A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs[1]

Zachary K. Baker and Viktor K. Prasanna
University of Southern California, Los Angeles, CA, USA
zbaker@halcyon.usc.edu, prasanna@ganges.usc.edu

Abstract

Intrusion detection for network security is a computation intensive application demanding high system performance. System level design, a relatively unexplored field in this area, allows more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance. By applying optimization strategies to the entire database, we reduce hardware requirements compared to architectures designed with single pattern matchers in mind. We present a methodology for system-wide integration of graph-based partitioning of large intrusion detection pattern databases. Integrating ruleset-based graph creation and min-cut partitioning, our methodology allows efficient multi-byte comparisons and partial matches for high performance FPGA-based network security. Through pre-processing, this methodology yields designs with competitive clock frequencies that are a minimum of 8x more area efficient than any other shift-and-compare architectures, and 2x that of other predecoded architectures.

towards

1 Introduction

Pattern matching for network security and intrusion detection demands exceptionally high performance. Much work has been done in this field [3, 4, 5, 8, 13, 16], and yet efficient, flexible, and powerful systems still have significant room for improvement. 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 flexible fabrics such as FPGA, provide opportunities for dramatic improvements.

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

Snort, the open-source IDS [15], and Hogwash [1] have thousands of content-based rules. A system based on these rulesets requires a hardware design optimized for thousands of rules, many of which require string matching against the entire data segment of a packet.

Previous designs have had excellent single-pattern performance, but when integrated into a system, their area-time performance plunges due to poor resource usage and interconnect and routing complexity. No longer can pattern matching systems be judged on the characteristics of a single matching unit. In a real environment where thousands of rules must coexist on a single FPGA, the pervasive matching unit-level view of Intrusion Detection cannot persist.

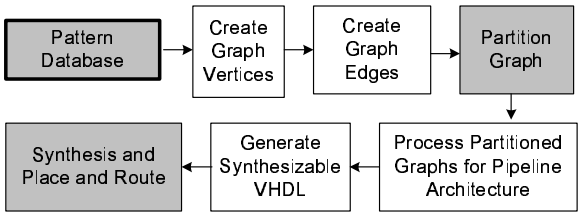

Figure 1. Process flow for synthesis of partitioned system

By carefully and intelligently processing an entire ruleset (Figure 1), we can *partition* a ruleset in order to optimize the area and time requirements of the system. By applying graph theory techniques early in the circuit design problem, we can produce efficient, highly parallel hardware backend technology for string matching, specifically directed toward intrusion detection and response applications. We provide dramatic improvements over the state-of-the-art, with initial results demonstrating that system of several hundred patterns, with a total of over 8200 characters, can operate at over 250 MHz, with a throughput/area performance 8x that of similar "shifting" architectures [16, 4], and 2x that of other predecoded architectures [5].

This paper consists of a brief review of previous work in string matching for network security, followed by an introduction to our approach. Section 4 describes our ideas

regarding the use of simple architectures made efficient through complex preprocessing. Next, we discuss the tool we have developed for automatically applying our optimizations and generating hardware descriptions. In Section 6 we provide the result of our experiments, and then finish with a description of a new strategy we have explored for increasing the throughput and efficiency of the design.

2 Related Work

Snort [15] 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, if necessary, are string-matched 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. If the IDS releases some uninspected packets to prevent buffer overflow, the uninspected packets provide an opportunity for the hacker. Clearly, this is not an effective solution for a maintaining a robust IDS.

SiliconDefense [9] has implemented a software tree-searching strategy that uses elements of the Boyer-Moore and Aho-Corasick algorithms to produce a more efficient search of matching rules in software.

Hardware, particularly FPGA-based approaches, can provide much higher performance through streaming, highly parallel archictures. We will review some of the recent works here.

In [13], a multi-gigabyte pattern matching tool with full TCP/IP network support is described. The system demultiplexes a TCP/IP stream into several substreams and spreads the load over several parallel matching units using Deterministic Finite Automata pattern matchers. In their architecture, a full place-and-route and reconfiguration is currently estimated at 7-8 minutes. Unfortunately, their performance drops dramatically when more than a few rules are integrated into the system, and there seems to be no solution at present to the slowdown problems.

