A Computationally-efficient Engine for Flexible Intrusion Detection[1]

Zachary K. Baker and Viktor K. Prasanna

Abstract

Pattern matching for network security and intrusion detection demands exceptionally high performance. This paper describes a novel systolic array-based string matching architecture using a buffered, two-comparator variation of the Knuth-Morris-Pratt (KMP) algorithm. The architecture compares favorably with the state-of-the-art hardwired designs while providing on-the-fly reconfiguration, efficient hardware utilization, and high clock rates. KMP is a well-known, computationally-efficient string matching technique that uses a single comparator and a precomputed transition table. Through the use of the transition table, the number of redundant comparisons performed is reduced. Through various algorithmic changes, we enable KMP to be used in hardware, providing the computational efficiency of the serial algorithm and the high throughput of a parallel hardware architecture. The efficiency of the system allows for a faster and denser implementation than any other RAM-based exact match system. We add a second comparator and an input buffer, then prove that the modified algorithm can function efficiently implemented as an element of a systolic array. The system can accept at least one character in each cycle, while guaranteeing that the stream will never stall. In this paper, we prove the bound on the buffer size and running time of the systolic array, discuss the architectural considerations involved in the FPGA implementation, and provide performance comparisons against other approaches.

[1]Supported by the United States National Science Foundation/ITR under award No. 0311823 and No. ACI-0325409 and in part by an equipment grant from the HP and Xilinx Corporations. A preliminary version of this paper appears in the proceedings of FPGA '04.

1 Introduction

Methods commonly used to protect against security breaches include firewalls with filtering mechanisms to screen out obviously dangerous packets, and intrusion detection systems which use much more sophisticated rules and pattern matching to sense potential malicious packets.

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 that imply an attack is taking place, or that some disallowed content is being transferred across the network. Current IDS pattern databases reach into the thousands of patterns, providing for a difficult computational task. The computational resources required are in the form of the parallel string matching against thousands of patterns.

Using highly-parallel, configurable string matching units, our technique provides the opportunity for dramatic improvements in performance and capability of these systems. Our architecture offers increased computational efficiency over brute-force systolic arrays, in which the input stream is shifted past a comparator equal in size to the pattern. The brute force approach requires $O(nk)$ comparisons, where k is the pattern size and n is the input stream size. Our approach requires $O(n + k)$ comparisons. This computational efficiency translates into a smaller and faster implemention than a comparable on-the-fly reconfigurable system.

Because the IDS must inspect at the line rate of its data connection, IDS pattern matching demands exceptionally high performance. This performance is dependent on the ability to match against a large set of patterns, and thus the ability to automatically optimize and synthesize large designs is vital to a functional network security solution.

Much work has been done in the field of string matching for network security [6, 12, 13, 16, 21]. However, the study of the *automatic design* of efficient, flexible, and powerful system architectures is still in its infancy.

A complete Intrusion Detection Systems (IDS) based on the Snort rules [20] requires a system optimized for thousands of rules, many of which require string matching against the entire data segment of a packet. High levels of performance are necessary to provide real-time string matching at trunk line speeds. Snort, the open-source IDS [20] has thousands of content-based rules. Each of these rules require that a packet be searched in its entirety for the occurrence of some “fingerprint” string. Using naïve methods, this is unworkable. Even with the most sophisticated algorithms, sequential microprocessor-based implementations cannot provide the level of service available in a customized hardware device. In [5], a Dual 1GHz Pentium III system, using 845 patterns, runs at only 50 Mbps. For small network with limited traffic and a maximum wire speed of 100 Mbps, the software approach might be acceptable. However, for larger networks and higher bandwidth connections, the software-based approach will be forced to not scan some packets and potentially let an attack pass undetected. A hardware-based approach is more capable of providing support for very high bandwidth connections that are increasingly common.

Parallel hardware architectures offer large advantages in time performance compared to software designs, due to easily extracted parallelism in the Intrusion Detection string matching problem. A general ASIC design would be fast but not suitable due to the dynamic nature of the ruleset – as new vulnerabilities and attacks are identified, new rules must be added to the database and the device configuration must be regenerated. However, a Field-Programmable Gate Array (FPGA) allows for exceptional performance due to the parallel hardware nature of execution as well as the ability to customize the device for a particular set of patterns. While each of our units can be reconfigured at runtime to handle changing patterns, in practice the patterns would be grouped according to size such that the unit sizes can be minimized. An FPGA can provide near-ASIC performance and parallelism, along with the ability to modify the hardware to a particular set of patterns.

FPGA solutions have recently become a popular strategy for implementing all manner of applications. The development of standard models [23] has allowed their use in deployable systems to expand into previously unexpected areas. Reconfigurable logic has become a very popular approach for network applications, from cryptography [24] to flow monitoring [17]. However, byte oriented operations such as string matching is one of the great strengths of FPGA fabrics. In the earliest work in regular expression matching [18, 19], a method was presented for matching regular expressions using a Non-deterministic Finite Automaton, implemented on a FPGA.

Our implementation of the modified KMP algorithm allows string matching and on-the-fly reconfiguration within the confines of a systolic array, increasing time and area performance compared to a global reconfiguration strategy. The power of the systolic array lies in its ability to reduce total device interconnect, and to provide the flexibility of a memory-based pattern matcher within the architecture. By allowing only short connections within a small, repeated unit, a design will achieve a higher operating frequency as well as lower area consumption.

Any string matching implementation where the number of byte-level comparators is less than the size of the pattern in bytes will require more than one comparison per byte whenever a partially matched pattern fails. Without appropriate precautions in the design of the algorithm and hardware, this can cause stalling that is unacceptable in a systolic array. We first prove that it is possible to create string matching unit that will not stall. We then present a proof of the minimum buffer size required. This allows us to implement in FPGA a stall-less RAM-based string matcher with smallest possible buffer.

We begin this paper by reviewing the related work in the string matching field, followed by an introduction to the Knuth-Morris-Pratt (KMP) algorithm. We then present our modifications to the algorithm, as well as some operational examples. We show that the modified algorithm can always accept at least one character per cycle, and prove the the minimum size of the buffer. Finally, we present the results of our algorithm implementation and a scheme for on-the-fly reconfiguration of the array.

2 Related Work

While the network intrusion detection problem is well defined, the performance measures are not. There are many approaches to the intrusion detection problem, and each has its own strengths and weaknesses. We assume in this paper that the two most important measures are throughput and area efficiency. Throughput, because the filtering needs to keep pace with the line rate. Area efficiency is important because there are a few thousand patterns that need to be matched. We propose that reconfigurability is the third vital metric: the pattern database does change from time to time, and the downtime and processing time for reconfiguration should be low. Hardwired approaches [1, 2, 3, 13, 22] require place and route of the entire device, and then a short bitstream reconfiguration time. An approach offering online reconfiguration, with little or no delay to the data stream, is ideal, and offered by our approach, first developed in [4], and a hash-based approach in [16].

