Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs

Zachary K. Baker, *Student Member*, *IEEE*, and Viktor K. Prasanna, *Fellow*, *IEEE*

Abstract—This paper presents a methodology and a tool for automatic synthesis of highly efficient intrusion detection systems using a high-level, graph-based partitioning methodology and tree-based lookahead architectures. Intrusion detection for network security is a compute-intensive application demanding high system performance. The tools implement and automate a customizable flow for the creation of efficient Field Programmable Gate Array (FPGA) architectures using system-level optimizations. Our methodology allows for customized performance through more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance.

Index Terms—Intrusion detection, graph algorithms, partitioning, performance, FPGA design.

1 Introduction

Network-connected devices often have vulnerabilities susceptible to exploitation. In order to protect individual systems and the entire network, network operators must ensure that attacks do not traverse their network links. One method for understanding the attacks on a network is an Intrusion Detection System (IDS). Intrusion Detection Systems use sophisticated rules utilizing string matching to detect potential malicious packets. In order to monitor attacks, a network administrator can place an Intrusion Detection System at a network choke-point such as a company's connection to a trunk line (Fig. 1). The IDS differs from a firewall in that it goes beyond the header, actually searching the packet contents for various patterns that imply an attack is taking place or that some disallowed content is being transferred across the network. Current IDS pattern databases reach into the thousands of patterns, providing for a difficult computational task.

Because the IDS must inspect at the line rate of its data connection, IDS pattern matching demands exceptionally high performance. This performance is dependent on the ability to match against a large set of patterns and, thus, the ability to automatically optimize and synthesize large designs is vital to a functional network security solution. Much work has been done in the field of string matching for network security [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. However, the study of the *automatic design* of efficient, flexible, and powerful system architectures is still in its infancy.

Snort, the open-source IDS [1], and Hogwash [2] have thousands of content-based rules. A system based on these rule sets requires a hardware design optimized for thousands of rules, many of which require string matching against the entire data segment of a packet.

These algorithms require significant computational resources. To support heavy network loads, high-performance algorithms are required to prevent the IDS from becoming the network bottleneck. Even with the most sophisticated algorithms, though, sequential microprocessor-based implementations cannot provide the level of service available in a customized hardware device. In [3], a Dual 1 GHz Pentium III system, using 845 patterns, runs at only 50 Mbps. For a small network with limited traffic and a maximum wire speed of 100 Mbps, the software approach might be acceptable. However, for larger networks and higher bandwidth connections, the uniprocessor approach may be forced to skip some packets and potentially let an attack pass undetected. SPANIDS [4] utilizes a cluster of Linux-based PCs to achieve the high bandwidth performance that we achieve through an FPGA. The main disadvantage of this approach is the physical space required for the cluster. We are interested in providing high-bandwidth intrusion detection on a per-port basis, in which each port in a large network switch would have independent IDS capabilities.

In Section 6, we show that a single FPGA device can support multigigabit rates with 2,000 or more patterns. We can achieve this performance using automated design strategies for creating hardware architectures.

Parallel hardware architectures offer large advantages in time performance compared to software designs, due to easily extracted parallelism in the Intrusion Detection string matching problem. An ASIC design would be fast but not suitable due to the dynamic nature of the rule set—as new vulnerabilities and attacks are identified, new rules must be added to the database and the device configuration must be regenerated. However, a Field-Programmable Gate Array (FPGA) allows for exceptional performance due to the parallel hardware nature of execution as well as the ability to customize the device for a particular set of patterns. An FPGA can provide near-ASIC performance and parallelism, along with the ability to modify the hardware to a particular set of patterns.

• *The authors are with the Department of Electrical Engineering—Systems, University of Southern California, EEB-200, 3740 McClintock Ave., Los Angeles, CA 90089-2562.*
E-mail: zbaker@usc.edu, prasanna@ganges.usc.edu.

Manuscript received 13 Aug. 2004; revised 20 July 2005; accepted 28 Mar. 2006; published online 2 Nov. 2006.
For information on obtaining reprints of this article, please send e-mail to tdsc@computer.org, and reference IEEECS Log Number TDSC-0122-0804.

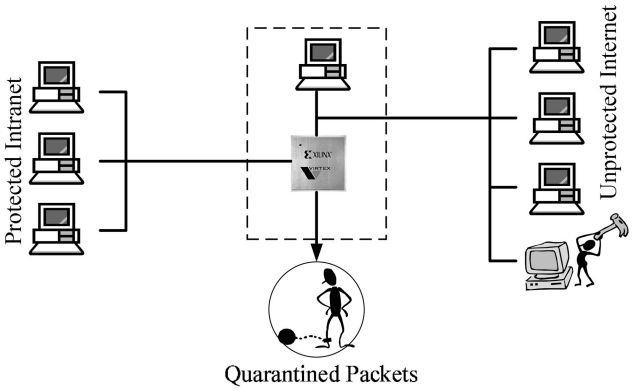


Fig. 1. Intrusion detection systems protect networks from external threats. The use of FPGA allows a system to take advantage of massive parallelism.

Early FPGA designs in the field [5], [6], [7] had excellent performance for a small number of patterns, but when integrated into a system, their area performance decreases due to poor resource usage, and their time performance is impacted by the interconnect and routing complexity.

Our basic architecture is a predecoded multiple-pipeline shift-and-compare matcher. While this approach can be considered "brute force" compared to a state machine approach [8], [5], [9] or a hashing approach [10], the simplicity of the units allows for exceptional area and time performance. The basic architecture, as described in detail below, reduces routing and comparator size by converting incoming characters into many bit lines, each representing the presence of single character. This approach has been explored by several researchers [11], [8], [3].

This basic architecture is extended in various ways. To allow for better area performance, we present a prefix tree architecture that allows for significant reduction in redundant comparisons by independently matching prefixes that are shared across several patterns. To provide increased throughput performance, we provide a design that replicates a fraction of the hardware to allow for exact matching for k bytes per cycle, where k is generally not greater than 8.

The architectures we have developed are only part of the contributions of this paper. To achieve better utilization of these architectures, system-level preprocessing steps are required, serving various functions, including partitioning, grouping, and code generation. These steps, by considering the entire set of patterns in lieu of naive hardware generation, produce higher efficiency in terms of patterns matched per unit area and unit time.

By intelligently processing an entire rule set (Fig. 2), our tool partitions a rule set into multiple pipelines in order to optimize the area and time characteristics of the system. The rule database is first converted into a graph representing the similarity of the rule set. Depending on the tool flow desired, the graph edges are weighted to provide higher connectedness between rules with similar characters; this allows for increased grouping of prefixes and/or general shared-character grouping, as required. The graph is partitioned based on the weighted graph and then prefixes are grouped for the tree architecture, if required. Based on the results of this preprocessing, the system is generated

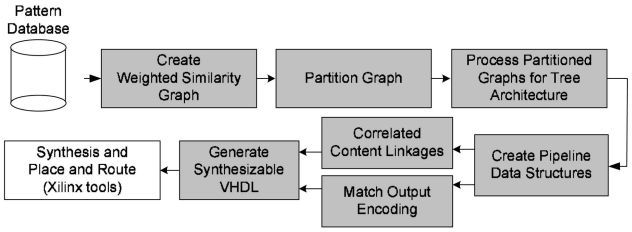


Fig. 2. Automated optimization and synthesis of a partitioned system.

from templates. By applying various graph partitioning operations and trie techniques to the problem, the tool more effectively optimizes large rule set as compared to naive approaches.

