# Time and Area Efficient Pattern Matching on FPGAs

Zachary K. Baker*
University of Southern California
Los Angeles, CA, USA
zbaker@halcyon.usc.edu

Viktor K. Prasanna
University of Southern California
Los Angeles, CA, USA
prasanna@ganges.usc.edu

## ABSTRACT

Pattern matching for network security and intrusion detection demands exceptionally high performance. Much work has been done in this field, and yet there is still significant room for improvement in efficiency, flexibility, and throughput. We develop a novel linear-array string matching architecture using a buffered, two-comparator variation on the Knuth-Morris-Pratt(KMP) algorithm. For small (16 or fewer characters) patterns, it compares favorably with the state-of-the-art while providing better scalability and reconfiguration, and more efficient hardware utilization.

KMP is a well-known, efficient string matching technique using a single comparator and a precomputed transition table. We add a second comparator and an input buffer, allowing the system to accept at least one character in each cycle and terminate after a number of clock cycles at maximum equal to the length of the input string plus the size of the buffer. The system also provides a clean, modular route to reconfiguring the patterns on-the-fly and scaling the system to support more units, using several rows of linear array elements. In this paper, we prove the bound on the buffer size and running time, and provide performance comparisons against other approaches.

## Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special Purpose Systems

## General Terms

Algorithms, Design, Security

## Keywords

String matching, network intrusion detection, Knuth-Morris-Pratt

*Supported by the United States National Science Foundation under award No. 0311823 and in part by an equipment grant from Intel Corporation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*FPGA'04,* February 22-24, 2004, Monterey, California, USA.
Copyright 2004 ACM 1-58113-829-6/04/0002 ...$5.00.

## 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. These techniques require significant computational resources, and, using highly-parallel adaptive soft processors, provide opportunities for dramatic improvements.

A complete Intrusion Detection Systems(IDS) based on the Snort rules [12] requires a system optimized for hundreds of rules, many of which require string matching against the entire data segment of a packet. We have developed a highly parallel hardware backend technology to dramatically increase the speed of string matching, specifically directed toward intrusion detection and response applications. The high level of performance that we provide is necessary to provide real-time string matching at Internet speeds. Snort, the open-source IDS [12] 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. Using more sophisticated algorithms or higher levels of parallelism, it becomes tenable. Most research in this area has not developed architectures that can handle more than a few hundred rules at reasonable speeds. By minimizing the number of comparators required to match a new input character each cycle, we use less 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, allowing little reduction in performance for any amount of scaling.

Our novel approach to runtime adaptability uses a pipelined, two-comparator, buffered implementation of the Knuth-Morris-Pratt algorithm (KMP)[7] to implement high-performance pattern matching, while yielding a unit design small enough to fit thousands of patterns on a single FPGA. 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 otherwise, but uses configurable memories storing patterns and precompiled jump tables to provide exceptional flexibility.

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. 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 $k/2$ where $k$ is the pattern size. By buffering the input we allow a linear array of pattern matching units to be daisy-chained together. This technique reduces the fanout and the interconnect distance by allowing units to be regularly arranged on an FPGA without long-wire interconnects. 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.

# 2. OUR APPROACH

Our approach to intrusion detection uses a modified version of the KMP algorithm and matching architecture (Figure 1) optimized for running on a linear array. The Knuth-Morris-Pratt algorithm (KMP)[7] is a sophisticated approach to string matching, providing $O(n + k)$ in the worst case, meaning that the a 2kB packet can be searched in only 2048*c comparisons [14]. While there are algorithms that run faster in the average case, there are none that run faster in the worst case. Because the linear array depends on consistent, non-fluctuating consumption of input characters, the KMP algorithm is the ideal solution for our needs.

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 approach, or wide parallel approaches.

There are string matching algorithms that have better average-case running times than KMP, but no single comparator algorithms with better worst-case characteristics. This is vital to our architecture as we will see later, 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 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 $n + k/2$ cycles plus some pipeline latency.

## 2.1 Introduction to KMP

KMP, developed by Knuth, Morris, and Pratt [7] 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 $\pi$-table[1], contains the length of the largest number of characters in the prefix of the pattern $P$ that match the suffix:

$$\pi[q] = \max\{\ j : j \leq q \text{ and } P[1 \ldots j-1] = P[q-j+1 \ldots q-1]\} \quad (1)$$

That is, the $\pi$-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 $\pi$-table, while "abaab" would have useful information because the beginning of the pattern shows up later in the pattern.