Snort [20] is a current popular option for implementing intrusion detection in software. It is an open-source, free tool that promiscuously taps the network and observes all packets. After TCP stream reassembly, the packets are sorted according to various characteristics and, in the worst case, are string-matched using the Boyer-Moore algorithm against rule patterns. However, the rules are searched sequentially on a general-purpose microprocessor. This means that the IDS is easily overwhelmed by consistently high rates of packets. The only option given by the developers to improve performance is to remove rules from the database or allow certain classes of packets to pass through without checking. A hacker tool can take advantage of this weakness of Snort and attack the IDS itself by sending worst-case packets to the network, causing the IDS to work as slowly as possible. The eventual uninspected packets that result provide an opportunity for the hacker. Clearly, this is not an effective solution for maintaining a robust IDS.

In [12], a CAM-powered software/hardware IDS is explored. A Content Addressable Memory (CAM) is used to match against possible attacks contained in a packet. Instead of matching one character per cycle, the tool uses CAM hardware to match the entire pattern at once as the data is shifted past the CAM. If a match is detected, the CAM reports the result to the next stage, and further processing is done to derive a more precise rule match. If a packet is found to match a rule, it is dropped or reported to the software IDS for further processing. This requires $O(mx)$ CAM memory cells and a great deal of routing for each m-character layer of x rules. Unfortunately, though, because matching is done in parallel across all rules and across all characters in one cycle, this sort of implementation requires a great deal of logic. While this does provide $O(n + m)$ worst-case rule matching time, it does so at the cost of a large amount of hardware.

In [9] a novel hashing mechanism utilizing Bloom filter is discussed. Their implementation of a hashing-table lookup using a moderate amount of logic and external or internal memory is an effective method to search thousands of strings for matches in a single pass. Rules can be added by changing only some data in the memory. Rules can be removed by maintaining some information in a software-based host. Neither requires reprogramming the FPGA. The filter is powerful but somewhat hindered by a tradeoff between the false positive rate and the number of rules contained in given memory size.

In [1, 2, 3, 5, 22], hardwired designs are developed that provide high area efficiency and high time performance by using replicated hardwired comparators in a pipeline structure. The hardwiring provides high area efficiency, but are difficult to reconfigure. Hardwiring also allows a unit to accept more than one byte per cycle, through replication. A bandwidth of 32 bits per cycle can be achieved with four hardwired comparators, each with the same pattern offset successively by 8 bits, allowing the running time to be reduced by 4x for an equivalent increase in hardware. These designs have adopted some strategies for reducing redundancy through pre-design optimization. The work in [6] was expanded in [5] to reduce the area by finding identical alignments between otherwise unattached patterns. Their preprocessing takes advantage of the shared alignments created when pattern instances are shifted by 1, 2, and 3 bytes to allow for the 32-bit per cycle architecture.

3 Our Approach

Our novel approach to runtime adaptability uses a pipelined, two-comparator, buffered implementation of the Knuth-Morris-Pratt algorithm (KMP)[14] to implement high-performance pattern matching, while yielding a unit design that is highly area efficient. Our architecture enables high-throughput, easily configurable intrusion detection for a variety of hardware platforms (FPGA or ASIC). Unlike other state-of-the-art architectures, our approach does not use hard-wired circuitry to implement pattern matching on an FPGA or ASIC, but uses re-programmable memories storing patterns and pre-compiled jump tables to provide exceptional flexibility.

By minimizing the number of comparators required to match a new input character each cycle, we use fewer hardware resources than other approaches. After finding the maximum buffer size requirements through careful analysis of the algorithm, we can produce a pattern matching unit that uses few FPGA resources, allowing more units to be integrated onto a single chip. By allowing only one-way communication between neighboring units, the architecture is suited, by design, for use in a linear array and in grids of units.

A significant contribution of this paper is a demonstration of the maximum size buffer required to implement a string matcher capable of accepting a character from the input in each cycle without resorting to k parallel comparators, where k is the pattern size, or an n element buffer, where n is the input stream size. The architecture that enables this is a buffered string matching system implementing a KMP-like pre-computation algorithm utilizing two comparators. This allows a matching unit to accept one character each cycle into a buffer of size $\log_\phi k$, where $\phi = \frac{1+\sqrt{5}}{2}$.

By buffering the input and guaranteeing that one character can always be accepted into the buffer from the stream, our work allows a systolic array of pattern matching units to be created. The unit-level buffering is a key idea, because without buffering whenever an individual unit stalls the input to recover from a failing comparison, the entire system must stall as well. This is unacceptable, as it would reduce the system throughput significantly as each individual unit contributed its own stalls to the system. The systolic array reduces the input fanout and the overall interconnect distance by allowing units to be regularly arranged on an FPGA without long-wire interconnects. Systolic designs have the benefit of having no global connections, except for the clock signal. The details of our architecture will be covered in Section 4 and we will show in a detailed proof in Section 6 that the buffer will never drop input characters, regardless of the input or pattern.

3.1 Overview of KMP

KMP, developed by Knuth, Morris, and Pratt [14] utilizes a pre-computed table to prevent redundant comparisons, reducing the worst-case running time from $O(kn)$ to $O(n+k)$. The pre-computed table, or π-table[2], tells the automata which pattern character to match against next. The function $f[q]$ gives the length of the largest number of characters in the prefix of the pattern P that match the suffix ending at $q-1$. The $\pi[q]$ function gives a more refined version of $f[q]$ in that when $P[q] = P[f[q]]$, the optimization $\pi[q] = \pi[f[q]]$ can be made to reduce redundant jumps.

$$\begin{aligned} f[q] = \max\{ \quad & j : j < q \text{ and} \\ & P[1 \ldots j-1] = P[q-j+1 \ldots q-1]\} + 1 \end{aligned} \tag{1}$$

[2]We use the π-table as equivalent to the usage of the *next* function in [14]; other references, such as [8], compares against the $(\pi+1)$th character whereas the original KMP compares against the πth pattern character

$$\pi[q] = \begin{cases} f[q], & \text{if } pattern[j] \neq pattern[f[q]] \\ \pi[f[q]], & \text{if } pattern[j] = pattern[f[q]] \end{cases} \tag{2}$$

That is, the π-table tells how much of the beginning of the pattern has already been matched at any position in the pattern. Of course, this only affects the system when the pattern has repeated strings; a pattern such as "abcdefg" would have no useful information in the π-table, while "abaab" would have useful information because the beginning of the pattern shows up later in the pattern. After the $\pi[q]$ definition is applied, there is some post-processing on the π-table, discussed later in this section.

The π-table below is for the worst-case pattern, a Fibonacci string [14]. $\pi[1] = 0$ (this is true for all patterns), meaning there is no possible match and the input pointer should be advanced. $\pi[4] = 2$, meaning the next character comparison is against $P[2]$ = 'b'. If this fails, the next comparison is against 'a'. The second comparison of 'a' is unavoidable because KMP is *historyless*, meaning that the system cannot remember what it has compared earlier. An important point is that hard-coded Finite Automata-based implementations can work more efficiently than a memory-based approach because the FA can compare all of the possible transitions in parallel, while the RAM-based approach must sequentially compare the incoming character against (potentially) several pattern characters.

q	1	2	3	4	5	6	7	8	9
P[q]	a	b	a	a	b	a	b	a	a
$f[q]$	1	2	1	2	2	3	4	3	2
$\pi[q]$	0	1	0	2	1	0	4	0	2

Our approach to intrusion detection uses a modified version of the KMP algorithm and matching architecture (Figure 1) optimized for running on a systolic array. The Knuth-Morris-Pratt algorithm (KMP)[14] is a sophisticated approach to string matching, providing $O(n + k)$ in the worst case. The exact worst case running time is $2n - k$.