This paper describes a methodology for creating Intrusion Detection Systems with *customized performance*, allowing a designer to mix and match from a collection of process steps and a family of architectures we have developed. We begin with an overview of related work in the field (Section 2), and then introduce the reader to our approach (Section 3). We will discuss our basic architecture and then move into the various methodology options that allow for customized performance. We give results for the basic architecture and its variations as compared to other work in the field (Section 6), and review the tools (Section 5) we have developed and some optimizations we have made to decrease the total tool flow latency.

2 Related Work in Automated IDS Generation

Snort [1] and Hogwash [2] are current popular options for implementing intrusion detection in software. They are open source, free tools that promiscuously tap the network and observe all packets. After TCP stream reassembly, the packets are sorted according to various characteristics and, if necessary, are string-matched against rule patterns. However, the rules are searched in software on a general-purpose microprocessor. This means that the IDS is easily overwhelmed by periods of high packet rates. The only option 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 allows packets to pass uninspected during overflow, an opportunity for the hacker is created. Clearly, this is not an effective solution for maintaining a robust IDS.

Automated generation of optimized generic architectures has been explored [12], [13], but domain-specific tools have a distinct performance advantage in network security. Automated IDS designs have been explored in [5], using automated generation of Nondeterministic Finite Automata. The tool accepts rule strings and then creates pipelined distribution networks to individual state machines by converting template-generated Java to netlists using Java-based Hardware Description Language (JHDL) [14]. This approach is powerful, but performance is

reduced by the amount of routing required and the logic complexity required to implement finite automata state machines. The generator can attempt to reduce the logic burden by combining common prefixes to form matching trees. This is part of the preprocessing approach we take in this paper.

Another automated hardware approach, in [15], uses more sophisticated algorithmic techniques to develop multigigabyte 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. In their architecture, a Web interface allows new patterns to be added, and then the new design is generated and a full place-and-route and reconfiguration is executed, requiring seven to eight minutes. As their tools have been commercialized in [16], they are not freely available to the community. However, their development work includes network reconfiguration [17], also explored in [18]. Network reconfiguration is important as it allows for effective deployment of security solutions to large numbers of customers without replicated expensive place-and-route hardware.

System-level optimization has been attempted in software by SiliconDefense [19]. They have implemented a software tree-searching strategy that uses elements of the Boyer-Moore [20] and Aho-Corasick [21] algorithms to produce a more efficient search of matching rules in software, allowing more effective usage of resources by preventing redundant comparisons.

Using some ideas from [22], [5] implements an FPGA design that deals with two special characteristics of firewall rule sets: The firewall designer has design time knowledge of the rules to implement and there are a 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.

The NFA concept is updated with predecoded inputs in [11] and [8]. These papers address the problem of poor frequency performance as the number of patterns increases, a weakness of earlier work. This paper solves most of these problems by adding predecoded wide parallel inputs to a standard NFA implementations. The result is excellent area and throughput performance (see Section 6).

In [6], 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. Unfortunately, though, because matching is done in parallel across all rules and across all characters in one cycle, this sort of implementation requires a great deal of logic. While this does provide $O(n+m)$ worst-case rule matching time, it does so at the cost of a large amount of hardware. Because of the hardware complexity and chip limitations, the CAM approach supports a limited number of units. However, the ability to configure the pattern memories on the fly is an advantage over hardwired approaches.

In [23], [11], [8], [24], [3], [25], hardwired designs are developed that provide high area efficiency and high time performance by using replicated hardwired comparators in a pipeline structure. The hardwiring provides high area efficiency, but are difficult to reconfigure. Hardwiring also allows a unit to accept more than one byte per cycle, through replication. A bandwidth of 32 bits per cycle can be achieved with four hardwired comparators, each with the same pattern offset successively by 8 bits, allowing the running time to be reduced by a factor of 4 for an equivalent increase in hardware. These designs have adopted some strategies for reducing redundancy through predesign optimization. The work in [24] was expanded in [3] to reduce the area by finding identical alignments between otherwise unattached patterns. Their preprocessing takes advantage of the shared alignments created when pattern instances are shifted by 1, 2, and 3 bytes to allow for the 32-bit per cycle architecture.

The notion of predecoding has been explored in [11] and [8] in the context of finite automata. The use of large, pipelined brute-force comparators for high speed was initiated in [24] and continued in [26], [25]. Predecoding in the context of brute-force comparators was developed simultaneously in [25], [3], and our work [23].

The work in [25] utilizes a less elaborate predesign methodology that is based on incrementally adding elements to partitions to minimize the addition of new characters to a given partition. The use of trees for building efficient regular expression state machines was initially developed in [5]. We explored the partitioning of patterns in the predecoded domain in [23]. We utilize these foundational works and build automatic optimization tools on top.

3 Our Approach

This research focuses on automatic optimization and generation of high-performance string-matching of high volumes of data against large pattern databases. The tool generates two basic architectures, a predecoded shift-and-compare architecture and a variation using a tree-based area optimization. In this architecture, a character enters the system and is "predecoded" into its character equivalent. This simply means that the incoming character is presented to a large array of AND gates with appropriately inverted inputs such that the gate output asserts for a particular character. The outputs of the AND gates are routed through a shift-register structure to provide time delays. The pattern matchers are implemented as another array of AND gates and the appropriate decoded character is selected from each time-delayed shift-register stage. The tree variation is implemented as a group of inputs that are prematched in a "prefix lookahead" structure and then fed to the final matcher stage. The main challenge in the tree structure is creating the trees; this is discussed in Section 4.1.

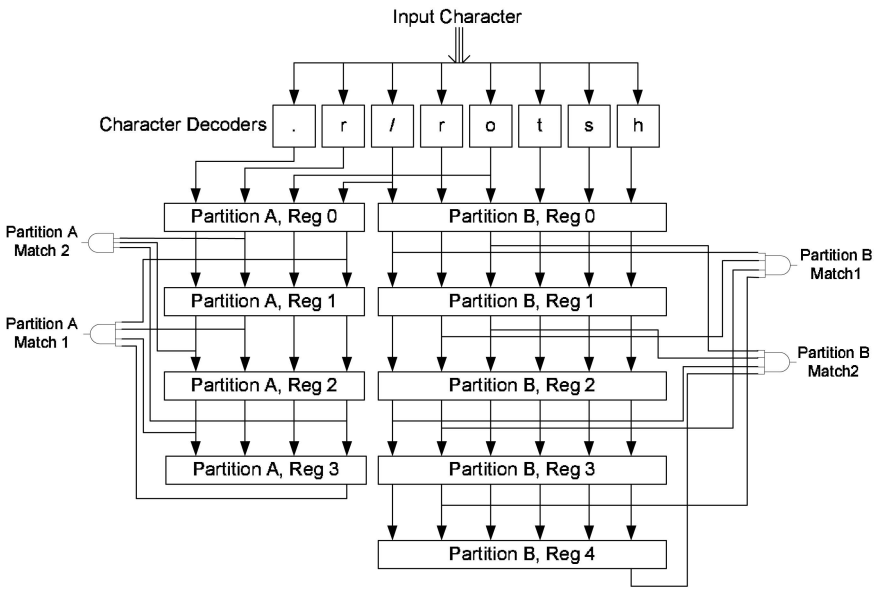


Fig. 3. The general architecture of our pipelined comparators design. Characters are converted to single bits in the first stage and then fed into the pipeline, where they become operands for the pattern comparators.