The $\pi$-table below is for the worst-case pattern, a Fibonacci string [7]. $\pi[1] = 0$, 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.

| $q$ | 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 |

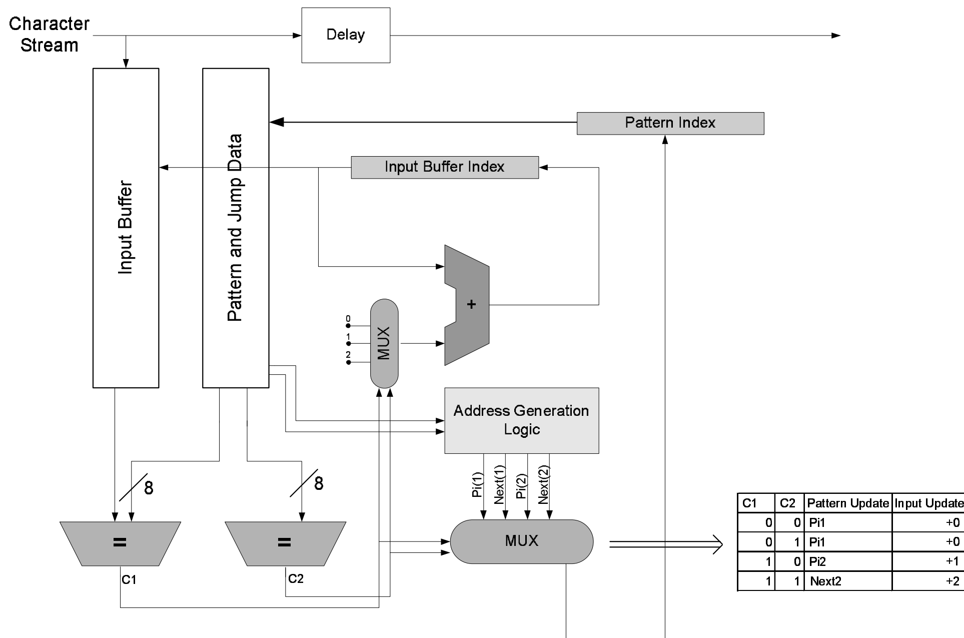
One important note is that when $P(i) = \pi(\pi(i))$, the optimization $\pi(i) <= \pi(\pi(i))$ can be made. This violates Equation 1 but provides much higher performance by removing comparisons that can never be true. This optimization was introduced in the original paper [7] but was omitted in [14].

# 3. RELATED WORK

Snort [12] 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. Some hacker tools even 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 a maintaining a robust IDS.

FPGA solutions attempt to provide a more efficient solution. In our previous work in regular expression matching [10, 11], we presented a method for matching regular expressions using a Non-deterministic Finite Automaton, implemented on a FPGA. Traditional serial machines require $O(2^n)$ memory and $O(1)$ per character. Our approach requires $O(n^2)$ space and $O(1)$ per character. Mapped onto the Virtex XCV100, the approach designed by our group was faster than the *grep* program on an 800 MHz Pentium III for best-case grep performance and orders of magnitude faster than the worst-case grep performance. Using our previous work, [6] implements an FPGA design that deals with two special characteristics of firewall rule sets: the firewall

[1] We use the $\pi$-table as equivalent to the usage of the *next* function in [7]; [14] always compares against $\pi + 1$ whereas the original KMP compares against $\pi$




| 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**

designer has design time knowledge of the rules to implement, and there are large number of rules. Because the rules are known beforehand, the firewalls can be programmed with precompiled rules placed in the rule set according to performance-optimizing heuristics. Their FPGA implementation shows improvements of 600 times the software program GNU *regex*.

In [5], a CAM-powered software/hardware IDS is explored. A Content Addressable Memory (CAM) is used to match against possible attacks contained in a packet. The tool applies the brute force technique using a very powerful, parallel approach. 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 decided 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. CAM hardware, commonly used in fully-associative caches, check a bit vector against all of the data contained within the memory, producing some sort of output if the two pieces of data match. Unfortunately, though, because matching is done in parallel across all rules and across all characters at 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. Because of the hardware complexity and chip limitations, the CAM approach can only provide 32 20-byte matching units on the Virtex XCV1000E, where, given the same external memory hardware, our approach allows several hundred units to run in parallel, with the same worst-case execution time. Also, we do not require that a single packet be matched against all of the rules, allowing packet-specific matching capabilities.