A more powerful work from the same group, in [6], is a novel hashing mechanism utilizing the Bloom filter. Their implementation of a hashing-table lookup using a Bloom filter is an effective method to search thousands of variously-sized patterns for matches in a single pass.

In [4, 16], 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 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. The main weakness are the p^2 increase in hardware for a p increase in throughput.

Finite state machine implementations have drawn research interest as well. We developed the use of nondeterministic finite automata for string matching in [14]. This work was extended for use in network security in [8], but it was found that the approach does not scale well. Device frequency trends downward dramatically as the number of rule increases above a few dozen. In [5] the scaling problem is addressed through a pre-decoding strategy, converting characters to single bit lines in the cycle before they are required in the state machine. This has proved effective in reducing the area of designs and allowing more patterns to be packed in a given FPGA device.

This paper is related to our work in [3], in which we developed 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 in performance efficiency while providing better scalability and reconfiguration, and more efficient hardware utilization. While this paper continues to focus on the optimization of area-time performance, we move forward using a novel preprocessing of rulesets. In the next section, we discuss our approach using preprocessing, and some of the encoding ideas independently developed in [5], and high-throughput, low-area evolutions of the simple, brute-force matching architectures of [4, 16].

3 Our Approach

Most previous research on high-performance string matching has been centered around the performance of single matching units. High performance is achieved for these single units, but system performance is not emphasized, because, for the most part, increasing the size of the rulesets dramatically decrease the performance of the systems. Most designs are dependent on either high on-chip bandwidth allowing data to be shuttled to large matching units, or fairly high percentages of control routing and high logic complexity. Both of these characteristics lead to poorly scaling system designs.

Carefully partitioning rule databases allows a system to more effectively utilize its hardware resources. The key to our performance gains is the idea that patterns sharing

characters do not need to be redundantly compared. Redundancy is an important idea throughout string matching; the Knuth-Morris-Pratt algorithm[3, 12], for instance, uses precomputing redundancy information to prevent needlessly repeating comparisons. We utilize a more dramatic approach; by pushing all character-level comparisons to the beginning of the comparator pipelines, we reduce the single character match operation to the inspection of a single bit.

Previous approaches to string matching, excepting [5], have centered around a byte-level view of characters. High performance designs even increased the base comparison size to 32 bits, providing high throughput by processing four characters per cycle. Increasing the number of bits processed at a single comparator unit increases the fan-in to single gates. Our approach moves in the opposite direction, to single-bit, or unary, comparisons. We decode an incoming character into a "one-hot" bit vector, in which a character maps to single bit. This early decoding is referred to as "shared decoding" in [5].

Unfortunately, without some reduction in the character set, unary representations are almost entirely useless due to the level of inefficiency caused by the huge number of bit lines required for the 256 character ASCII set. However, if the character set can be reduced, the number of bit lines can be dramatically reduced. The most trivial example of reduced sets is DNA matching, where the only characters relevant are A,T,C,G, represented as four one-hot bits. A more interesting example is string matching for network security, where thousands of patterns need to be matched simultaneously at high throughput rates.

Because intrusion detection requires a mix of case sensitive and insensitive alphabetic characters, numbers, punctuation, and hexadecimal-specified bytes, there is an interesting level of complexity. However, each string only contains a few dozen characters, and those characters tend to repeat across strings. Using techniques from graph theory, the patterns are partitioned n-ways such that the number of repeated characters within a partition is maximized, while the number of characters repeated between partitions is minimized, the system can be composed of n pipelines, each with a minimum of bit lines.

The results presented later in this paper use a series of 4 sets of patterns, all subsets of the Nikto ruleset of the Hogwash database [1]. We have arbitrarily selected sets of 204, 361, 602, and 1000 rules to provide some idea as to the scaling behavior of the systems.

4 Simple Architectures, Complex Preprocessing

Our unary design utilizes a simple pipeline architecture for placing the appropriate bit lines in time (Figure 2). Because of the small number of total bit lines required (generally around 30) and extensive pipelined fanout to the individual comparators, adding delay registers adds little area to the system design. The new design takes the the general brute force matching technique used by UCLA [4] and Crete [16] and moves the character decoding to the first stage in the pipeline, and reduces the overall size of the individual comparators by one-eighth. Each pipeline contains only the characters required by the patterns for which the pipeline is responsible. The length of each pipeline is equal to the length of the longest pattern in the pipeline. The maximum latency of the system as a whole is equal to the length of the longest pipeline plus some lead-in and lead-out cycles for collecting results from each pipeline and providing a few register stages to and from the IO pads. The lead-in is 3 cycles, and the lead-out is 8 cycles for systems with less than 500 rules and 4 for larger systems.

