Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs[1]

Zachary K. Baker and Viktor K. Prasanna
zbaker@usc.edu, prasanna@ganges.usc.edu

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, implemented with a tool suite we release for public use, allows for customized performance through more efficient communication and extensive reuse of hardware components for dramatic increases in area-time performance.

Keywords: Intrusion Detection, Graph Algorithms, Partitioning, Performance, FPGA Design

[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. Portions of this paper appear as preliminary versions in FCCM '04 and FPL '04.

1 Introduction

The continued discovery of programming errors in network-attached software has driven the introduction of increasingly powerful and devastating attacks [1, 2, 3]. Attacks can cause destruction of data, clogging of network links, and future breaches in security. In order to prevent, or at least mitigate, these attacks, a network administrator can place a firewall or Intrusion Detection System at a network choke-point such as a company's connection to a trunk line (Figure 1). A firewall's function is to filter at the header level; if a connection is attempted to a disallowed port, such as FTP, the connection is refused. This catches many obvious attacks, but in order to detect more subtle attacks, an Intrusion Detection System (IDS) is utilized. The IDS differs from a firewall in that it goes beyond the header, actually searching the packet contents for various patterns that imply an attack is taking place, or that some disallowed content is being transferred across the network. Current IDS pattern databases reach into the thousands of patterns, providing for a difficult computational task.

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]. However, the study of the *automatic design* of efficient, flexible, and powerful system architectures is still in its infancy.

Snort, the open-source IDS [9], and Hogwash [10] 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.

These algorithms require significant computational resources. To support heavy

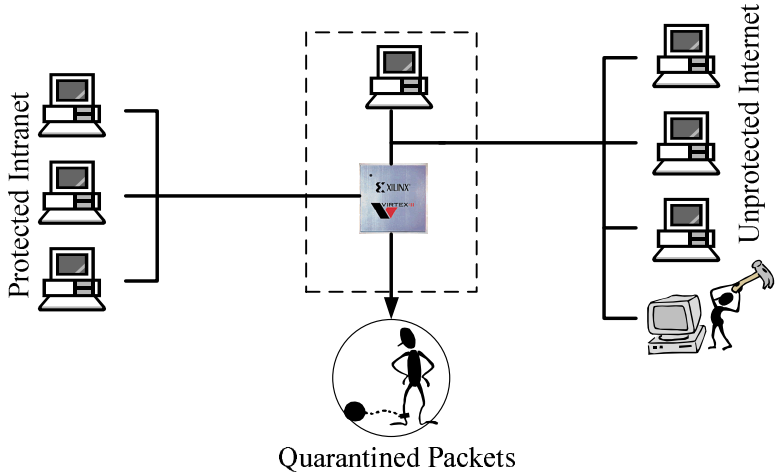

Figure 1: Intrusion detection systems protect networks from external threats. The use of FPGA allows a system to take advantage of massive parallelism.

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 [11], a Dual 1GHz Pentium III system, using 845 patterns, runs at only 50 Mbps. In Section 6 we show that a single FPGA device can support multi-Gigabit rates with 1000 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. A general ASIC design would be fast but not suitable due to the dynamic nature of the ruleset – as new vulnerabilities and attacks are identified, new rules must be added to the database and the device configuration must be regenerated. However, a Field-Programmable Gate Array (FPGA) allows for exceptional performance due to the parallel hardware nature of execution as well as the ability to customize the device for a particular set of patterns. An FPGA can provide near-ASIC performance and parallelism, along with the ability to modify the hardware to a particular set of patterns.

Early FPGA designs in the field [4, 7, 12] 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.

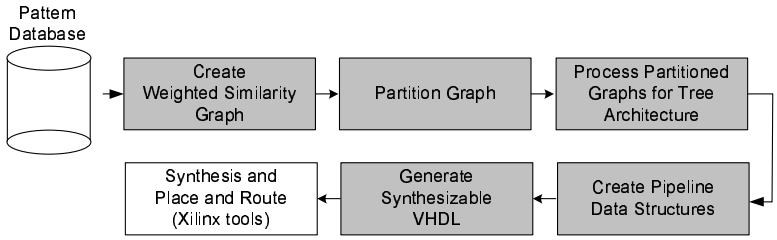

Figure 2: Automated optimization and synthesis of partitioned system

Our basic architecture is a pre-decoded multiple-pipeline shift-and-compare matcher. While this approach can be considered "brute force" compared to a state machine approach [4, 13, 14] or a hashing approach [15], 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 basic architecture is extended in various ways. To allow for better area performance, we present a partial tree architecture that allows for significant reduction in redundant comparisons by independently matching prefixes that are shared across a range of 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. To provide high throughput with exceptional area efficiency, we provide an architecture that sacrifices exactness and allows for an increased false positive rate.