Another hardware approach, in [9], uses more sophisticated algorithmic techniques. In various incarnations, their work has turned into multi-gigabyte sophisticated pattern matching tools with full TCP/IP network support. 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 authors have transferred their designs into a for-sale solution [1] that can operate at OC-48 (2.4 Gbps) rates. In their architecture, a full place-and-route and reconfiguration is currently estimated at 7-8 minutes.

In [4] 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 in a given memory size.

In [3, 13], a hardwired design is developed that provides high area efficiency and high time performance by using replicated hardwired 32-bit comparators in a pipeline structure. The main weaknesses are the $p^2$ increase in hardware for a $p$ increase in throughput and the inflexibility in the hardware. The matching technique proposed is to use four 32-bit 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. The authors use about 100 rules, "the most common attacks," and have implemented only these patterns in the FPGA. Because the comparators are hardwired, they are fast, but are also inflexible and moreover require a full place-and-route for any change in their pattern set. Because [13] provides the highest throughput for the largest number of rules to date,

we focus our comparisons against their results. In a related work [2], we develop some of these ideas using a unary-coded technique with automated optimization design tools.

## 4. ARCHITECTURE

The KMP algorithm generally 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 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 there is no match in the first character, the result of the second comparison is not considered. However, when the first comparison does match, the result of the second comparison is considered. This allows 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 matching prefixes of the pattern are compared against. During this time, however, data continues to enter into the buffer. In Figure 2 we see the input pointer and the incoming buffer 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, until a failing comparison occurs. This causes the input pointer to stall and produces 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, the same speed as the input to the buffer. There is no situation where the read and write buffers can actually wrap around the buffer and collide, falling into either of the gray areas on either side. Buffer overruns or underruns would cause input characters to be lost and is undesirable. We prove that this can never happen in the next section.

After careful timing analysis of the design, it became clear that the large contributions to the period can be split into two 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 we need to update the pointers each cycle based on results from the current cycle.

The solution is a $C$-slowing technique [8], something more akin to a fine grain multi-threaded architecture, where two pattern matching units essentially stay out of phase with

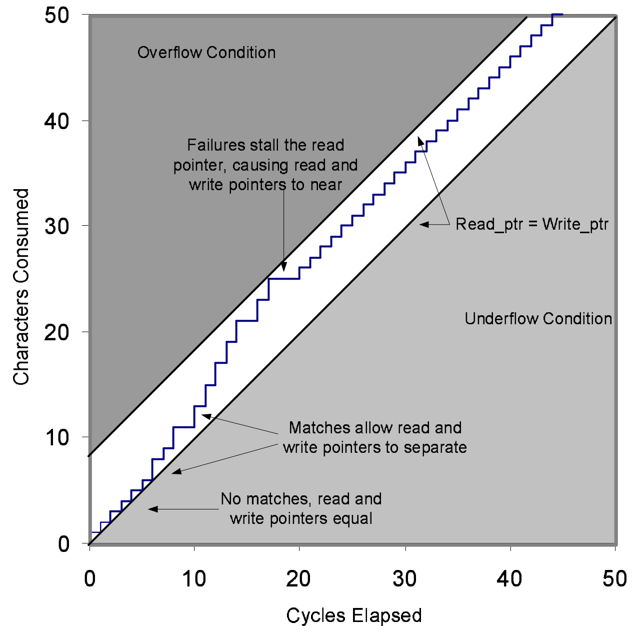



**Figure 2: Separation of read and write pointers in the input buffer**

each other. The two units actually share the same hardware except for the memory areas 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 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 doubles the latency for a single pattern, each unit provides matching for two patterns with only a small increase in area. Thus any clock period performance increase provided by having higher utilization of the circuitry translates directly to increased total performance. By pipelining the units, we increased the place-and-route clock frequency from 221 MHz to 285 MHz.

## 5. OPERATIONAL EXAMPLES

We will explore a sequence of cases, working up to the proof of the general case 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 shows which two characters are being compared in parallel in each cycle.

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 $j$, and causes the same pattern index in the pattern to be compared in the next cycle. The result is that we advance the input index one character in one cycle. This fulfills our requirement for ending up no deeper in the pattern after any sequence of operations.