The first and obvious problem is the number of characters that might need to be matched for any input pattern. Data is encoded for a reason, after all, and having a bit for every character is not efficient. The routing of a large amount of bit lines is fairly expensive, but because of the pipelining between units, there is little time performance penalty. If the design automation tools can partition a ruleset to 30 bits, or roughly the same amount of data routing as in [16], and then make the actual matching units at least 8x smaller (32 down to 4 bits for [4]), we can expect to see dramatic performance (area-time) increases.

In the whole of the Snort database, there are only about 100 different characters ever matched against. Some of those are case insensitive, or can be made case insensitive without loss of generality, reducing the number of unique characters to roughly 75. Hexadecimal-specified non-character bytes are expanded to equivalent character representations. The next step is to partition the patterns into several groups such that the minimum number of characters have to be piped through the circuit; that is, we give each group of patterns a pipeline, and go through various heuristic tricks to attempt to reduce the pipeline register width.

The graph creation strategy is as follows. We start with a collection of patterns, represented as nodes of a graph. Each pattern is composed of characters. Every node with a given letter is connected by an edge to every other node with that letter. In Figure 3 we illustrate this operation in pseudo code, leading to the graph definition in Figure 4.

This produces a densely connected graph, almost 40,000 edges in a 361 vertex graph. Our objective is to partition the big graph into two or more smaller groups such that the number of edges between nodes within the group is maximized, and the number of edges between nodes in different groups is minimized. Each pipeline supplies data for a single group. By maximizing the edges internal to a group and minimizing edges outside the group which must be dupli-

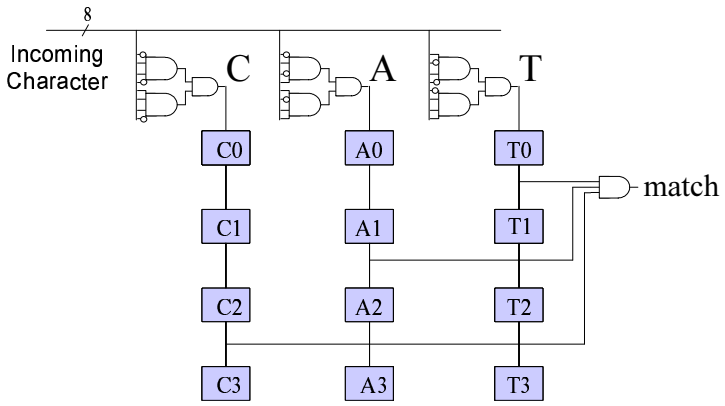

Figure 2. Illustration of unary pre-decoded architecture

```
create graph vertex for each pattern in ruleset;

  for each pattern L in ruleset
      for each pattern K in ruleset not L
          if some character in pattern(L) matches some character in pattern(K)
             add edge between vertex L and vertex K
```

Figure 3. Graph creation before partitioning

$S_k = \{a : a \in C \mid a \text{ appears in } k\}$
$V_R = \{p : p \in T\}$
$E_R = \{(k,l) : k,l \in T,\ k \neq l \text{ and } S_k \cap S_l \neq \emptyset\}$

Figure 4. Graph creation before partitioning; a vertex is added to graph R for each pattern p in the ruleset T and an edge is added between any vertex-patterns that have a common character in the character class C

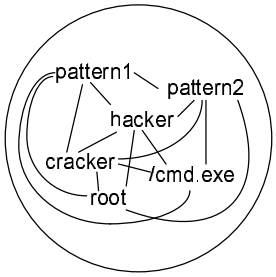

Figure 5. Unpartitioned graph of all patterns; each node is a pattern and each edge is a common character

cated, we reduce the width of the pipeline registers, but improve the usage of any given character within the pipeline.

This problem is a standard graph theory problem in physical design automation known as "mincut" problem. A single circuit needs to be split into two chips, but the two sections are interdependent. The mincut solution minimizes the number of wires connecting the two chips, thus decreasing the number of pins and reducing energy consumption and clock period. The most popular strategy is an iterative improvement heuristic by Kernighan and Lin [11] (1970).