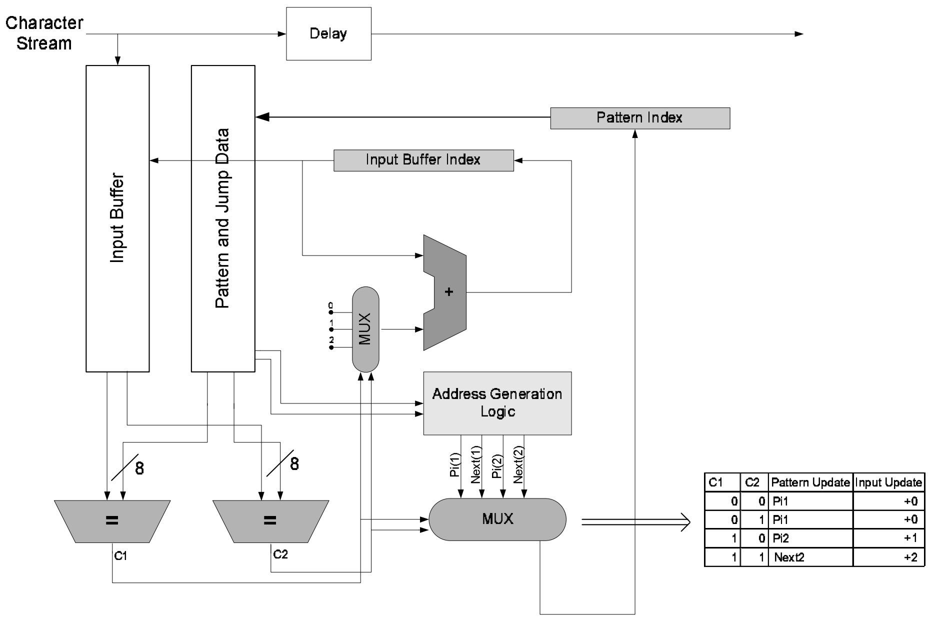
KMP achieves these speedups by creating a table of allowable/possible matches, preventing redundant comparisons. The main drawback of KMP is the slightly more complicated control circuitry and lack of parallelism. Data cannot be shifted one position at a time as in the naïve or wide parallel shift-and-compare approaches.

There are string matching algorithms that have better average-case running times than KMP, but there is no single comparator algorithm with better worst-case characteristics. Because the systolic array depends on consistent, non-fluctuating consumption of input characters, the KMP algorithm is the ideal solution for our needs. This is vital to our architecture, but it is important to network security in general because an attacker might attempt to cause the IDS to drop packets by flooding it with specially designed packets. If a IDS is overwhelmed by traffic, most configurations will allow some packets through without screening, or subject them only to a cursory examination. This provides an opportunity for an attacker to subvert the system. This type of attack is shown to be useless against our design, due to the guarantee of performance in Section 6. Our contribution to the field of KMP research is to prove a worst-case buffer size requirement such that a string matching unit can accept an input character each cycle and end in $(n + \log_\phi k)$ cycles, where $\phi = \frac{1+\sqrt{5}}{2}$.

4 Architecture

The KMP algorithm uses a single comparator and can move forward at most one character in the input string per cycle. However, in some situations, the input will be stalled while the comparator makes additional comparisons against the same character. *We extend KMP by using two comparators, then prove in Section 6 that with a very small buffer we are guaranteed that a character can be accepted from the input string during every cycle.*

The general architecture is shown in Figure 1. Here we have two comparators feeding a multiplexer that determines the new index values for the pattern and input memories. The



C1	C2	Pattern Update	Input Update
0	0	Pi1	+0
0	1	Pi1	+0
1	0	Pi2	+1
1	1	Next2	+2

Figure 1: General architecture of two-comparator design

input buffer is preloaded with the first $k/2$ characters. This allows the gap between the characters read from the buffer and the characters entering the buffer to vary as matches occur.

If first character match is not successful, the result of the second comparison is discarded. However, when the first comparison does match, the result of the second comparison is relevant. The two comparators allow the system to use input characters faster than they enter into the buffer. This is important because when the comparison fails, the buffer stalls on the current character until all possible alternate alignments of the pattern are compared against. During this time, data continues to enter into the buffer, as the input stream cannot be stalled.

Figure 2 illustrates the interaction between the incoming stream and the matching unit. Because the data arrives in the buffer at a regular rate, there is a limit to the number of characters that can be taken out of it. If the read pointer exceeds the write pointer, an underflow condition occurs. This is illustrated by the darkened area above the upper diagonal. The more difficult problem is overflow, captured by the darkened area

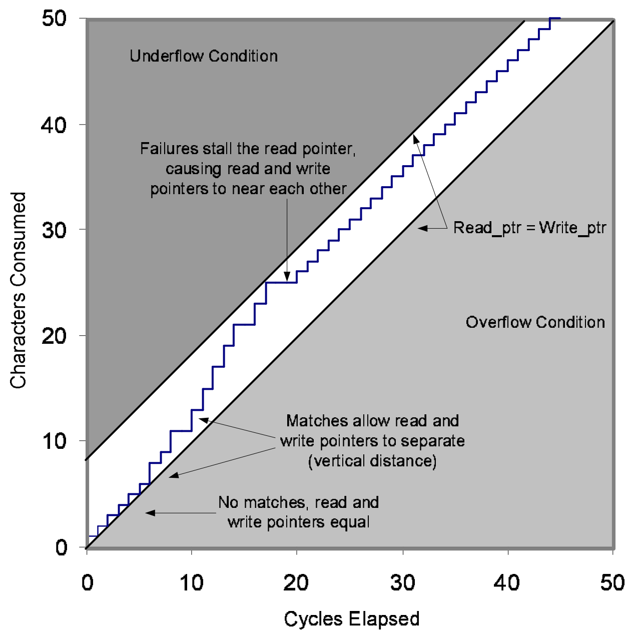


Figure 2: Separation of read and write pointers in the input buffer. As time progresses to the right, the number of consumed characters moves upward. The rate at which is moved upward is governed by the state of the matching system. The system prevents the FIFO from running out of new characters (underflow, as delimited by the top diagonal) and our proofs guarantee that overflow will not occur (overflow, delimited by the bottom diagonal).

below the lower diagonal. In the overflow situation, the read pointer falls behind the write pointer and the buffer runs out of space. In an IDS system, overflow would allow attacks to pass undetected. In the figure, we see the input (read) pointer and the incoming data (write) pointer as they vary during a pattern matching operation. The buffer initially loads, producing the initial separation between the upper and lower lines. When there are no successful matches, the input pointer proceeds diagonally upward. When the input matches the pattern, the write index and the read indexes get closer together (in Figure 2, this is represented by any vertical movement), until a failing comparison occurs. This causes the input pointer to stall, represented by a horizontal line in the graph. If, by some combination of inputs the read and write pointers near, implying a nearly empty character buffer, the unit will only utilize one of the comparators, slowing the processing of the buffered packet to one character per cycle. This matches the speed of the stream input to the buffer. By design, there is no situation where the read and write buffers can actually wrap around the buffer and collide, falling into either of the gray (overflow and underflow) areas. Buffer overruns or underruns would cause input characters to be lost and is undesirable. We prove that our design prevents this from happening in the next section.