The second case, shown in Figure 4.b, is a bit more compli-

Figure 3: Pipelined architecture of two-comparator design

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

**Figure 4: Character failures; 'm' is a match, 'f' is a failure, 'd' is a "don't care"**

cated. Here, we compare the first ($P(1)$) and second ($P(2)$) characters of the pattern with the first and second characters of the input, $i(j)$ and $i(j+1)$ in the first cycle. The first character matches ($i(q) = P(1)$), and the second character fails ($i(j+1) \neq P(2)$). 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 and we fail on the third character $P(3)$ as in Figure 4.c. We gain a cycle in the comparison of $P(1)$, $P(2)$, 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) 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. If $\pi[3] = 1$, and $i[\pi[3]] = P(1) \neq i[q+2]$, then we advance to the next input character. Starting at $q = 0$ and advancing to $q = 2$, then failing back to $q^* = 0$ (as required for the important property that any sub-sequence of comparisons also maintains the non-decreasing buffer gap) 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 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 (PME) 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 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 $\pi$-transitions that require comparisons.
*architecture*: the architecture in question is the string matching in Figure 1
*character position*: the position of a characters 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 $\pi$-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 $\pi$-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. □

Because full-cycle traversals are the worst-case situation, if the condition is satisfied for all $q$ when $q^* = 1$, we can be satisfied it 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, $P[1,2,\ldots,n\text{-}1,n,\ldots]$, when $P[q] = \alpha$ for $q = 1 \ldots n$ and $\alpha$ is some pattern character.

**Proof:** The $\pi$-table is defined as the maximum $j$ not greater 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 $\pi$ transitions. Moreover, $n$ must be at least 2. This is the only case where sequences of single-character $\pi$-jumps (more than one single-character jump in a row) may occur. Fortunately, the original KMP algorithm [7] 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 certainly may be different character. This causes an interesting situation, as $\pi[q] = q - 1$ can exist for only the $n$th 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 $\pi$-jump. Due to the original KMP algorithm, only patterns having the identical leading $(n-1)$ characters, followed by a different $n$th character, can cause $\pi[n] = n - 1$, and thus there will exist only one single-character $\pi$-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 in the 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:**
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:

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

**Proof:**
First, we define 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

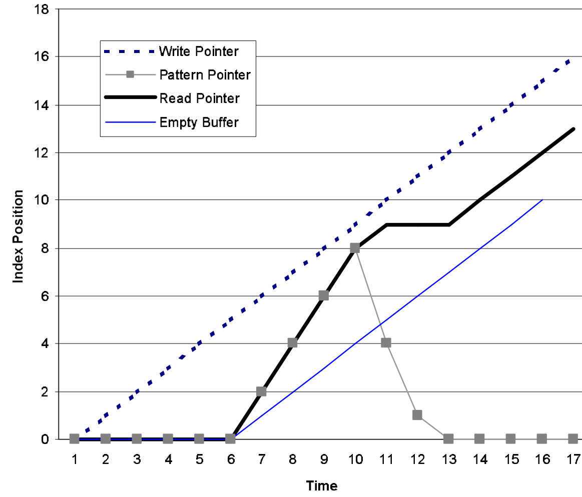



**Figure 5: Various relevant pointers during a simulation of the two-comparator algorithm. The input sequence matches the 16 character pattern for the first 15 characters of the worst-case Fibonacci string pattern. The pointers into the buffer are do not wrap at the buffer boundaries for clarity.**

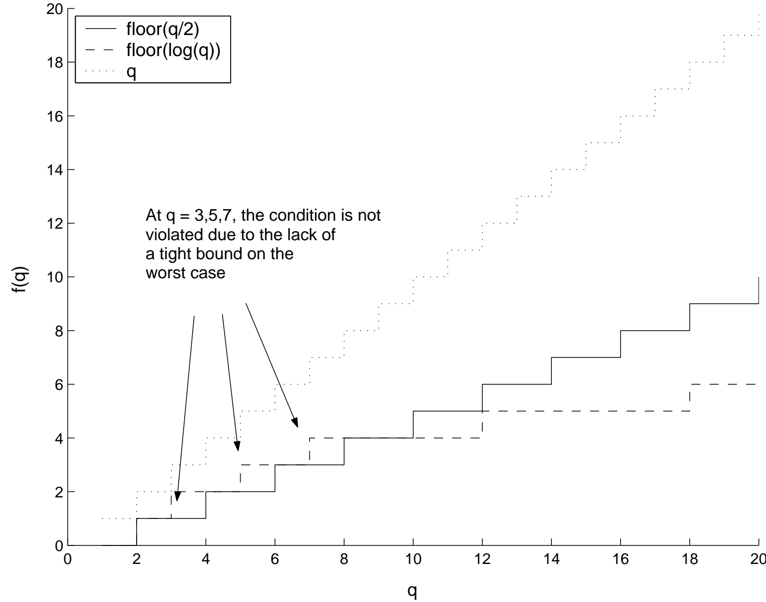



**Figure 6: Using $q^* = 1$, we graph $C_c$, $c_{matching}$, $c_{failing}$ in the formula $C_c \geq c_{matching} + c_{failing}$**