With our domain-specific analysis and generation tool, extensive automation partitions a rule database into multiple independent pipelines. This allows a system to more effectively utilize its hardware resources. The key to our performance gains is the idea that characters shared across patterns do not need to be redundantly compared. Redundancy is an important idea throughout string matching; the Knuth-Morris-Pratt algorithm [9], [27], for instance, uses precomputed redundancy information to prevent needlessly repeating comparisons. We utilize a more foundational approach; by pushing all character-level comparisons to the beginning of the comparator pipelines (Fig. 3), we reduce the character match operation to the inspection of a single bit.

Previous approaches to string matching have all been centered around a byte-level view of characters. Recent work by our group and others [23], [11], [8], [3], [25] has utilized predecoded, single-bit character reencodings in lieu of delivering 8-bit wide data paths to every pattern-matching unit. High-performance designs have increased the base comparison size to 32 bits, providing high throughput by processing four characters per cycle. However, increasing the number of bits processed at a single comparator unit increases the delay of those gates. The predecoding 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 a single bit. As mentioned, other groups have explored the use of predecoded characters. The architecture in [25] utilizes SRL16 shift registers, where we utilize single-cycle delay flip-flops. This reduces our reliance on Xilinx-style hardware elements and may reduce interconnect costs allowing hardware to spread out more on the device.

This allows efficient multibyte comparisons, regular expressions, prefix trees, and even partial matches using simple sum-of-products expressions.

Unfortunately, without some reduction in the character set, unary representations suffer from the inefficiency caused by the huge number of bit lines required for the 256 character ASCII set. In a set of long patterns utilizing every character in the character space with low repetition, a binary encoding such as the ASCII encoding would likely be the most efficient strategy.

However, if the character set can be reduced, the number of bit lines can be similarly 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. String matching for network security is a more interesting application as thousands of real-world 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. In the entire Hogwash database, there are only about 100 different characters ever used. Some of those are case insensitive or can be made case insensitive without loss of generality and we can convert hexadecimal-specified bytes into their escaped character equivalents (0-9, A-F). This reduces the number to roughly 75 characters. The pattern sets are then broken into smaller, independent pieces. Generally, the optimal number of partitions n is between two and eight. Other researchers [3], [25] have explored other partitioning methods. The approach in [3] breaks the rule set into a group of smaller modules to allow for faster compilation. The approach in [25] is similar to ours in that its purpose is to cluster the most similar patterns together. Their strategy adds patterns to a set of groups based on the minimum set difference between the existing group and the pattern. The min-cut strategy we utilize may allow for more flexibility should the

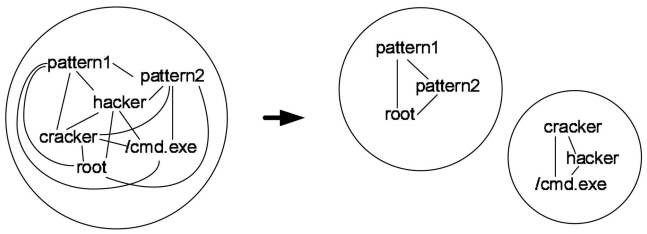


Fig. 4. A partitioned graph. By reducing the cut between the partitions, we decrease the number of pipeline registers.

initial assignment groups prove nonoptimal. Because the heuristics used for the min-cut problem allow all elements to move between groups as the "optimal" partitioning is reached, our approach may be less susceptible to getting stuck in a local minimum due to poor initial random seeds. However, it is difficult to compare the effectiveness of the approaches.

We utilize the Metis partitioning library to partition the patterns after they have been converted into a min-cut problem [28]. 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 is then generated, composed of n pipelines, each with a minimum number of bit lines. The value of n is determined from empirical testing; we have found $n = 2{-}4$ most effective for rule sets of less than 400 patterns. Conversely, for the 603 and 1,000 pattern rule sets, the maximum time performance is achieved with eight partitions. However, as the area increases moderately as the number of partitions increases, the area-time trade-off must be considered.

Our partitioning strategy can partition a rule set to roughly 30 bits, or about the same amount of data routing as one of the 4-byte replicated architectures ([24], [26]). However, the matching units are least 8 times smaller (32 down to 4 bits in an encoded design such as in [24]), and we have removed the control logic of a KMP-style design such as in [9].

Our unary design utilizes a simple pipeline architecture for placing the appropriate bit lines in time. Because of the small number of total bit lines required (generally around 30) adding delay registers adds little area to the system design. Our new design takes the general straightforward matching technique used in [24], [26], but moves the character decoding to the first stage in the pipeline (as in [8]) and reduces the overall size of the individual comparators by one-eighth, as illustrated in Fig. 3.

First, the patterns are partitioned into several groups (Fig. 4) such that the minimum number of letters have to be piped through the circuit; that is, we give each group of patterns a pipeline and go through various heuristic methods to attempt to reduce the pipeline register width. The effect of minimizing the number of characters is to reduce the interconnect burden in each partition pipeline, allowing for better time performance. In the results section, we show that this approach is effective.

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 letters. Every node with a given letter is connected by an edge to every other node with that letter. We formalize this operation as follows:

$$S_k = \{a : a \in C \mid a \text{ appears in } k\}, \quad (1)$$

$$V_R = \{p : p \in T\}, \quad (2)$$

$$E_R = \{(k,l) : k,l \in T,\ k \neq l \text{ and } S_k \cap S_l \neq \emptyset\}. \quad (3)$$

A vertex V is added to graph R for each pattern p in the rule set T and an edge E is added between any vertex-patterns that have a common character in the character class C.

This produces a densely connected graph, with almost 40,000 edges in a graph containing 361 vertices. Each pipeline supplies data for a single group, as illustrated in the system-level schematic in Fig. 3. By maximizing the edges internal to a group and minimizing edges outside the group which must be duplicated, we reduce the width of the pipeline registers and improve the usage of any given character within the pipeline. We utilize the METIS graph partitioning library [28].