Beyond the basic extensions to the KMP algorithm and our buffered systolic array design, we also have designed a C-slowed version of the individual units. After careful timing analysis of the design, it became clear that the large contributions to the period can be split into two independent sections; the memory access and comparisons, and the multiplexing of incremented pointers. After the initial memory read, the memory essentially stands idle for the remainder of the cycle, and the combinational logic is idle until the completion of the memory read. Given this situation, pipelining is a general technique that is an obvious choice for increasing the speed of the system, but at first glance it seems unworkable. Pipelining generally does not make sense for single-character-oriented string matching architectures because they need to update the pointers each cycle based on results from the current cycle.

The solution is a C-slowing technique [15]. Similar to a fine grain multi-threaded architecture, the two pattern matching units essentially stay out of phase with each other. The two units share the same hardware except for the memory segment storing the two different patterns. This is an exciting approach because the combinational logic and input buffers occupy more than 75% of the utilized area. Adding a few more state registers and doubling the pattern memory produces only a small increase in the total resources required, from 50 slices to 57 slices for a 16 character pattern and 65 slices for a 32 character pattern. Because of the simplicity of the pipeline, the two units can flip back and forth between the memory and combinational pipeline stages using only the pipeline registers, requiring no extra control logic. While this strategy increases the latency for a single pattern, each unit provides matching for two patterns with only a small increase in area. Any increase in clock frequency due to the pipelining translates directly to increased total performance. By pipelining the units, we increased the place-and-route clock frequency from 221 MHz

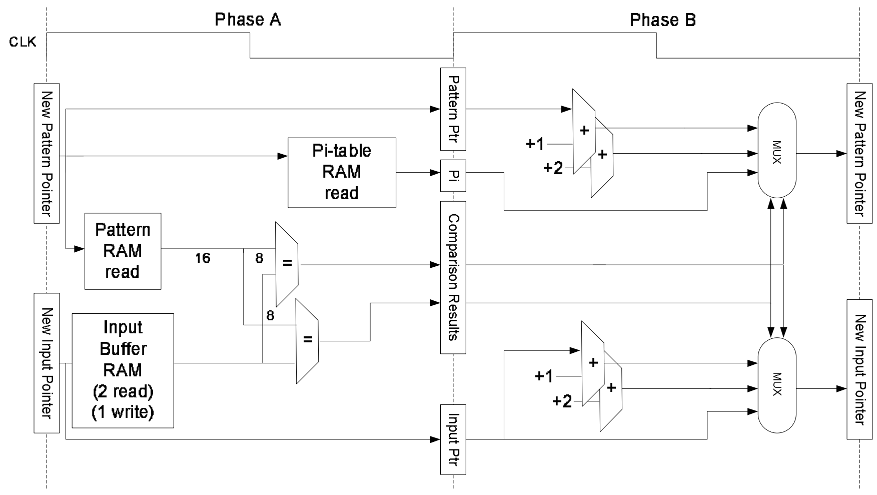


Figure 3: Pipelined architecture of two-comparator design

to 285 MHz.

5 Operational Examples

In this section, we will explore a sequence of cases, working up to the proof of the general behavior in the next section. The cases are illustrated in Figure 4. In the figure, f signifies a failing match attempt, m is a successful match, and d is a 'don't care' character which does not figure in the current comparison cases. In each section of the figure, the hat ($\hat{}$) shows which two characters are being compared in parallel in each cycle. In the context of a real example, Figure 4.c illustrates a situation in which the input matches a pattern up to the third character. For instance, the pattern *abcdef* matched against "abrst" will find the failing comparison on the second cycle, as *ab* are both compared in the first cycle. Thus, the number of "matching cycles" is two. Because the matching failed on the third character, the number of "characters consumed" is three. The number of "failing cycles" is determined by the behavior of the algorithm. Within the first three characters in any pattern, the number of alternate alignments is less than or equal to one. For instance, a pattern "abc" has only one required alternate comparison, the first character. After failing the comparison for 'c', the first character 'a' compared and then a new input character is

	Failure Position		**Matching Cycles**	**Failing Cycles**	**Characters Consumed**
a.	First Character	$\overparen{f - d} - d - d$	1	0	1
b.	Second Character	$\overparen{m - f} - d - d$	1	1	2
c.	Third Character	$\overparen{m - m} - \overparen{f - d} - d - d$	2	1	3
d.	Fourth Character	$\overparen{m - m} - \overparen{m - f} - d - d$	2	2	4

Figure 4: Character failures; 'm' is a match, 'f' is a failure, 'd' is a "don't care". By comparing two characters in each cycle, the system can ensure that the number of characters consumed is equal to or larger than the sum of number of clock cycles spent successfully matching the input plus the cycles spent stalling while the matching failed.

processed. Thus, the number of cycles before a new character is processed (the number of "failing cycles") is one.

In Figure 4.a, the first character comparison fails. Regardless of the other characters in the string, a failing first character always advances the input string pointer. Also, it causes the same pattern character to be compared in the next cycle. The result is that we advance the input index by one character in one cycle. This fulfills our requirement for ending up no deeper in the pattern after any sequence of operations. In the following text, q^* is the index of the first character in the stream to be compared, and q is the index of the last character to be compared.

The second case, shown in Figure 4.b, is a bit more complicated. Here, we compare the first ($P(1)$) and second ($P(2)$) characters of the pattern with the first and second characters of the input, $i(q^*)$ and $i(q^*+1)$ in the first cycle, where $j = q^*$. The first character matches ($i(q^*) = P(1)$), and the second character fails ($i(q^* + 1) \neq P(2)$). Because only $P(1)$ matched, we advance the input by one byte ($j = j + 1$), but start the pattern from $P(1)$ again.

The next case is when both of the first two characters match (q^*, $q^* + 1$)and we fail on the third character $P(3)$ as in Figure 4.c. We gain a cycle in the comparisons $i(q^*) == P(1)$ and $i(q^* + 1) == P(2)$, as we advance the input by two bytes in one cycle. When $P(3)$ fails in the second cycle, the only legal shift for $q > 2$ (as shown in Lemma 2 in Section 6) is of at least two characters: $\pi[3] \leq 1$. As we are looking for the worst case path to $q^* = 0$, we take the path where each comparison fails. Given a failing comparison at $(q+2) = 3$, the next comparison is at $\pi[3] = 1$. If $P[\pi[3]] = P(1) \neq i[q+2]$, then we advance to the next input character. Starting at $q = 0$ and advancing to $q = 2$, then failing back to $q^* = 0$ requires a total of three cycles for three advances of the input pointer.