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 naïve hardware generation, produce higher efficiency in terms of patterns matched per unit area and unit time.

By intelligently processing an entire ruleset (Figure 2), our tool partitions a ruleset 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 ruleset. 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 pre-processing, the system is generated from templates. By applying various graph partitioning operations and trie techniques to the problem, the tool more effectively optimizes large ruleset as compared to naïve approaches.

This paper describes a methdology 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 [9] and Hogwash [10] 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 [16, 17, 18], but domain-specific tools have a distinct performance advantage in network security. Automated IDS designs have been explored in [4], using automated generation of Non-deterministic 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)[19]. 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 logic burden by combining common prefixes to form matching trees. This is part of the pre-processing approach we take in this paper.

Another automated hardware approach, in [8], uses more sophisticated algorithmic techniques to develop multi-gigabyte pattern matching tools with full TCP/IP net-

work 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 7-8 minutes. As their tools have been commercialized in [20], they are not freely available to the community. However, their development work includes network reconfiguration [21], also explored in [22]. 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 [23]. They have implemented a software tree-searching strategy that uses elements of the Boyer-Moore [24] and Aho-Corasick [25] 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 [26], [4] 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 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 [14]. This paper addresses the 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 [7], 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 can only provide 32 20-byte matching units on the Xilinx Virtex XCV1000E FPGA device.

In [11, 27, 28], hardwired designs are developed that provide high area efficiency and high time performance by using replicated hardwired comparators in a pipeline structure. The hardwiring provides high area efficiency, but are difficult to reconfigure. Hardwiring also allows a unit to accept more than one byte per cycle, through replication. A bandwidth of 32 bits per cycle can be achieved with four hardwired comparators, each with the same pattern offset successively by 8 bits, allowing the running time to be reduced by 4x for an equivalent increase in hardware. These designs have adopted some strategies for reducing redundancy through pre-design optimization. The work in [5] was expanded in [11] 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 [14] in the context of finite automata. The use of large, pipeline brute-force comparators for high speed was initiated in [5] and continued in [6, 28], although [28] utilizes SRL16 shift registers where we uti-

lize single-cycle delay flip-flops. Their work utilizes a less elborate 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 [4]. We explored the partitioning of patterns in the pre-decoded domain in [27]. 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 pre-decoded shift-and-compare architecture, and a variation using a tree-based area optimization. In this architecture, a character enters the system and is "pre-decoded" 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 pre-matched 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.

With our domain specific analysis and generation tool, extensive automation partitions a rule databases 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

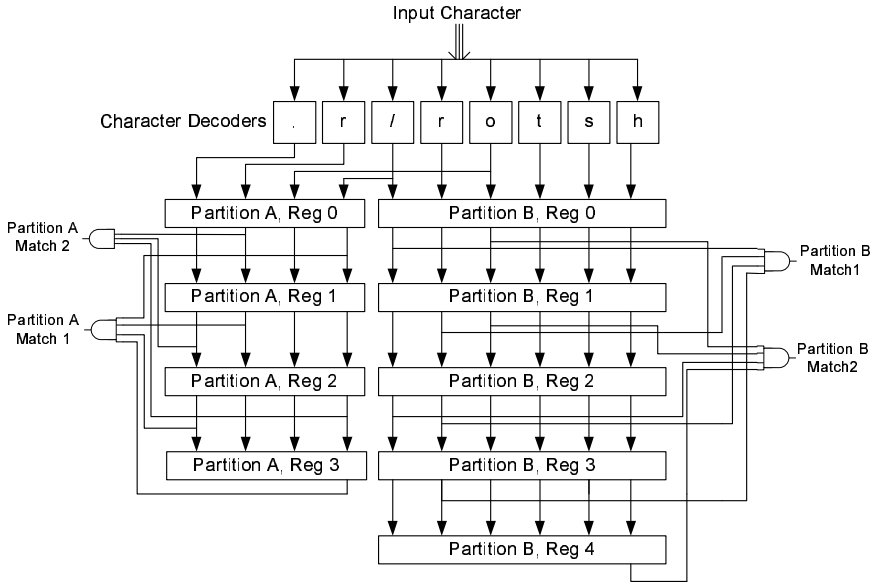

Figure 3: General architecture of our pipelined comparators design. Characters are converted to single bits in the first stage and then fed into the pipeline, where they become operands for the pattern comparators.

algorithm [13, 29], 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 (Figure 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 [14, 27] has utilized pre-decoded, single-bit character re-encodings in lieu of delivering 8-bit wide datapath 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 pre-decoding 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. This allows efficient multi-byte 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 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 2 and 8. Using the Metis partitioning library to solve the min-cut problem [30], 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 rulesets of less than 400 patterns. Conversely, for the 603 and 1000 pattern rulesets, the maximum time performance is achieved with eight partitions. However, as the area increases moderately as the number of partitions

increases, the area-time tradeoff must be considered.