One clear problem is that of large rule sets (> 500 patterns). In these situations, it is essentially impossible for a small number of partitions not to require the entire alphabet and common punctuation set. This reduces the effectiveness of the partitioning step. However, if we add a weighting function, the use of partitioning is advantageous as the database scales toward much larger rule sets. The weighting functions is as follows:

$$W_E = \sum_{i=1}^{min(|k|,|l|)} [(min(|k|,|l|) - i) \text{ if } (k(i) == l(i)) \text{ else } 0]. \quad (4)$$

The weight W_E of the edge between k and l is equal to the number of characters $k(i)$ and $l(i)$ in the pattern, with a first character equivalence weighted as the length of the shorter pattern. The added weight function causes patterns sharing character locality to be more likely to be grouped together.

The addition of the weighting function in (4) allows the partitioning algorithms to more strongly group patterns with similar initial patterns of characters. The weighting function is weak enough not to force highly incompatible patterns together but is strong enough to keep similar prefixes together. This becomes important in the tree-based prefix sharing approach, described in Section 4.1.

4 Customized Performance

Given the basic unary architecture, we can now diverge from the basic partitioned flow and create a series of architectures providing *customized* performance. While the basic unary architecture has far higher area efficiency than any other architecture (see Section 6), there are still performance characteristics that can be further optimized. We explore several variations:

- Tree-Based Prefix Sharing.
- High-Throughput Architecture.
- Correlated Content Layer.

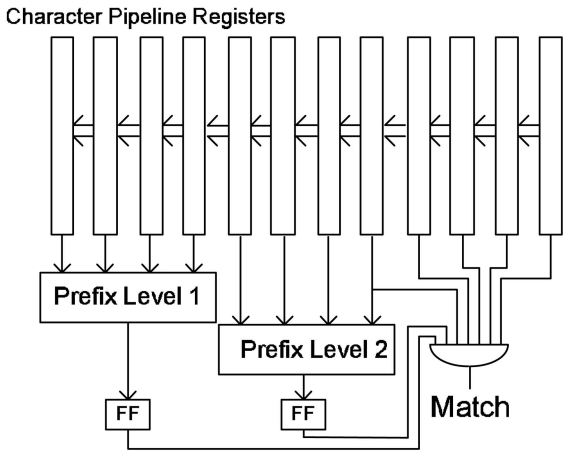


Fig. 5. An illustration of the tree-based hardware reuse strategy. Results from two appropriately delayed prefix blocks are delayed through registers and then combined with remaining suffixes. The key to the efficiency of this architecture is that the prefix matches are reused, as well as the character pipeline stages.

4.1 Tree-Based Prefix Sharing

The first optimization we make is a further area-optimization that allows for sharing of often compared groups of characters, first described in [5] and continued by [11] and others but optimized here for use with a 4-bit FPGA lookup table. We are only interested in finding the pattern prefixes that are shared among matching units in a partition. By sharing the matching information across the patterns, the system can reduce redundant comparisons. This strategy allows for increased area efficiency, as hardware reuse is high. However, due to the increased routing complexity and high fanout of the blocks, it can increase the clock period. This approach is similar to the *trie* strategy utilized in [5], in which a collection of patterns is composed into a single regular expression. Their NFA implementation could not achieve high frequencies, though, limiting its usefulness. Our approach, utilizing a unary-encoded shift-and-compare architecture and allowing only prefix sharing and limited fanout, provides higher performance.

Fig. 5 illustrates the tree-based architecture. Each pattern (of length greater than eight characters as, otherwise, the whole pattern would fit in the two prefixes) is composed of a first-level prefix and a second-level prefix. Each prefix is matched independently of the remainder of the pattern. After an appropriate pipeline delay, the two prefixes and the remainder of the pattern are combined to produce the final matching information for the pattern. This is effective in reducing the area of the design because large sections of the rule sets share prefixes. The most common prefix is /scripts, where the first and second-level prefixes are used together. The 4-character prefix was determined to fit easily into the common FPGA 4-bit lookup table, but it turns out that four-character groups are highly relevant to intrusion detection as well. Patterns with directory names such as /cgi-bin and /cgi-win can share the same first-level prefix, and then each have a few dozen patterns that share the -bin or -win second-level prefix.

In Table 1, we show the various numbers of first and second-level prefixes for the various rule sets we utilized in our tests. Second-level prefixes are only counted as different within the same first-level prefix. For this table, we created our rule sets using the first n rules in the Nikto rule set [2]. There is no intentional preprocessing of the rule sets before the tool flow. The table shows that, on the average, redundant prefix comparisons can be reduced two to three times through the use of the tree architecture. However, some of this efficiency is reduced due to the routing and higher fanout required because of the shared prefix-matching units.

TABLE 1
An Illustration of the Effectiveness of the Tree Strategy for Reducing Redundant Comparisons

	Number of Prefixes	
No. of Patterns	First Level	Second Level
204	83	126
361	204	297
602	270	421
1000	285	528
2000	285	743

On the average, the tree architecture is smaller and faster than the partitioning-only architecture. In all cases, the partitioned architectures (both tree and no-tree) are faster than the nonpartitioned systems.

4.2 High-Throughput Architecture

The basic architecture described earlier emphasizes both time and area performance but is centered around an 8-bit input stream. This architectural variation provides significantly increased throughput by replicating hardware. The effect of this approach is to trade some of the area efficiency of the basic architecture (and the prefix-tree variation) for throughput. This is an effective approach and still yields architectures with better area performance than other designs.

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 on a single stream, 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 data path width into the device. This is a natural extension to the basic architecture and has been explored by other researchers. The first use of multibyte per cycle architectures was in [24], where 32 bits were routed to individual comparator units. The authors updated their multiple byte approach with predecoding in [3], allowing significantly higher area performance. An NFA architecture supporting up to 16 bytes per cycle was introduced in [8]. In this architecture, up to 100 Gb/s rates are possible, but at high area costs.

The use of k-byte data words complicates the design, as k essentially independent pattern offsets must be detected. 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 $1{,}000$ pattern

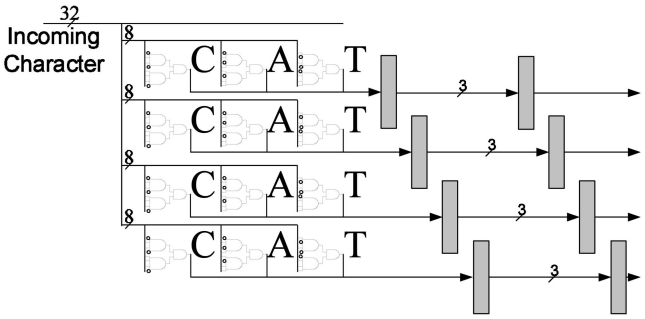


Fig. 6. An Illustration of a four-way front end. Character decoders are replicated to allow for four different beginning offsets.

TABLE 2
Performance Results for Four and Eight-Way Architectures (32 and 64-Bit Data Paths, Respectively)

	Number of Rules	Number of Partitions	Area (slices)	Clock Period (ns)
Four Way	204	3	3153	5.27
	361	4	4780	6.64
	602	3	9332	7.95
	1000	8	15010	7.1
Eight Way	204	3	4525	6.2
	361	4	7737	7.24

database now presents roughly the routing complexity of a $1{,}000k$ pattern database.