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 [7] proved that the worst case number of $\pi$-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 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 6), because the number of $\pi$-transitions decreases logarithmically with the position in the pattern. Pattern positions less than 8 seem to be much 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 | ~~2~~ 0 | ~~F~~ T |
| 4 | 2 | 2 | T |
| 5 | 3 | ~~3~~ 1 | ~~F~~ T |
| 6 | 3 | 3 | T |
| 7 | 4 | ~~4~~ 3 | ~~F~~ T |
| 8 | 4 | 4 | T |
| 9 | 5 | 4 | T |
| 10 | 5 | 4 | T |

**Figure 7: Correctness table based on worst-case bounds, which are found in Figure 6 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 cycles out and correspondingly adjust the success flag.**

We note that 3, 5, and 7 are not successful (as also indicated in Figure 6), based on the worst-case formulation provided by Knuth, Morris, and Pratt in their original paper [7]. 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:

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $P[i]$ | a | b | a | a | b | a | b | a | a |
| $\pi[i]$ | 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. 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. □

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

The number of clock cycles required to completely match a pattern starting from the first position, $\lceil k/2 \rceil$, is actually greater than the precise buffer size required, $\log_\phi k$. 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 slightly larger buffer allows the architecture to offer on-the-fly reconfiguration.

To conclude this section, we have proved that with two comparators and a $k/2$ buffer a matching unit can accept one character per cycle, while guaranteeing that no character will ever be dropped nor any pattern match overlooked.

# 7. FUNCTIONAL SIMULATION

In this section we present the results of a functional simulation of an 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 5.

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 5, 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.

# 8. 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.

Our design provides reconfiguration that proceeds at the

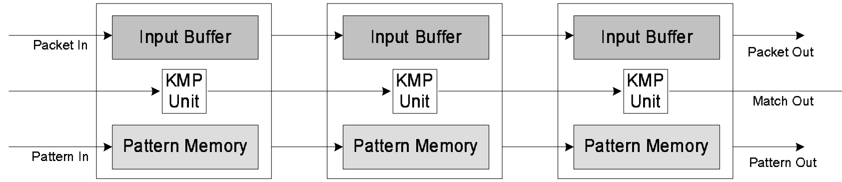



**Figure 8: Linear array of matching units. Each input buffer is of size $k/2$, each pattern memory is of size $k$.**

| Implementation | Total Work Performed | Input Bits | Freq (MHz) | Throughput |
|---|---|---|---|---|
| USC(no pipeline) | $2n$ | 8 | 221 | 8*freq=1.8 Gb/s |
| USC(pipelined) | $2n$ over two patterns | 8 | 285 | 8*2 * freq/2 units = 2.4Gb/s[2] |
| Los Alamos[5] | $128n$ | 32 | 68 | 32*freq=2.2Gb/s |
| Wash U. - DFA[9] | $8n$ | 8 | 80 | 8*freq=0.640Gb/s[3] |
| Wash U. - Bloom[4] | $8n$ | 8 | 100 | 8*freq=0.8Gb/s[3] |
| UCLA [3] | $128n$ | 32 | 90 | 32*freq=2.88Gb/s |
| U/Crete[13] | $128n$ | 32 | 340 | 32*freq=10.8Gb/s |

**Figure 9: Throughput and total work performed, e.g., the total number of comparisons made (considers only a single stream, regardless of how many streams are processed in the original work)**

| | | Unit Size | | Performance | |
|---|---|---|---|---|---|
| Design | Throughput | 16 char | 32 char | 16 char | 32 char |
| USC(no pipeline) | 1.8Gb/s | 92 | 102 | 19.6 | 17.6 |
| USC(pipeline) | 2.4Gb/s/2 | 120/2 | 130/2 | 20.0 | 18.4 |
| Los Alamos[5] | 2.2Gb/s | 243 | 486 | 9.1 | 4.5 |
| Wash U. - DFA[9] | 0.952Gb/s | 260 | 520 | 3.7 | 1.8 |
| Wash U. - Bloom[4] | 0.8Gb/s | 0.76[4] | 0.76 | 1058 | 1058 |
| UCLA [3] | 2.88Gb/s | 160 | 320 | 18.0 | 9.0 |
| U/Crete[13] | 10.8Gb/s | 269 | 538 | 40.1 | 20.1 |