The fourth and final case we will investigate before proving that all cases meet the non-decreasing buffer gap condition is a failure in the fourth position, $P(4)$, as shown in Figure 4.d. This is very similar to the third, except during the second cycle the input advances by one, and because of Lemma 2, $\pi[4] \leq 2$, thus allowing the failure sequence $\pi[4] = 2$, and $\pi[2] = 1$, and $\pi[1] = 0$. In comparison with the previous case, there is an additional jump, thus requiring a total of four cycles, but the input is advanced as well.

6 Buffer Size Requirement

The general KMP architecture is difficult to utilize in a pipelined grid architecture of matching elements because the algorithm can stall and suspend the processing of new characters in unpredictable ways. Buffering the entire input sequence is one solution, but unworkable as input sizes become large. Theorem 2 shows that the maximum buffer size required for our architecture is only $\log_\phi k$, where k is the pattern size.

Assumptions: We receive a single byte of input data each cycle. Total input length is n bytes. Pattern length is m bytes. The pattern-matching element contains a RAM buffer which can provide simultaneous reads from two different addresses and write to a third address in one cycle. Note that the discussion of the proof refers to Figure 1; the pipelined architecture in Figure 3 simply stretches a single state update across two cycles but complicates the counting in the proof.

Definitions:

to consume a character: to advance the pointer of the input buffer, ending the comparisons against the current input character

cycles: a traversal between q^* and q and then back to q^* or earlier, consuming $q - q^* + 1$ characters from the buffer. Not to be confused with clock cycles

sub-cycles: failing comparisons that lead to successful comparisons at positions larger than 1; that is, a cycle that ends because a prefix of the pattern matches the current suffix of the input

cycles in: the number of clock cycles until a failing comparison occurs, including the clock cycle that contains the first failing comparison

cycles out: the number of clock cycles after a failing comparison until the next increment of the input pointer, not including the first failing comparison or the cycle after the pointer is advanced. This number is equal to the π-transitions that require comparisons

architecture: the architecture in question is the string matching unit in Figure 1

character position: the position of a character in a string, the leftmost character being character 0 and the rightmost character being $n - 1$

In Lemma 1, we show that the worst case for the algorithm are failures that cause full-cycle traversals of the π-table, that is, where the pattern pointer starts and ends at the first pattern character.

Lemma 1

If the algorithm correctness is maintained in full-cycle failures, the algorithm is correct in sub-cycle failure situations.

Proof: Failures that cause sub-cycles, that is, failing comparisons that lead to successful comparisons at positions larger than 1, require fewer clock cycles than full-cycle traversals. The relation $q - \pi[q] \geq 2$ for $q > 3$ holds except for the case detailed in Lemma 2. Because each π-transition skips at least one character for $q > 3$, and in general makes much larger skips due to the logarithmic increase in the number of steps as a function of position, the situation is always better than at positions deeper in the pattern. $\square$

Because full-cycle traversals are the worst-case situation, if the condition is satisfied for all q when $q^* = 1$, we can be satisfied the system will work for intermediate (sub-cycle) cases.

Lemma 2 proves that a pattern with the first $(n-1)$ characters identical followed by a different character is the only pattern that can cause a transition that moves one element backwards. Because this type of pattern can cause the architecture to fail, we seek to understand it fully and then eliminate it.

Lemma 2

Transitions of the form $\pi[q] = q - 1$ for $q < n$ can exist for a pattern, P[1,2, …, n-1, n, …], where $P[q] = \alpha$ for $q = 1 \ldots n$ and α is some pattern character.

Proof: The π-table is defined as the maximum j less than q such that $P[1 \ldots j-1] = P[q-j+1 \ldots q-1]$

In the case $\pi[q] = q - 1$, j = n - 1. Substituting for j,
$P[1 \ldots (n-2)] = P[2 \ldots (n-1)]$. $\square$

Lemma 2 proves that all characters $P[1]$ through $P[n-1]$ must be identical to produce single-character π transitions. Moreover, n must be at least 2. This is the only case where sequences of single-character π-jumps (more than one single-character jump in a row) may occur. Fortunately, the original KMP algorithm [14] prevents sequences of repeated failing comparisons, and thus, this case is impossible.

While characters $P[1]$...$P[n-1]$ must be identical, nothing is guaranteed about $P[n]$, which may be a different character. This causes an interesting situation, as $\pi[q] = q - 1$ can exist for only the nth character. For $\alpha_1 \neq \alpha_2$, $n = 2$, the situation is identical to

Case b in Section 5. For $n > 2$, there must have been at least one pair of characters successfully matched in a single cycle. This gives the system the space to make a single-byte π-jump. Due to the original KMP algorithm, only a pattern having identical leading $(n-1)$ characters, followed by a different nth character, can cause $\pi[n] = n-1$, and thus there will exist only one single-character π-jump in any cycle. This is acceptable and does not lead to buffer overruns.

In the proof of Theorem 1, we show that the architecture accepts, on average over certain controlled time intervals, at least one character per cycle.

The buffer enqueues characters to use when the input pointer stalls. The input pointer stalls when a character comparison fails and some number of additional comparisons have to be made against the same input character. This proof is important because we need a guarantee that the buffer in the architecture will never run out of space during an input stall caused by non-matching input characters. While trivial with a n-sized input buffer, we provide only a $k/2$ buffer for each matching unit.

Theorem 1 proves that, for any sequence of matching and failing the number of cycles required to advance into the pattern to position q and then fail out to the starting character or earlier (q^*) is less than the number of characters processed from the input buffer in the same number of cycles. That is, the system advances farther when it matches than it gets behind when it fails. In order for this to work, the system has to take advantage of the two comparators provided and "get ahead," or decrease the number of unprocessed characters in the input buffer. *When the failure occurs and the input pointer stalls for several cycles, the number of unprocessed characters will increase. We call this the "non-decreasing buffer gap," meaning that the gap between the read and write pointers cannot have decreased at the end of a cycle.* If this condition is satisfied, no possible input sequence will cause the system to not accept a character each cycle, or fail to fully process each character.

Theorem 1:

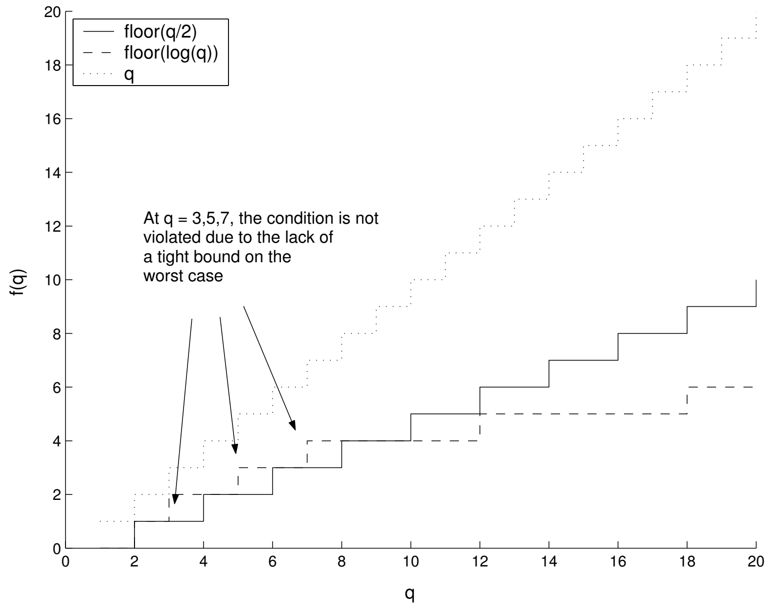


Figure 5: Using $q^* = 1$, we graph floor(q/2), floor(log(q)), and q, which correspond to C_c, $c_{matching}$, and $c_{failing}$, respectively, in the formula $C_c \geq c_{matching} + c_{failing}$