A key contribution of this paper is the dramatically reduced complexity of the mincut operation by partitioning not netlists, but the set of patterns. Because the operation is at a much higher level, at the layer of the patterns and character similarities rather than gates and wires, the system not only requires much less pruning by the synthesis tool, but allows for much easier pre-synthesis performance estimation.

These characteristics allow the designer to more easily create multi-device detection systems. However, it is not necessary to turn to multiple devices, as simply creating independent pipelines on a single chip can reduce clock period by reducing routing delays. Section 6 shows that placing partitioned pipelines on a single device can yield a clock period as much as 20% lower than an unpartitioned system with the same functionality.

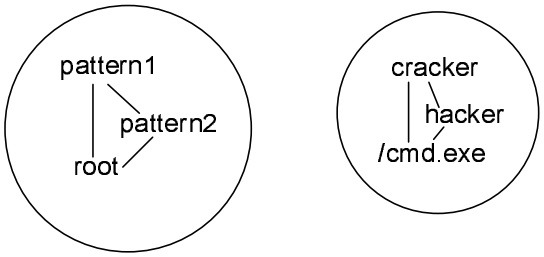

Figure 6. Partitioned graph; by reducing the cut between the partitions we decrease the number of pipeline registers

5 VHDL Synthesis Tool

In order to partition the graphs and generate corresponding hardware descriptions, we utilize a tool we developed, described in [2]. Utilize the METIS graph partitioning library [10], the graph is partitioned into separate sections for separate synthesis. Each pattern within a given partition is written out, and a VHDL file is generated for each partition. If the partitions are to be placed on a single device, a VHDL wrapper is also generated appropriately given the partitioning parameters. This process is very efficient, taking less than 10 seconds on a Pentium III-based machine. The size of the VHDL files for the 361 ruleset total roughly 300 kB, but synthesize to a minimum of 715 slices, or 1430 logic cells. While the automation tools handle the system-level optimizations, the FPGA synthesis tools handle the low-level, logic pruning operations. This allows the VHDL generation after partitioning to proceed unintelligently, creating a full pipeline for the depth of the maximum length pattern for each character utilized in the set of patterns included in the partition. During synthesis, the logic that is not required is pruned – if a character is only utilized in the beginning of a pattern, it will not be carried to the end of the pipeline. If a character is only used by one pattern in the ruleset, and in a sense wastes resources by inclusion in the pipeline, pruning can at least minimize the effect on the rest of the design.

One question that might be raised is in the scaling of the graph creation. To implement the pseudo code in Figure 3, we require an outer and an inner loop to compare all patterns k and l; the outer loop requires n iterations, and the inner loop requires $n-1$ iterations. Comparing the utilized-character sets S_k and S_l requires a number of comparisons equal to the product of the size of the two sets.

The total number of character comparisons during the graph creation operation is

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(|S_i||S_j|) - \sum_{i=1}^{n}|S_i|^2$$

The worst case graph size is $(n-1)(n)/2$ edges for n vertices.

The size of the utilized-character sets are limited in size, generally less than 50 and average between 10 and 20. For our analysis, we can consider them constant, making the time complexity of the sort $O(n^2)$, with a space complexity of $O(n^2)$.

The time complexity of general graph partitioning problem using the Kernighan-Lin algorithm is $O(n^2 \log n)$, with a space complexity equal to the size of the input graph. Thus the time complexity of the complete process is $O(n^2 \log n)$ with a space complexity of $O(n^2)$.

Because of the large number of patterns in current intrusion detection databases [15, 1], creating the pattern-connection graphs and subsequently partitioning the graphs is an expensive operation.However, even with the large memory requirements for representing the graphs, the process flow requires little time. Several hundred rules can be compiled into a graph, the graph can be partitioned, and the partitions can be generated in less than ten seconds. With several thousand rules, compilation times run into the minutes. However, all code except the partitioning tool is written in Perl, a runtime language. Regardless of the implementation, the automatic design tools occupy only a small fraction of the total hardware development time, as the place and route of the design requires orders of magnitude more time.

6 Performance Results

This section will present results based on the graph generated, partitioned pipelines generated automatically by our tool. The results are based on rulesets of 204, 361, 602, and 1000 patterns, subsets of the Nikto ruleset of the Hogwash database [1]. The main trend we see in the results is that the predecoded unary architecture provides dramatic area improvements and good time performance. We then trade a small amount of the area improvement for up to a 20% increase in time performance through the use of partitioning.