**Figure 10: Pattern size, unit size (in logic cells; one slice is two logic cells), and performance (in Mb/s/logic cell). Throughput is assumed to be constant over variations in pattern size**

same rate as the flow of the input packet into the buffer. One troubling aspect of other designs in the field is the amount of time required to change the patterns in the unit. 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 [12], rules are sorted into categories based on port number and protocol. All current systems, including our design, cannot support all of the possible rules in a single device. By allowing the device to be reconfigured for every new type of packet the system would actually be practical for fully protecting a network.

The architecture of our system is dependent on several small banks of memory that hold the various patterns and jump tables. These buffers give us the leeway necessary to load new patterns. The pipelined system allows us to accept a character into the buffer in every cycle. In order to support on-the-fly reconfiguration we must be able to load the new pattern into the unit memory in $k/2$ cycles, because if the first $k$ incoming characters happens to match the pattern, it will finish in $k/2$ cycles due to the two comparators. However, there is an active comparison cycle only every other system cycle due to the pipelining. Thus we only need $k$ cycles to reconfigure, and that reconfiguration can be done between the end of a packet and the next packet, without delay.

By providing a pattern and jump table input bus for each of the units, we easily fulfill the system requirements for reconfiguration. By daisy-chaining the units together into a linear array, shown in Figure 8, the pattern information can pass from unit to unit without system-level control. After the leftmost unit's pattern memory is full, the unit immediately sends the incoming patterns to the next unit in the chain. Because the incoming packet has to pass though each unit's buffer before it is sent to the next unit we are guaranteed that each pattern will be fully loaded in unit $i$ before the new input fully loads unit $i$'s buffer and matching can start. By switching the currently matching characters to the next element, we can fill the $i+1$ buffer without dropping any characters or matching less than one character per cycle, on average. Using this strategy, a row of the pipelined units can be reconfigured in $pk$ cycles given a 16 bit pattern data path. However, on-the-fly reconfiguration comes at a price; the front-to-back latency for the pipelined system is also $pk$. It is possible to reconfigure the architecture such that the total latency is $p+k$, but in this configuration only a single unit can be reconfigured on-the-fly.

In Figure 9, we compare the total work performed for various implementations, and its relation to the number of input bits and throughput. We find that hardwired, fast 32-

bit implementations can perform remarkably well in terms of throughput. However, they do not offer the reconfigurability 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 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 linear 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 [9], 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, 13], 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. Because of these concerns we define our performance metric as throughput (freq * number of bits per cycle) divided by the size of a 16 or 32 character unit (using the per-character numbers from [13]). This metric rewards systems that have small and highly efficient units, but also those with high operational frequency and parallelism. In Figure 10 we can see that in terms of Megabit/sec/slice, our design compares favorably with designs in the 32 character category.

# 9. CONCLUSION

This paper has presented a novel linear-array string matching architecture using a buffered, two-comparator variation on the Knuth-Morris-Pratt algorithm. For small (16 character) patterns, it 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 $\pi$-table generation.

[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

# 10. REFERENCES


[1] Global Velocity, http://www.globalvelocity.info/, 2003.

[2] Z. K. Baker and V. K. Prasanna. Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs. Submitted to DAC '04, 2004.

[3] Y.H. Cho, S. Navab, and W.H. Mangione-Smith. Specialized Hardware for Deep Network Packet Filtering. In *Proceedings FPL 2002: 12th International Converence on Field-Programmable Logic and Applications*, Sept 2002.

[4] 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 HOTi03*, 2003.

[5] M. Gokhake, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards Gigabit Rate Network Intrusion Detection. In *Proceedings of FPL2002*, 2002.

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

[7] D.E. Knuth, J. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. In *SIAM Journal on Computing*, 1977.

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

[9] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of FCCM 2003*, April 2003.

[10] R. Sidhu, A. Mei, and V. K. Prasanna. String Matching on Multicontext FPGAs using Self-Reconfiguration. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb 1999.

[11] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, April 2001.

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

[13] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In *Proceedings of FPL2003*, 2003.

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