We illustrate our four-way architecture in Figs. 6 and 7. 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 2. 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 in clock frequency is acceptable.

4.3 Correlated Content Layer

Current IDS pattern databases reach into the thousands of patterns, providing for a difficult computational task. Further complicating the matter are rules that have multiple patterns. In these rules, the patterns can be linked together (correlated) by a distance constraint. Correlation can reduce false positives by increasing the number of unique elements that form the attack. Correlation also forms the backbone of many attacks. Overflow attacks, for instance, are often defined by a large amount of data before a null character. By distance linking some invariant pattern such as "user" with the null character, the attack can be detected by noting that the null character comes too many bytes after the invariant.

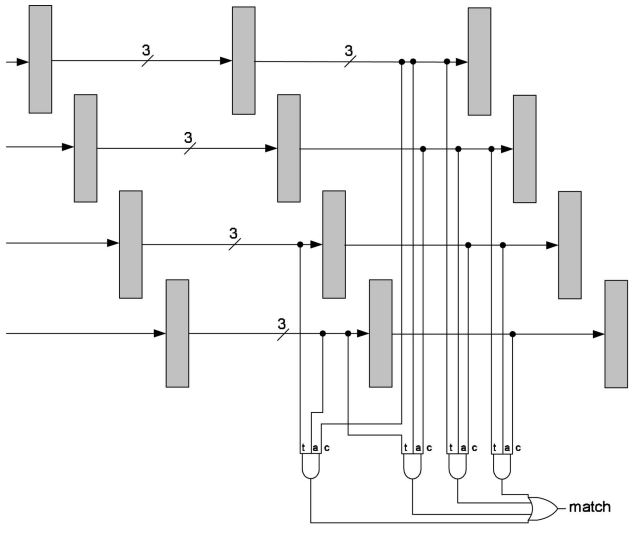


Fig. 7. An illustration of four-way back-end matchers. The pipeline moves each block of decoded characters forward by four character positions and pattern comparators select each decoded character line appropriately.

Most IDS implementations at this point only consider the basic string-matching problem. However, the correlated content problem is becoming more popular. Through the use of state machines, the architecture is sufficiently generic that it can capture the behavior of any regular expression. However, by forcing the hardware to generalize to an unneeded degree, opportunities for extracting performance are lost. It is generally unnecessary to provide this level of flexibility for intrusion detection applications, especially given the cost of highly complex hardwired state machines. Another approach is to utilize a back end that connects the matching results together using small embedded memories [29]. Our approach is to integrate the correlation tightly with the matching itself, before the match results are encoded.

We consider two commonly used extensions, "*distance:*," which requires a minimum number of bytes between two strings, and "*within:*," which sets a maximum number of bytes between two strings.

As discussed earlier, simplicity is the key to performance on FPGAs. We apply this design paradigm to the correlated content layer architecture. The utilization of structures that the FPGA is built to implement can significantly improve time and area results. Toward these goals, we utilize the inherent strength of counter implementation of the Virtex-style FPGAs. The fast carry logic allows a designer to create counters with better area-time performance than a pipelined state machine.

By default, the result of each pattern comparison is sent into a large priority encoder that provides the external host with the appropriate pattern identifier. However, correlated content rules are not based on a single rule, but the sequential detection of two or more rules. Thus, the tool simply disconnects the output of the first comparator from the normal result encoder and reroutes the match information. Depending on the type of constraint required, the system sets a counter and a *valid* bit. The counter is linked to comparator logic to determine the satisfaction of the rule constraints. The counter-linked comparators progressively activate (in the case of *distance* constraint) or deactivates (in the case of *within* constraint) the content matchers deeper in the rule. This is illustrated in Fig. 8.

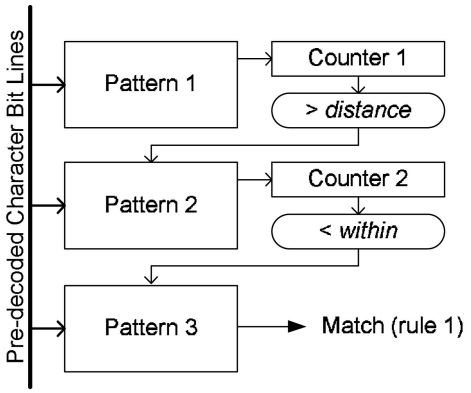


Fig. 8. An architectural meta77layer to provide multicontent rule support.

Table 3 details an interesting comparison in resource utilization and performance in the correlation layer. The experiments are based on the Snort Web-CGI database, which has 347 rules requiring content matches and 420 total strings to consider.

The first row of Table 3 gives results for the full architecture, including linked strings and distance constraints. The second row removes the counter, but keeps the rules linked. The clock rate increases by 0.1 ns, which is insignificant, but expected, as the counter comparison does add another bit to the comparator (the comparator operators are implemented such that the result is available at the clock edge of the next cycle). The most interesting point is that the counters add only a few additional slices to the overall area consumption. This is partly an artifact of the FPGA fabric. The string-matching architecture is composed mostly of Lookup Tables (LUTs) and flip-flops that create the pipeline structure. In the sample rule set, there are 73 strings that are linked to a previous match. Because of this linking, there are fewer entries in the result tree. Thus, while adding the counters requires some flip-flops, the overall design is balanced because of the reduced number of memory elements required for the result tree.

TABLE 3
The Snort Web-CGI Database:
347 Rules Requiring Content Searches Composed of
a Total of 420 Strings and 4,921 Characters

Database Type	Clock Rate	Area
Web-cgi with correlated content	4.4 ns	4590
Web-cgi with no counters	4.3 ns	4372

The Web-CGI database was chosen as it has a significant number of the "within" and "distance" extensions implemented. The additional layer of correlation state machines insignificantly reduces time performance. Area is in logic cells.

5 Tool Performance

After partitioning, each pattern within a given partition is written out and a VHDL file is generated for each partition. A VHDL wrapper with Digital Clock Managers for supported Xilinx chips is also generated, given the partitioning parameters. The size of the VHDL files for the 361 rule set total roughly 300 KB in 9,000 lines. While the automation tools handle the system-level optimizations, the FPGA synthesis tools handle the low-level optimizations. During synthesis, the logic that is not required is pruned—if a character is only utilized in the shallow end of a pattern, it will not be carried to the deep end of the pipeline. If a character is only used by one pattern in the rule set and in a sense wastes resources by inclusion in the pipeline, pruning can at least minimize the effect on the rest of the design.

The worst-case graph size is $(n-1)(n)/2$ edges for n vertices. The utilized-character sets are limited in size, generally less than 50 and averaging 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 the 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. Through the use of the four-character block, we implement the tree structure in $O(n)$ operations. 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 [1], [2], creating the pattern-connection graphs and subsequently partitioning the graphs is an expensive operation. The Hogwash rule set of roughly 7,000 strings creates a graph occupying over 215 MB. However, even with these large memory requirements, the process flow requires little time. For our tests, we use the Nikto rule set of the Hogwash database [2].