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 [4, 16], 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 perfor-

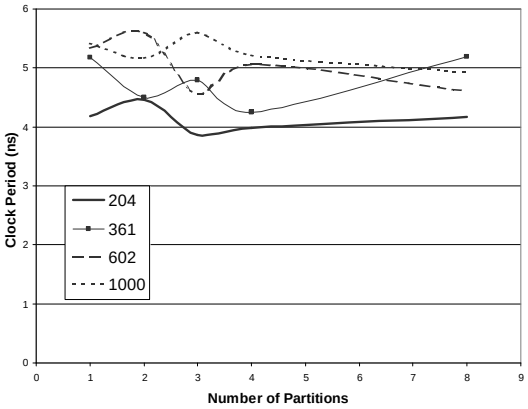

Figure 7. Clock period results for various numbers of partitions for 204, 361, 602, and 1000 patterns

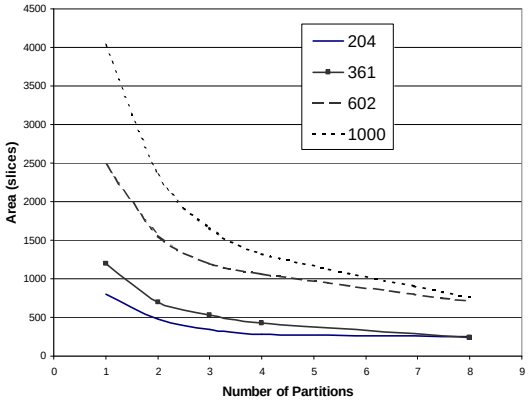

Figure 8. Average partition area (in slices) for various numbers of partitions for 204, 361, 602, and 1000 patterns. Note that the areas approaches lower limits as the number of partition increase, implying the number of partitions should be kept at a minimum

mance metric as throughput (freq * number of bits per cycle) divided by the size of an average pattern (using the per-character numbers from [16]). This metric rewards systems that have small and highly efficient units, but also those with high operational frequency and parallelism.

The results comparison is based on a 16 character pattern. The area of a 16 character pattern is determined from the published information, namely the full system area and the total number of characters matched, and then scaled to a 16 character pattern. While the capabilities of each of the other architecture we compare against vary greatly, as some architectures include support for regular expressions, use of external memory, full TCP stacks, and various FPGA devices, we present the data here so as to give an idea as to where the field stands. The most useful data is the area efficiency, which can be compared fairly between hardware families and generations.

We utilized our tool described in Section 5 to generate system code for various numbers of partitions. Tables 2, 3, 4, and 5 contain the system characteristics for this architectural design, with the system clock and area, the average partition area based on the number of partitions, and the area and clock rate of the first partition (as implemented separately). We provide as a baseline [5] as they also implement a pre-decoded unary architecture. Table 1 contains comparisons of our system versus designs from other researchers. Figure 7 and 8 contain area and speed data for various number of partitions for 204, 361, and 602 patterns.

In Table 2, 3, 4, and 5 we see that the maximum system clock is around 250 MHz for all designs. The system area increases as the number of partitions increases, but the average partition area decreases consistently with increasing partitions. Our clock speed, for an entire system, is in line with the fastest single-comparator designs of other research groups. The startling result is the comparison of per-character area requirements with other designs. The smallest of designs in the published literature is in [8], implementing a Non-deterministic Finite Automata in hardware. That design occupies roughly three logic cells per character. Our design occupies 0.35 logic cells per character (0.17 slices per character), or almost 2.8 characters per logic cell. This is an 8x improvement in efficiency over other shift-and-compare architectures [4, 16] (after allowing for a 4x increase in hardware due to higher throughput), and a 2x improvement over the unary-based NFA design in [5].

Figure 7 shows the clock period of a single-chip system using various numbers of partitions. We see that partitioning can cause the clock period to be reduced by as much as 1 ns., or a full 20% in some cases. It is interesting to note that the effect of the partitioning is unpredictable for fewer than 4 partitions: maxima and minima occur at the same number of partitions for various number of rules. However, while it is useful to break designs into multiple chips, it is clear that large numbers of partitions for a small number of rules provide no area benefit. While time performance is increased, the throughput/area performance metric is compromised by more than 3-4 partitions. Due to the exceptional area efficiency provided by the unary representation, though, we can easily afford to trade area for time performance. While the overhead of replicating comparators for many characters in each partition causes a reduction in efficiency, for larger rulesets (into the thousands of patterns), partitioning into dozens of independent pipelines may prove to be an effective strategy.