For any sequence of successful comparisons followed by a sequence of failing comparisons where the initial character position at t_i is greater than or equal to the final position at t_f and the number of characters removed from the buffer at $t_t = consumed_t$, the following relation holds:

$$C_c = \frac{\sum_{t=t_i}^{t_f} consumed_t}{t_f - t_i} \geq 1$$

Proof:

First, we algebraically create an equivalent relation. We multiply through by $t_f - t_i$, then substitute the cycle equivalence $t_f - t_i = c_{matching} + c_{failing}$, where $c_{matching}$ is the number of system cycles in which both comparisons succeeded and $c_{failing}$ is the number of system cycles spent in which at least one comparison failed. This yields

$$\sum_{t=t_i}^{t_f} (consumed_t) \geq c_{matching} + c_{failing}$$

For convenience, we define $\sum_{t=0}^{t_f} (consumed_t)$, the total number of characters consumed from cycle t_i to t_f, as C_c.

We count the number of cycles starting when the system requests input character i_{j+q^*} (using the input index j for generality) and pattern character P_{q^*}, successfully matches forward until input character i_{j+q} and pattern character P_q. This requires $c_{matching} = \lfloor \frac{q-q^*+1}{2} \rfloor$ clock cycles, counting from 1. If the pattern is not matched, a failure will occur. The worst-case number of cycles *until the next input character is requested from the buffer* occurs when the pattern pointer jumps all the way to the 1st pattern character. In this case, the original KMP authors [14] proved that the worst case number of π-transitions, and equivalently, the number of stalling cycles where no input characters are consumed, is $\log_\phi q^*$, where $\phi = \frac{1+\sqrt{5}}{2}$. What we need, though, is the number of cycles until we fail to a position less than or equal to the starting position q^*. This allows us to prove that sub-cycles of matching and failing also conserves the buffer appropriately. This is simple enough because the KMP algorithm is history-less, that is, only the current state matters, allowing our accounting to start and end where we like, namely, q^* and q.

The upper bound on the number of failing cycles $c_{failing}$ between q^* and q is the difference between the upper bound of jumps for q and the lower bound for q^*.

For the worst-case pattern, the upper bound $max\_jumps(q)$ on the number of cycles from q is $\log_\phi(q)$.

Lemma 1 showed that the worst case for the cycles/characters consumed condition is actually where $q^* = 1$. This simplifies the analysis of the condition by allowing us to set $min\_jumps(q^*) = 0$, yielding

$c_{failing} = max\_jumps(q) - min\_jumps(q^*) = \log_\phi(q) - 0$

The final component required is the number of input characters processed. Because we no longer require input character j at the end of our accounting, we can define $C_c = q$ - q^* + 1. Putting the pieces together, we get:

$q - q^* + 1 \geq \lfloor \frac{q-q^*+1}{2} \rfloor + \lfloor \log_\phi(q) \rfloor$

When q and q^* are large, say, above 8, there is no question that the condition is satisfied (see Figure 5), because the number of π-transitions decreases logarithmically with the position in the pattern. Pattern positions less than 8 are more challenging in analysis. In particular, for $q < 8$ because $\log_\phi q > \frac{q}{2}$.

C_c	$C_{matching}$	$C_{failing}$	Successful?
1	1	0	T
2	1	1	T
3	2	$\cancel{2}$ 1	$\cancel{F}$ T
4	2	2	T
5	3	$\cancel{3}$ 1	$\cancel{F}$ T
6	3	3	T
7	4	$\cancel{4}$ 3	$\cancel{F}$ T
8	4	4	T
9	5	4	T
10	5	4	T

Figure 6: Correctness table based on worst-case bounds, which are found in Figure 5 to overstate true worst-case numbers for some low values of q. In cases where this affects the correctness of the analysis we fix the number of failing cycles and correspondingly adjust the success flag.

We note that 3, 5, and 7 are not successful (as also indicated in Figure 5), based on the worst-case formulation provided by Knuth, Morris, and Pratt in their original paper [14]. Unfortunately, the worst case bound is not as tight as required. We have already shown that the q=3 case maintains the buffer requirements. We can look at the Fibonacci string, proved to be the worst-case possible pattern in the original paper, and see that the worst-case bound overstates the actual worst-case in these important cases:

q	1	2	3	4	5	6	7	8	9
$P[q]$	a	b	a	a	b	a	b	a	a
$\pi[q]$	0	1	0	2	1	0	4	0	2

Inspecting q = 3, 5, and 7, we can count the number of transitions before we can move to the next input character. $\pi(3) = 0$, so the number of cycles for a failure is actually zero.

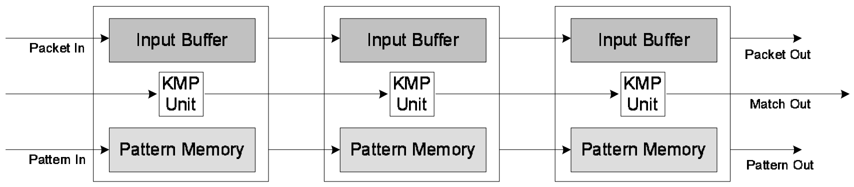


Figure 7: Linear array of matching units. Each input buffer is of size $k/2$, each pattern memory is of size k.

For $q = 5$, $\pi(5) = 1$, thus the number of transitions is 1. For $q = 7$, $\pi(7) = 4$, $\pi(4) = 2$, $\pi(2) = 1$. Adding, there are a maximum of three transitions for q = 7. Substituting our new, accurate values into the table, we see that each "worst-case" violation is valid. This proves that our architecture requires equal or fewer cycles to compare the pattern against the input than the number of input characters advanced during the same period, or, equivalently, over any sequence of matches followed by a sequence of failures, the average number of characters removed from the buffer is at least 1. $\square$

Theorem 2: Using two comparators, our KMP architecture requires a buffer of only $k/2$ characters, where k is the pattern length to support a one character per cycle throughput, regardless of pattern or input sequence.

Proof: The two comparators allow two characters to be accepted from the buffer during each cycle. It is thus possible to match an entire k length pattern in $k/2$ cycles. In Theorem 1, we proved that at least one character is removed from the buffer in every clock cycle, on average. By Lemma 2, the maximum number of consecutive cycles in which the system can remove zero characters from the buffer is equal to $\log_\phi k$, where $\phi = \frac{1+\sqrt{5}}{2}$.

The $\log_\phi k$ cycles in which characters are not removed from the buffer have already been accounted for in the analysis of Theorem 1. That is, enough characters have been removed from the buffer to allow for $\log_\phi k$ cycles to pass without removing data. However, because characters continue to be added to the buffer regardless of the removal rate, the buffer needs to be large enough to allow $\log_\phi k$ characters to enter without causing overflow.