In the 361 pattern, 8,263 character system, the design automation system can generate the character graph, partition, and create the final synthesizable, optimized VHDL in less than 10 seconds on a desktop-class Pentium III 800 MHz with 256 MB RAM. The 1,000 pattern, 19,584 character rule set requires about 30 seconds. The 2,000 pattern, 39,278 character rule set requires about 90 seconds.

All of the code except the partitioning tool is written in Perl, a runtime language. The automatic design tools occupy only a small fraction of the total hardware development time, as the place and route of the design to FPGA takes much longer, roughly 10 minutes to complete for the 361 pattern, 8,263 character design, and several hours for larger designs. A partial solution to this problem lies in incremental synthesis, a strategy for reducing hardware generation costs through reuse of a previous generation's place and route information.

5.1 Optimized Incremental Design

A problem with recent designs utilizing hardwired comparator modules is in the requirement for a full place-and-route to make any change to the design, no matter how small. Because of the exceptional area and time efficiency possible with the customized design paradigm, this issue has been largely ignored. There are various approaches to allowing for changes in the rule set. The Bloom filter approach [10] allows changes through use of hash-indexed memory. Another approach uses memory-stored patterns that are retrieved and compared as necessary [30].

Our approach is based on finding the optimal partition to modify and then only placing-and-routing that section. By

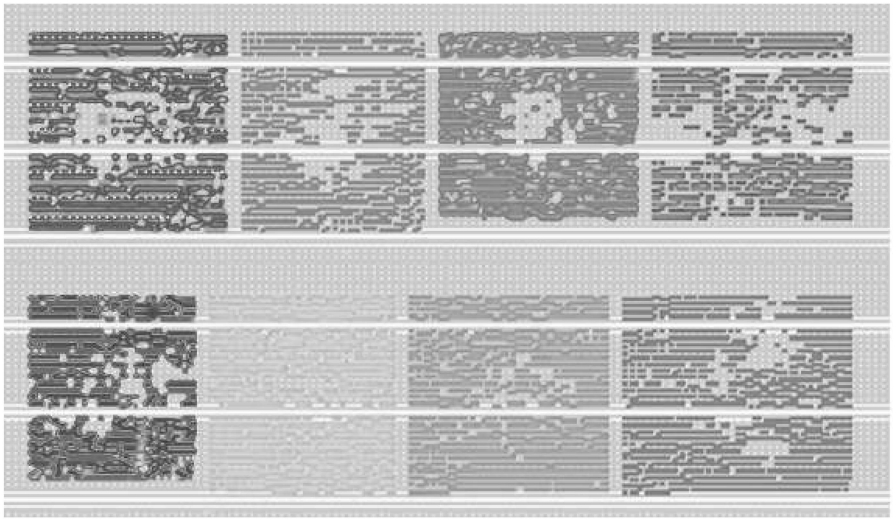


Fig. 9. Area constraints and placed modules on Virtex II Pro 50 device (image is cropped).

leaving the unmodified partitions alone, the overall time for recompiling a rule set is significantly reduced.

For the situation of adding a rule, we utilize the min-cut partitioned graph produced for the initial design. Determining the optimal partition to add a new pattern to is a fairly trivial task, requiring only a consideration of characters already mapped to the partition and preexisting prefixes. The partition least modified by the addition of the new rule is determined by comparing the predecoded bits already within the partition.

We formalize this operation as follows, where S_{p^*} is as defined in (1), the set of characters required to represent the new pattern p^*. The set difference between the characters currently represented in P_i and the characters that are present in S_{p^*} is δ_i. The partition which will require the addition of the minimum number of new characters is the optimal partition P_j. The optimal partition is selected from the set of partitions P:

$$\delta_i = (S_{p^*} \backslash P_i). \tag{5}$$

$$\text{Find j such that } |\delta_j| = \min_{i=0}^{P} |(\delta_i)|. \tag{6}$$

$$\text{Characters to add to partition } j \text{ are in } \delta_j. \tag{7}$$

This VHDL code describing this partition is then regenerated by the tool, requiring insignificant time. If the new pattern shares a prefix with some other pattern in the partition, the partial result of the previous pattern is mapped to the new pattern, reducing new wiring. The removal of rules is far easier, only the connections to the final result encoder are removed. The new partition code is sent to the incremental synthesis and place-and-route functions of Xilinx ISE 6.2. The tool only resynthesizes the modified modules. Because of the previously defined area constraints, each pipeline module is independent of the others. Thus, only the routing in the modified module requires place and route.

In our current implementation, we first manually create the area constraints on the device. Each partition is generated as an individual module, and each module is allotted sufficient space on the device. The Xilinx PACE tool estimates how many slices are required, and we find it useful to provide a 20 percent allowance to ease routing congestion within the module. The entire design is synthesized, placed, and routed, and the guide files are maintained for the next incremental change.

An example of the area constraints and finished placement is illustrated in Fig. 9 on the Virtex II Pro 50 device. Each box is the manually defined area constraint. Within each box are the placed logic elements used by that partition. Notice that the percentage utilization of each constrained area varies; this can also be used to determine the appropriate partition to which a new pattern should be added.

Our results show that for a change of one pattern in a single partition in system with p partitions (assuming the partitions are balanced), the time for place-and-route is reduced to $1/p$ plus some overhead for reprocessing the guide files. This overhead can be fairly large (approaching 50 percent of the total PAR time). In terms of real time, the first place-and-route takes slightly more than eight hours and modifying one partition requires one hour to process the guide file and one hour to place-and-route the single partition. However, without the use of incremental place-and-route, the system would require a completely new place-and-route, or p times additional time.

There are other techniques using TCAMs [31], [6] and memory-based techniques [9], [10], [30] that achieve fast modification of patterns using memory-based approaches. However, these approaches are often constrained by the amount of on-chip memory resources or require an external RAM.

6 Summary of Results

This section presents results based on partitioning-only unary and tree architectures generated automatically by our tool. The results are based on rule sets of 204, 361, 602 and 1,000 patterns, subsets of the Nikto rule set of the Hogwash database [2].

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 [32]. We have done performance verification on the Xilinx ML-300 platform [33]. This board contains a Virtex II Pro XC2VP7, a small device on the Virtex II spectrum. We have subsets of the database (as determined to fit on the device) and they

TABLE 4
Partitioning-Only Unary Architecture: Clock Period (ns) and Area (Slices) for Various Numbers of Partitions and Patterns in Sets

		Number of Patterns in Ruleset				
	No. Partitions	204	361	602	1000	2000
Clock Period	1	4.179	5.175	5.33	5.41	6.6
	2	4.457	4.497	5.603	5.17	5.79
	3	3.863	4.798	4.556	5.6	5.2
	4	3.986	4.244	5.063	5.22	5.35
	8	4.174	5.193	4.602	4.93	5.1
Area	1	800	1198	2466	4028	6260
	2	957	1394	3117	4693	7017
	3	1043	1604	3607	5001	7261
	4	1107	1692	4264	5285	8977
	8	2007	1891	5673	6123	11021
Total chars in ruleset:		4518	8263	12325	19584	39278
Characters per slice (max):		5.64	6.89	4.99	4.86	6.27

execute correctly at the speeds documented in Table 4. In our tests, input data was stored in an onboard RAM and then streamed into the IDS architecture. A commodity network processor would provide a reordered network stream to the FPGA in a deployable system, as they support the high bandwidth serial communications pins provided by Xilinx Virtex devices.