The dramatic improvement in area efficiency due to the unary strategy, and the partitioning possible due to the pre-

Design	Throughput	Unit Size	Performance
USC Unary (1 byte)	1.79 Gb/s	5.7	315
USC Unary (4 byte)	6.1 Gb/s	22.3	271
USC Unary (8 byte)	10.3 Gb/s	32.0	322
USC Unary (Prefilter)	6.4 Gb/s	9.4	682
Los Alamos[7]	2.2 Gb/s	243	9.1
Wash U.[2][13]	0.952 Gb/s	260	1.8
UCLA[4]	2.88 Gb/s	160	18.0
U/Crete[16]	10.8 Gb/s	269	40.1
GATech[5]	0.8 Gb/s	12.8	62.1

Table 1. Throughput, unit size per 16 byte pattern (in logic cells; one slice is two logic cells), and performance (in Mb/s/cell). Throughput is assumed to be constant over variations in pattern size

Number of Partitions	[5]	1	2	3	4	8
System Clock (ns)	10	4.179	4.457	3.863	3.986	4.174
System Area (slices)	1807	800	957	1043	1107	2007
Partition Area (avg)	1807	800	478	347	276	250
Single Pipeline area	1807	800	527	306	394	299
Single Pipeline clock	10	4.179	4.326	4.285	4.381	4.209

Table 2. Performance results using various numbers of partitions compared against the shared-decoder architecture in [5]. The total number of characters before partitioning is 4518 over 204 rules

processing of patterns and their unary representations has serious repercussions in the number of patterns that can be matched with a single device, the energy expenditure of the device, and the possibility of placing other complex structure on the device at the same time. One disadvantage of this work, however, is the requirement for place-and-route to make any change in patterns, unlike the designs in [3, 6].

Figure 8 shows the average number of slices per partition for various numbers of partitions over the three sets of patterns. This chart shows that increasing the number of partitions for a multi-device system is ineffective for small pattern sets due to overhead redundancy, but is progressively more effective as the size of the pattern set increases.

For the 361 pattern, 8263 character system, the character graph can be generated, partitioned, and the final synthesizeable, optimized VHDL is created in less than 10 seconds on a Pentium III 800 MHz with 256 MB RAM. The 1000 pattern system requires roughly 30 seconds. The synthesis and place and route is executed on a four processor Pentium Pro time-sharing system with 1GB of RAM, and takes roughly 10 minutes using "highest" place-and-route effort. The synthesis tool is Synplicity Synplify Pro 7.2 and the place and route tool is Xilinx ISE 5.2.03i. The target device is the Virtex II Pro XC2VP100 with -7 speed grade.

[2]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.

7 High-throughput Architecture

The basic architecture described earlier emphasizes both time and area performance, but is centered around an 8-bit input stream. While the frequency performance of the generated architectures is very high, the 8-bit input limits the throughput potential. At 8-bits per cycle, in order to reach a 10 Gbps rate, the device would have to run at 1.25 GHz. Clearly, current FPGA technology cannot support this. The best option, therefore, is to increase the datapath width into the device. The use of k-byte data words complicates the design, however, because now k essentially separate pattern offsets must be detected. That is, we have to guarantee that the beginning of a pattern will start on a word boundary, and thus while we may launch the network stream into the pipeline at k-bytes per cycle, k separate offsets must be detected as well. Thus, the final comparator stage of a 1000 pattern database now presents roughly the routing complexity as a $1000k$ pattern database.

This allows us to easily reach much higher throughput rates while not requiring

We illustrate our 4-way architecture in Figures 9 and 10. It is important to note that while the front and back end comparators are replicated k times, the pipeline itself is shortened by k times, providing some relief from the increase in area. The results of our experiments are shown in Table 6.

