] I. Sourdis and D. Pnevmatikatos. A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.

[18] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proceedings of the 32nd Annual Intl. Symposium on Computer Architecture (ISCA 2005)*, 2005.

[19] Xilinx, Inc. `http://www.xilinx.com`.

[20] F. Yu, R. Katz, and T. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *Proceedings of the Twelfth IEEE International Conference on Network Protocols (ICNP)*, 2004.