We utilized the tool set to generate architectural descriptions for various numbers of partitions. Table 4 contains the system characteristics for partitioning-only unary designs and Table 5 contains our results for the tree-based architecture. As our designs are much more efficient than other shift-and-compare architectures, the most important comparisons to make are between "1 Partition" (no partitioning) and the multiple partition cases. Clearly, there is an optimal number of partitions for each rule set; this tends toward two or three below 400 patterns and toward eight partitions for the 1,000 pattern rule set. The clock speed gained through partitioning can be as much as 20 percent, although this is at the cost of increased area. The tree approach produces greater increases in clock frequency at a lower area cost. The 602 pattern rule set shows the most dramatic improvements when using the tree approach, reducing area by almost 50 percent in some cases; the general improvement is roughly 30 percent. Curiously, the unpartitioned experiments actually show an increase in area due to the tree architecture, possible due to the increased fanout when large numbers of patterns are sharing the same prefixes in one pipeline.

In Table 4, we see that the maximum system clock is between 200 and 350 MHz for all designs. The system area increases as the number of partitions increases, but the clock frequency reaches a maximum at three and four partitions for sets under 400 rules and at eight partitions for larger rule sets. Our clock speed, for an entire system, is in line with the fastest single-comparator designs of other research groups.

In Table 4, the results are collected and sent to the output by an OR-tree. In this architecture, eight match signals are collected and OR-ed together by two four-input lookup tables. These blocks of eight fit the FPGA architecture well. In this style, a controller determines which pattern was matched after the fact, based on the known delay of the pipeline and a simple lookup in the list of patterns sorted on the reverse ordering of their characters.

Another option, more common in the field, is to provide an encoding of the results. This can be very expensive, up to 90 percent more expensive in terms of area. We have

TABLE 5
Tree Architecture: Clock Period (ns) and Area (slices) for Various Numbers of Partitions and Patterns in Sets

		Number of Patterns in Ruleset				
	No. Partitions	204	361	602	1000	2000
Clock Period	1	4.89	5.25	5.43	5.35	6.96
	2	4.18	4.27	4.80	4.22	5.2
	3	3.99	4.15	4.32	5.08	4.97
	4	4.10	4.10	4.54	4.69	5.44
	8	4.03	4.43	4.63	4.9	5.51
Area	1	773	1165	2726	4654	5967
	2	729	1212	2946	3170	7008
	3	931	1410	2210	5010	8391
	4	1062	1345	2316	5460	9276
	8	1222	1587	2874	6172	11502
Total chars in ruleset:		4518	8263	12325	19584	39278
Characters per slice (max):		6.19	7.09	5.577	6.17	6.58

TABLE 6
Partitioning-Only Unary Architecture: Clock Period (ns) and Area (slices) for Various Numbers of Partitions and Patterns Sets

		Number of Patterns in Ruleset				
	No. Partitions	204	361	602	1000	2000
Clock Period						
(OR-tree)	1	4.18	5.18	5.33	5.41	6.6
(Encoder)	1	5.09	4.5	4.91	5.19	7.96
Area						
(OR-tree)	1	800	1198	2466	4028	6260
(Encoder)	1	1246	1972	4017	9789	11261
Total chars in ruleset:		4518	8263	12325	19584	39278

TABLE 7
Performance Comparison of Various Approaches

Design	Bytes	Device	Throughput	LC/Chr	Perf.
USC Unary	4518	V II-Pro 100	2.07 Gb/s	0.46	4.5
USC Unary	39278	V II-Pro 100	1.56 Gb/s	0.56	2.79
USC Unary w/ Encoder	4518	V II-Pro 100	1.6 Gb/s	0.55	2.91
USC Unary w/ Encoder	39278	V II-Pro 100	1.0 Gb/s	0.57	1.75
USC Unary - Prefix Tree	4518	V II-Pro 100	2.00 Gb/s	0.42	4.76
USC Unary - Prefix Tree	39278	V II-Pro 100	1.53 Gb/s	0.31	4.94
USC Unary - 4 byte	4518	V II-Pro 100	6.1 Gb/s	0.72	8.4
USC Unary - 4 byte	19584	V II-Pro 100	4.5 Gb/s	0.65	6.9
USC Unary - 8 byte	4518	V II-Pro 100	10.3 Gb/s	2.0	5.15
USC Unary - 8 byte	8263	V II-Pro 100	8.8 Gb/s	1.87	4.72
Los Alamos (FPL '03)[6]	640	V E 1000	2.2 Gb/s	15.2	0.15
UCLA - 4 byte (FPL '02)[24]	1611	Altera EP20k	2.88 Gb/s	10	0.29
UCLA - 4 byte (FCCM '04)[3]	19021	Sprtn 3 2000	3.2 Gb/s	0.71	4.5
U/Crete - 4 byte (FPL '03) [26]	2457	V II 6000	8 Gb/s	19.4	0.41
U/Crete - 4 byte (FCCM '04) [25]	18032	V II 6000	9.7 Gb/s	3.56	2.72
GATech - 1 byte (FPL '03)[11]	17537	V 1000	0.8 Gb/s	1.1	0.72
GATech - 4 byte (FCCM '04)[8]	17537	V II 8000	7.0 Gb/s	3.1	2.23

Area/character is in logic cells (one slice is two logic cells) and performance (in Gbps/cell/character). Throughput is assumed to be constant over variations in pattern size. If not specified, the USC designs use an OR result tree. USC designs use optimal #/partitions From Tables 5 and 4.

implemented a priority encoder to allow for equivalent comparisons between designs. Some designs preprocess the rule set to ensure that multiple matches cannot be active in a given block at any given time [3]. This allows the use of a simple encoder instead of the priority encoder, at some area and clock savings. We have designed a heavily pipelined priority encoder design that does not cause much reduction in time performance and has a moderate area impact.

In Table 6, we compare single partition (unpartitioned) OR-tree results with the corresponding priority encoder results.

Table 7 contains comparisons of our system-level design versus individual comparator-level designs from other researchers. We only compare against designs that are architecturally similar to a shift-and-compare discrete matcher, that is, where each pattern at some point asserts an individual signal after comparing against a sliding window of network data. We acknowledge that it is impossible to make fair comparisons without reimplementing all other designs. We have defined performance as throughput/area, rewarding small, fast designs. In this metric, architectures produced by our tools are exceptional.

Our tree design occupies roughly one slice per 5.5 to 7.1 characters. While this approach is somewhat limited by only accepting eight bits per cycle,the area efficiency allows smaller sets of patterns to be replicated on the device, as in Section 4.2.