Number of Partitions	[5]	1	2	3	4	8
System Clock (ns)	10	4.93	4.497	4.798	4.244	5.193
System Area (slices)	3305	1198	1394	1604	1692	1891
Partition Area (avg)	3305	1198	697	534	399	236
Single Pipeline area	3305	1198	847	737	537	434
Single Pipeline clock	10	4.93	4.95	4.967	4.478	4.46

Table 3. Performance results using various numbers of partitions compared against the shared-decoder architecture in [5]. The total number of characters before partitioning is 8263 over 361 rules

Number of Partitions	[5]	1	2	3	4	8
System Clock (ns)	10	5.333	5.603	4.556	5.063	4.602
System Area (slices)	4930	2466	3117	3607	4264	5673
Partition Area (avg)	4930	2466	1558	1202	1066	709
Single Pipeline area	4930	2466	1657	1350	994	803
Single Pipeline clock	10	5.333	5.228	4.875	4.872	4.781

Table 4. Performance results using various numbers of partitions compared against the shared-decoder architecture in [5]. The total number of characters before partitioning is 12325 over 602 rules

For these experiments, we have utilized the optimal number of partitions from the basic unary architecture. Overall, it is clear that the increase in area is less than k times the 8-bit architecture, and the decline clock frequency is acceptable.

8 Pre-filtering Architecture

In Table 1 we notice the remarkable throughput possible in the shift-and-compare designs of [4] and [16].

The pattern the system is searching for can occur start at any byte position within an input string. That is, there is no guarantee that the start of an offending input will be word-aligned. Thus, replication of matching hardware is required to avoid missing non-32 bit word-aligned strings. Because a single character is expressed as an encoded 8-bit form, accepting 4 characters at a time makes it impossible to accurately match a input string with a single comparator. Instead, four comparators are utilized, each with the pattern offset by a large amount to match all possible starting positions of a character string. This increases the resource requirements for 32-bit architectures dramatically. However, using our architecture, the amount of replication is reduced.

Our 32-bit architecture is a sum-of-products design that allows 4 bytes to be matched per cycle with an increase only in pred-decoding logic, with little increase in routing area or the number of matching comparators. The use of the same datapath is possible by allowing some uncertainty into the design. That is, the character comparators at the start of the pipeline are replicated four times, and the OR of their outputs is fed into the unary character pipeline. This allows up to 4 unary bit lines to be active in the previously one-hot system in any given pipeline stage. In this setup, each pattern can be properly matched at 4 different offsets, necessary to allow for 32 bits to be accepted in each cycle. The pattern comparators in the pipeline operate normally.

However, improper matches (false positives) can occur if characters in the wrong offset happen to be triggered. That is, “cat” and “cct” or “caa” will all trigger a match. Each improper pattern match has a character that is valid in a different alignment. A negative result guarantees that the input stream never matches any of the patterns. A positive result means that the input stream may match the input, and some post processing is necessary.

We implemented the 204 pattern ruleset with the additional front-end comparators. Our four-byte design runs at 200 MHz and occupies only moderately more area, increasing from 957 slices for the original design up to 1270 slices in exchange for four times more throughput. At 32 bits per cycle, a 200 MHz system can match at 6.4 Gb/s.

The main problem with increasing throughput using time-overlapped multiplexing is the potential for false positives. Like [6], there is no possibility of false negatives, but there is an increasing likelihood of false positives as the number of independent streams increases. However, when used as a high-throughput prefilter, it is reasonable to place a second filter behind the prefilter, allowing the secondary filter to perform more complex and exact processing at a much slower rate.

This work can be compared effectively against the Bloom filter in [6] as they both produce results that have

Number of Partitions	[5]	1	2	3	4	8
System Clock (ns)	10	5.41	5.17	5.60	5.22	4.93
System Area (slices)	7833	4028	4693	5001	5285	6123
Partition Area (avg)	7833	4028	1585	1670	1365	771
Single Pipeline area	7833	4028	2928	2474	2307	1778
Single Pipeline clock	10	5.41	4.89	4.68	4.60	4.53

Table 5. Performance results using various numbers of partitions compared against the shared-decoder architecture in [5]. The total number of characters before partitioning is 19584 over 1000 rules