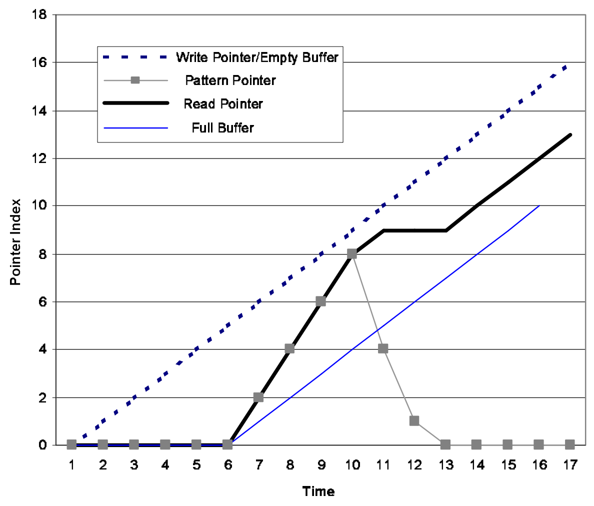


Figure 8: Various relevant pointers during a simulation of the two-comparator algorithm. The input sequence matches the 16 character pattern for the first 15 characters of the worst-case Fibonacci string pattern. The pointers into the buffer do not wrap at the buffer boundaries for clarity.

Thus, the maximum required size of the buffer is $\log_\phi k$. $\square$

The number of clock cycles required to completely match a pattern starting from the first position, $\lceil k/2 \rceil$, is greater than the precise buffer size required, $\log_\phi k$, for $k > 8$. The number of clock cycles for failing is always less than or equal to the number of cycles for matching, for patterns larger than eight characters. We could limit the buffer to the number of cycles for failing comparisons, but the larger buffer allows the architecture to offer on-the-fly reconfiguration as well as the ability to do pipelining via C-slowing.

Next we present the results of a functional simulation of a contrived input against a 16-character pattern. The pattern used is the Fibonacci string, thereby exercising the worst-case situation for the buffer. Various relevant pointers during a simulation of the two-comparator algorithm are shown in Figure 8.

During the first six cycles, the $\log_\phi k$ size buffer is loaded, and then the algorithm starts. In the simulation, the input sequence matches the first 15 of 16 characters in the pattern. Because of the two comparators, the system only requires 8 cycles to match to the end of the pattern. When the failure occurs in the 8th cycle, the read pointer stalls for several cycles (in Figure 8, the stall is manifested as a horizontal line), approaching the point at which the read pointer is equal to the write pointer mod $\log_\phi k$. If the read pointer was to collide with the write pointer, data loss would occur due to overflow. However, before the buffer overflows the pattern pointer reaches the first character, and the read pointer resumes incrementing.

7 Results

In this section we present our results and define some performance measures to compare against the competing architectures. We targeted the Virtex II Pro xc2vp4 device with -7 speed grade. We use the Xilinx ISE 5.2i and Mentor Graphics ModelSim 5.7 development tools.

Implementation	Total Work	Input Bits	Freq (MHz)	Throughput
USC(no pipelining)	$2n$	8	221	8*freq=1.8 Gb/s
USC(pipelined)	$2n$	8	285	8*2 * freq/2 = 2.4Gb/s[3]
Los Alamos[12]	kn	32	68	32*freq=2.2Gb/s
Wash U. - DFA[16]	fn	8	80	8*freq=0.640Gb/s[4,5]
Wash U. - Bloom[9]	kn	8	100	8*freq=0.8Gb/s[4]
UCLA RDL [5]	kn	32	100	32*freq=3.2 Gb/s
U/Crete[21]	kn	32	340	32*freq=10.8Gb/s

Table 1: Throughput and total work performed, e.g., the total number of byte comparisons made.

In Table 1, we compare the total work performed for various implementations analytically, and its relation to the number of input bits and throughput. In Figure 9 we illustrate the number of byte comparisons required for detecting the matching string at the end of a 32 or 1600 byte input string. We find that hardwired, fast 32-bit implementations can perform remarkably well in terms of throughput. However, they do not offer the reconfig-

[2]Two comparators use the same hardware due to the pipelining

[3]Each unit in this design advances by one byte in each cycle, but the system is composed of four units working in parallel, increasing the total throughput to at least 2.4Gb. We consider only a single unit.

[4]Unit size determined by converting block RAM to equivalent distributed RAM structures for area comparisons

[5]Variable f is the number of comparisons made for each step in the FSM

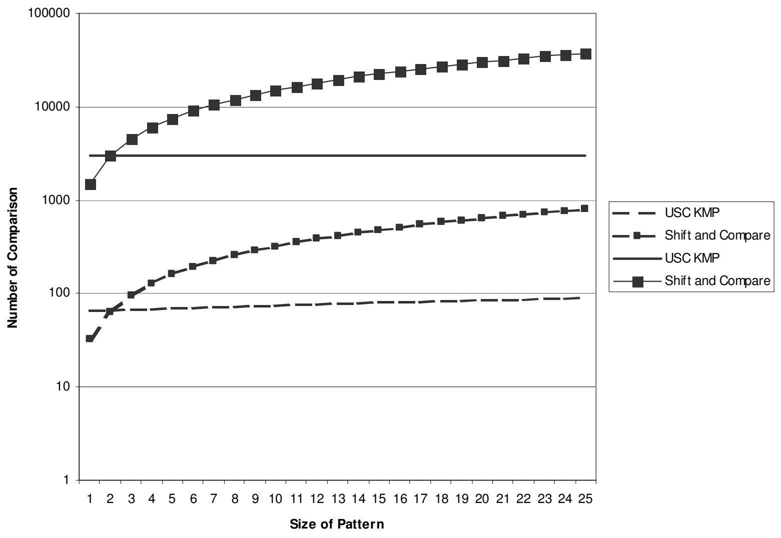


Figure 9: Overall number of comparisons (work) for a variable length search string over a 32 or 1500 (TCP/IP MTU). The k length pattern is matched starting at position $n - k$. Work for shift and compare designs is far greater, but more comparisons are made per cycle.

urability that we can offer, nor can they compete area-wise. An important consideration in our design strategy is in byte parallelism. Increasing input size by p times, while increasing the throughput by p times, increases the number of comparators by p^2 times. This work efficiency allows our architecture to maintain a larger number of matching units, even though our architecture is not hard-wired. Moreover, our architecture, due to the systoic array of elements and careful buffer design, can provide on-the-fly reconfiguration of a new ruleset at the same time a new packet is loaded into the buffers. Because our performance metrics measure both throughput and area efficiency, we consider exchanging a linear increase in throughput for a quadratic increase in area to be not in our best interest. This leaves us with state machine based implementations such as [16], and creative architectures based on traditional string matching algorithms such as KMP.

Many previous papers on network string matching have provided fast throughput on a few patterns but have not been able to scale well because of fanout delays and the complexity of their matching units. This puts a severe limitation to their application in real network security applications, where hundreds if not thousands of rules must be simultaneously matched at line rates. Other designs [3, 6, 21], with their small, simple pipelined and hardwired design, have come closer to producing efficient designs as the they can fit one hundred or so of the most common patterns on a device. Unfortunately, as the number of pattern matching units increases the system speed drops dramatically, and, as parallelism increases, the area requirements increase quadratically.