7 Conclusion

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

After a small preprocessing phase, our tool automatically generates designs with competitive clock frequencies that are a minimum of two times more area efficient than any other discrete-comparator-based shift-and-compare design. By trading some of the area efficiency of the basic architecture for throughput, we can reach sustained throughput rates above 10 Gbps, with area-time performance still much higher than any other implementation.

Acknowledgments

This research was supported by the US National Science Foundation Information Technology Research Program under award number ACI-0325409 and in part by an equipment grant from Hewlett-Packard. Portions of this paper appear as preliminary versions in FCCM '04, FPL '04, and ANCS '05.

REFERENCES

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

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

[3] Y. Cho and W.H. Mangione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," *Proc. 12th Ann. IEEE Symp. Field Programmable Custom Computing Machines (FCCM '04),* pp. 125-134, 2004.

[4] L. Schaelicke, K. Wheeler, and C. Freeland, "SPANIDS: A Scalable Network Intrusion Detection Loadbalancer," *Proc. Computing Frontiers Conf.,* pp. 315-322, 2005.

[5] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *Proc. 10th Ann. Field-Programmable Custom Computing Machines (FCCM '02),* pp. 111-120, 2002.

[6] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection," *Proc. 13th Ann. ACM/SIGDA Int'l Conf. Field-Programmable Logic and Applications (FPL '03),* pp. 404-413, 2003.

[7] R. Sidhu, A. Mei, and V.K. Prasanna, "String Matching on Multicontext FPGAs Using Self-Reconfiguration," *Proc. Seventh Ann. ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays (FPGA '99),* pp. 217-226, 1999.

[8] C.R. Clark and D.E. Schimmel, "Scalable Parallel Pattern Matching on High Speed Networks," *Proc. 12th Ann. IEEE Symp. Field Programmable Custom Computing Machines (FCCM '04),* pp. 249-257, 2004.

[9] Z.K. Baker and V.K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," *Proc. 12th Ann. ACM Int'l Symp. Field-Programmable Gate Arrays (FPGA '04),* pp. 223-232, 2004.

[10] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Implementation of a Deep Packet Inspection Circuit Using Parallel Bloom Filters in Reconfigurable Hardware," *Proc. 11th Ann. IEEE Symp. High Performance Interconnects (HOTi '03),* pp. 49-51, 2003.

[11] C.R. Clark and D.E. Schimmel, "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns," *Proc. 13th ACM/SIGDA Int'l Conf. Field-Programmable Logic and Applications (FPL '03),* pp. 956-959, 2003.

[12] B. So, M.W. Hall, and P.C. Diniz, "A Compiler Approach to Fast Design Space Exploration in FPGA-Based Systems," *Proc. ACM Conf. Programming Language Design and Implementation (PLDI '02),* pp. 165-176, June 2002.

[13] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, and P. Banerjee, "FPGA Hardware Synthesis from MATLAB," *Proc. VLSI Design Conf.,* pp. 299-304, Jan. 2001.

[14] P. Bellows and B. Hutchings, "JHDL: An HDL for Reconfigurable Systems," *Proc. Sixth Ann. IEEE Symp. Field Programmable Custom Computing Machines (FCCM '98),* pp. 175-184, 1998.

[15] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," *Proc. 11th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '03),* pp. 31-38, 2003.

[16] Global Velocity, http://www.globalvelocity.info/, 2005.

[17] P. Jones, S. Padmanabhan, D. Rymarz, J. Maschmeyer, D. Schuehler, J. Lockwood, and R. Cytron, "Liquid Architecture," *Proc. 18th Ann. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS '04),* pp. 202-210, 2004.

[18] Y. Ha, P. Schaumont, M. Engles, S. Vernalde, F. Patargent, L. Rijnders, and H.D. Man, "A Hardware Virtual Machine for Networked Reconfiguration," *Proc. IEEE Conf. Rapid System Prototyping (RSP '00),* pp. 194-199, June 2000.

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

[20] R. Boyer and J. Moore, "A Fast String Searching Algorithm," *Comm. ACM,* vol. 20, no. 10, pp. 762-772, Oct. 1977.

[21] A. Aho and M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Comm. ACM,* vol. 18, no. 6, pp. 333-340, June 1975.

[22] R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching using FPGAs," *Proc. Ninth Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '01),* pp. 227-238, 2001.

[23] Z.K. Baker and V.K. Prasanna, "A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs," *Proc. 12th Ann. IEEE Symp. Field Programmable Custom Computing Machines (FCCM '04),* pp. 135-144, 2004.

[24] Y. Cho, S. Navab, and W. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," *Proc. 12th ACM/SIGDA Int'l Conf. Field-Programmable Logic and Applications (FPL '02),* pp. 452-461, 2002.

[25] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," *Proc. 12th Ann. IEEE Symp. Field Programmable Custom Computing Machines (FCCM '04),* pp. 258-267, 2004.

[26] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System," *Proc. 13th Ann. ACM/SIGDA Int'l Conf. Field-Programmable Logic and Applications (FPL '03),* pp. 880-889, 2003.

[27] D. Knuth, J. Morris, and V. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Computing,* pp. 323-350, 1977.

[28] 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/, 2004.

[29] M.E. Attig and J.W. Lockwood, "A Framework for Rule Processing in Reconfigurable Network Systems," *Proc. 13th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '05),* pp. 225-234, 2005.

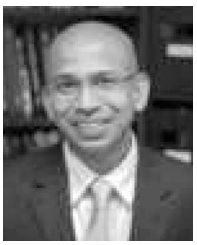
[30] Y. Cho and W.H. Mangione-Smith, "Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network," *Proc. 13th Ann. IEEE Symp. Field Programmable Custom Computing Machines (FCCM '05),* pp. 215-224, 2005.

[31] F. Yu, R. Katz, and T. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," *Proc. 12th IEEE Int'l Conf. Network Protocols (ICNP),* pp. 174-183, 2004.

[32] Xilinx Inc., "Virtex II Pro Series FPGA Devices," http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+%Pro+FPGAs, 2004.

[33] Xilinx Inc., "ML-300 Development Board," http://www.xilinx.com/ml300, 2004.

Zachary K. Baker received the MS degree in electrical engineering from the University of Southern California in August 2002. He is now a PhD candidate at USC, where he is currently a research assistant (since March 2001). His research interests include hardware architectures for pattern matching, intrusion detection, and data mining. He has published and presented his work at several international workshops and conferences. He is a student member of the IEEE.



Viktor K. Prasanna received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation at the Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University. He is a professor of electrical engineering and professor of computer science at the University of Southern California (USC). His research interests include high performance computing, parallel and distributed systems, network computing and embedded systems. He is the steering committee cochair of the International Parallel and Distributed Processing Symposium (IPDPS) (merged IEEE International Parallel Processing Symposium (IPPS) and Symposium on Parallel and Distributed Processing (SPDP)). He is the steering committee chair of the International Conference on High Performance Computing (HiPC). He serves on the editorial board of the *Journal of Parallel and Distributed Computing*. He is the editor-in-chief of the *IEEE Transactions on Computers*. He was the founding chair of the IEEE Computer Society's Technical Committee on Parallel Processing. He is a fellow of the IEEE.