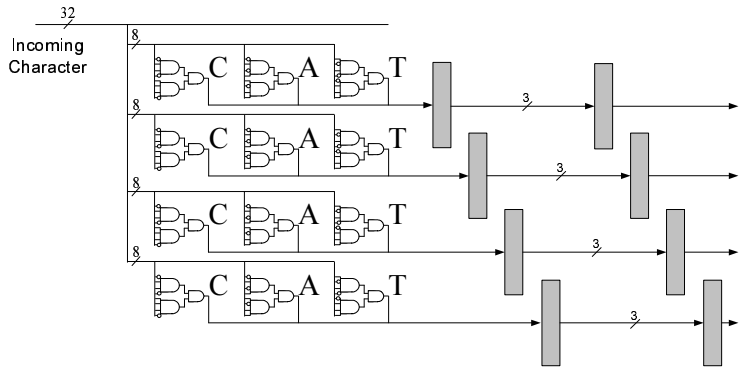

Figure 9. Illustration of 4-way front end. Character decoders are replicated to allow for four different beginning offsets

some possibility of false positives. The Bloom filter provides matching against thousands of patterns at 2.4 Gb/s, at an unit area cost of 1.4 logic cells required for a 16 character pattern (on average). While we can match at 6.4 Gb/s, our false positive rate is much higher.

9 Conclusion

This paper has discussed a methodology for system-wide integration of graph-based partitioning of large intrusion detection pattern databases. By optimizing at a system level, considering an entire set of patterns instead of individual string matching units, our strategy allows more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance.

Through pre-processing, this methodology yields designs with competitive clock frequencies that are a minimum of 8x more area efficient than any other shift-and-compare architectures [16, 4], and 2x that of other predecoded architectures [5].

Our increased throughput design, at the expense of a high false-positive rate, provides an area-time performance approaching than of the Bloom filter [6]. Our architecture and pre-processing allows for area-efficient, high-performance intrusion detection for network security.

References

[1] Hogwash Intrusion Detection System. `http://hogwash.sourceforge.net/`.

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

[3] Z. K. Baker and V. K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of FPGA '04*, 2004.

[4] Y. H. Cho, S. Navab, and W. H. Mangione-Smith. Specialized Hardware for Deep Network Packet Filtering. In *Proceedings FPL '02*, Sept. 2002.

[5] C. R. Clark and D. E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Proceedings of FPL '03*, 2003.

[6] 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 HOTi '03*, 2003.

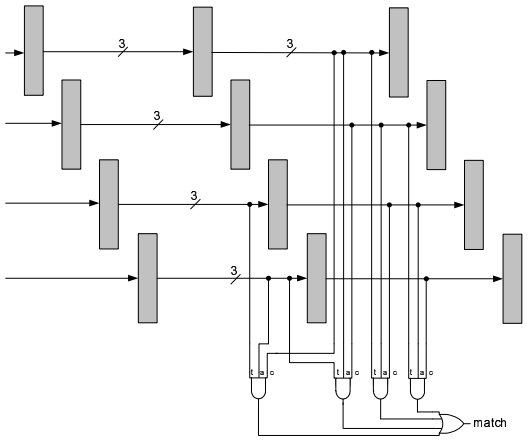

Figure 10. Illustration of 4-way back end matchers. The pipeline moves each block of decoded characters forward by four character positions, and pattern comparators select each decoded character line appropriately

	Number of Rules	Number of Partitions	Area (slices)	Clock Period (ns)
Four Way	200	3	3153	5.27
	400	4	4780	6.64
	600	3	9332	7.95
	1000	8	15010	7.1
Eight Way	200	3	4525	6.2
	400	4	7737	7.24

Table 6. Performance results for 4 and 8-way architectures(32 and 64 bit datapaths, respectively)

[7] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett. Granidt: Towards Gigabit Rate Network Intrusion Detection. In *Proceedings of FPL '02*, 2002.

[8] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proceedings of FCCM '02*, 2002.

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

[10] G. Karypis, R. Aggarwal, K. Schloegel, V. Kumar, and S. Shekhar. METIS Family of Multilevel Partitioning Algorithms. `http://www-users.cs.umn.edu/~karypis/metis/`.

[11] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs, 1970. Bell System Tech.

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

[13] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of FCCM '03*, Apr. 2003.

[14] R. Sidhu, A. Mei, and V. K. Prasanna. String Matching on Multicontext FPGAs using Self-Reconfiguration. In *Proceedings of FPGA '03*, Feb 1999.

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

[16] I. Sourdis and D. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In *Proceedings of FPL '03*, 2003.