In order to scale the design, we have developed a Relationally Placed Macro (RPM) which allows the performance of the unit to be reliable duplicated in volume on large devices. It also provides predictable area, faster place and route, and can more easily meet timing when scaled.

Without the RPM, performance degrades noticeably after more than 50 units or so are placed on a device. This scaling issue is a problem that is less noticeable in the shared decoder strategy we and others have pursued [1, 5, 7, 22]. One novel approach we have considered is a hybrid architecture that utilizes the prefiltering architecture we advanced in [1] and the KMP units from this paper. In the prefiltering architecture, the filter has exceptionally small area per character requirements, but can filter a high data rate input stream into a small subset of potential matches. It is a "prefilter" because the false-positive rate is fairly high. A small bank of KMP units are configured on-the-fly as needed by the prefilter. The KMP units serve as an exact-match backend. In this way, the exact-match run-time reconfiguration abilities of the KMP architecture are tapped along with the scalable performance characteristics of the prefiltering architecture.

8 Reconfiguration of Arrays of Units

Our design provides reconfiguration that proceeds at the same rate as the flow of the input packet into the buffer. One troubling aspect of other designs in the field [6, 13, 16, 21] is the amount of time required to change the patterns in the unit, usually requiring some place-and-route effort and partial reconfiguration. While much work has been published on effective strategies for this [2, 10], the schemes are never as effective as a system with architectural support for on-the-fly reconfiguration.

Recent viruses have infected hundreds of thousands of hosts in the first few minutes of activity. Given the power of modern worms and other hacking attacks, waiting for a complete place and route of a hard-wired design while a network is overrun is not ideal. Thus, fast reconfiguration is useful in dynamic network environments. In other situations, fast reconfiguration is necessary to support the entire rule set. For instance, in the Snort ruleset [20], rules are sorted into categories based on port number and protocol. It is challenging in current systems, including our design, to support all of the possible rules in a single device. By allowing the system to be reconfigured as required for different types of packets, the system can more effectively meet the needs of practical network security.

The architecture of our system is dependent on several small banks of memory in each unit that hold the patterns and jump tables. These buffers give us the time required to load new patterns.

There are two key architectural requirements for on-the-fly reconfiguration. One is that there is enough space in the buffer to make room for the reconfiguration delay. The input stream could be delayed during reconfiguration to allow for the new patterns be loaded. However, since the pattern being changed is no longer relevant there is no reason the system should not stop monitoring and hold the pattern pointer at zero during reconfiguration. Assume, then, that the currently buffered characters will be lost. The second requirement is the ability to actually utilize the buffers as a $k/2$ delay element without adding significant hardware. The buffer pointer that addresses the current pair of input characters under comparison does not consistently increment, and thus it is not suited to our needs. However, the incoming character (write) pointer increments once per cycle. Thus, it can be used without changing the original buffer design.

By providing a pattern input bus as well as a stream input bus, we fulfill the requirements for reconfiguration. To create a system, the units are daisy-chained together into a linear array, shown in Figure 7. In this way the pattern information can pass from unit to unit without system-level control.

The first step in reconfiguration is to set the buffer pointer to the empty position. This is done regardless of the current matching state – as the unit's pattern is being modified, the match information is no longer relevant. The buffer will fill with $k/2$ characters while the pattern and jump table are filled (the buffer is filled exactly as in the initial buffer loading during initial start-up). We have provided a wide data path such that the pattern memory can be filled in $k/2$ cycles, in time for the buffer to be filled (one element per cycle to the size limit of $k/2$ cycles) such that matching can begin.

Because the incoming packet is loaded into the unit's buffer at the same rate as the pattern, we are guaranteed that each pattern will be fully loaded in unit i when the new input fully loads unit i's buffer. Using this strategy, a p element row of k character units can be entirely reconfigured in $pk/2$ cycles given a double-word pattern data path, with no delay to the input stream.

9 Conclusion

This paper has presented a novel systolic-array string matching architecture using a buffered, two-comparator variation on the Knuth-Morris-Pratt algorithm. The architecture competes favorably with the state-of-the-art while providing on-the-fly reconfiguration, better scalability due to the simplicity of the linear architecture, and more efficient hardware utilization. For patterns of size 32 characters it competes with any current published results, even parallel hardwired comparator approaches, without requiring place-and-route between pattern configurations. Our future work includes context-sensitive on-the-fly partial reconfiguration of patterns and on-board π-table generation.

References

[1] 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.

[2] Z. K. Baker and V. K. Prasanna. Automated Incremental Design of Flexible Intrusion Detection Systems on FPGAs. In *The Eighth Annual Workshop on High Performance Embedded Computing (HPEC '04)*, 2004.

[3] Z. K. Baker and V. K. Prasanna. Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of the 14th Annual International Conference on Field-Programmable Logic and Applications (FPL '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 William 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. H. Cho, S. Navab, and W. H. Mangione-Smith. Specialized Hardware for Deep Network Packet Filtering. In *Proceedings the Tenth ACM/SIGDA International Conference on Field-Programmable Logic and Applications (FPL '02)*, 2002.

[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] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, McGraw-Hill, 1990.

[9] 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.

[10] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs. *IEEE Proceedings on Computers and Digital Techniques*, 147(3):181–188, May 2000.

[11] P.W. Dowd and J.T. McHenry. Network Security: It's Time to Take it Seriously. *IEEE Computer Magazine*, 31(9), 1998.

[12] M. Gokhake, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards Gigabit Rate Network Intrusion Detection. In *Proceedings the Eleventh Annual ACM/SIGDA International Conference on Field-Programmable Logic and Applications (FPL '03)*, 2002.

[13] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proceedings of the Tenth Annual Field-Programmable Custom Computing Machines (FCCM '02)*, 2002.

[14] D.E. Knuth, J. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. 6:323–350, 1977.

[15] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. Technical Report 13, Digital Systems Research Center, 1986.

[16] 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.

[17] D. Nguyen, J. Zambreno, and G. Memik. Flow Monitoring in High-Speed Networks with 2D Hash Tables. In *Proceedings of the 14th Annual International Conference on Field-Programmable Logic and Applications (FPL '04)*, 2004.

[18] R. Sidhu, A. Mei, and V. K. Prasanna. String Matching on Multicontext FPGAs using Self-Reconfiguration. In *Proceedings of the Seventh Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '99)*, 1999.

[19] 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.

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

[21] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In *Proceedings the Eleventh Annual ACM/SIGDA International Conference on Field-Programmable Logic and Applications (FPL '03)*, 2003.

[22] 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.

[23] R. Vaidyanathan and J. L. Trahan. *Dynamic Reconfiguration: Architectures and Algorithms.* Kluwer Academic/Plenum Publishers, 2004.

[24] J. Zambreno, D. Nguyen, and A. Choudhary. Exploring Area/Delay Tradeoffs in an AES FPGA Implementation. In *Proceedings of the 14th Annual International Conference on Field-Programmable Logic and Applications (FPL '04)*, 2004.