Our partitioning strategy, can partition a ruleset to roughly 30 bits, or about the same amount of data routing as one of the 4-byte replicated architectures ([5, 6]). However, the matching units are least 8x smaller (32 down to 4 bits for the design in [5]), and we have removed the control logic of a KMP-style design such as [13].

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 straight-forward matching technique used in [5, 6], but moves the character decoding to the first stage in the pipeline (as in [14]), and reduces the overall size of the individual comparators by one-eighth, as illustrated in Figure 3.

First, the patterns are partitioned into several groups (Figure 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:

$$S_k = \{a : a \in C \mid a \text{ appears in } k\} \tag{1}$$

$$V_R = \{p : p \in T\} \tag{2}$$

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

A vertex V is added to graph R for each pattern p in the ruleset 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, 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 Figure 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 [30].

One clear problem is that of large rulesets (>500 patterns). In these situations it is essentially impossible for a small number of partition to not 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 rulesets. 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 Equation 4 allows the partitioning algorithms to more strongly group patterns with similar initial patterns of characters. The weighting function is weak enough to not 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.

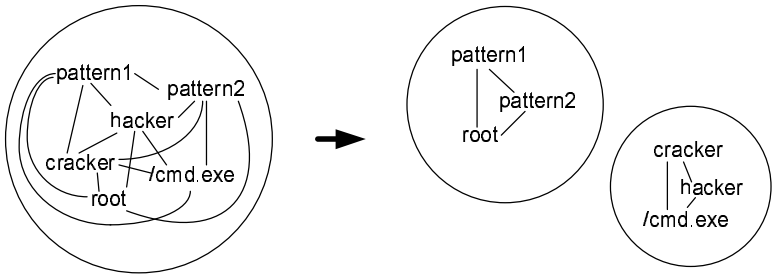

Figure 4: Partitioned graph; by reducing the cut between the partitions we decrease the number of pipeline registers

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 is still performance characteristics that can be further optimized. We explore several variations:

- Tree-based Prefix Sharing
- High-throughput Architecture
- Pre-filtering Architecture

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.

We have developed a prefix-tree strategy to find 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

	Number of Prefixes	
No. of Patterns	First Level	Second Level
204	83	126
361	204	297
602	270	421
1000	288	523

Table 1: Illustration of effectiveness of tree strategy for reducing redundant comparisons

approach is similar to the *trie* strategy utilized in [4], in which a collection of patterns is composed into a single regular expression. Their DFA 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 much higher performance.

Figure 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 together to produce the final matching information for the pattern. This is effective in reducing the area of the design because large sections of the rulesets 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 Virtex-style 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 rulesets we utilized in our tests. Second-level prefixes are only counted as

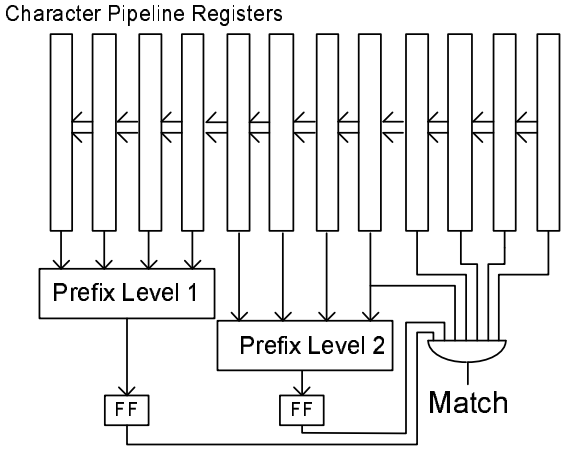

Figure 5: Illustration of tree-based hardware reuse strategy. Results from two appropriately delayed prefix blocks are delayed through registers and then combined with remaining suffix. The key to the efficiency of this architecture is that the prefix matches are reused, as well as the character pipeline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

different within the same first-level prefix. For this table, we created our rulesets using the first n rules in the Nikto ruleset [10]. There is no intentional pre-processing of the rulesets before the tool flow. The table shows that, on the average, redundant prefix comparisons can be reduced by 2 to 3x 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.

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 non-partitioned 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

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

Table 2: Tree Architecture: Clock period (ns) and area (slices) for various numbers of partitions and patterns sets

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 datapath width into the device. The use of k-byte data words complicates the design, however, because now 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 1000 pattern database now presents roughly the routing complexity of a $1000k$ pattern database.

We illustrate our 4-way architecture in Figures 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 3. For these experiments, we have utilized the optimal number of partitions from the basic unary architecture. Overall, it is clear

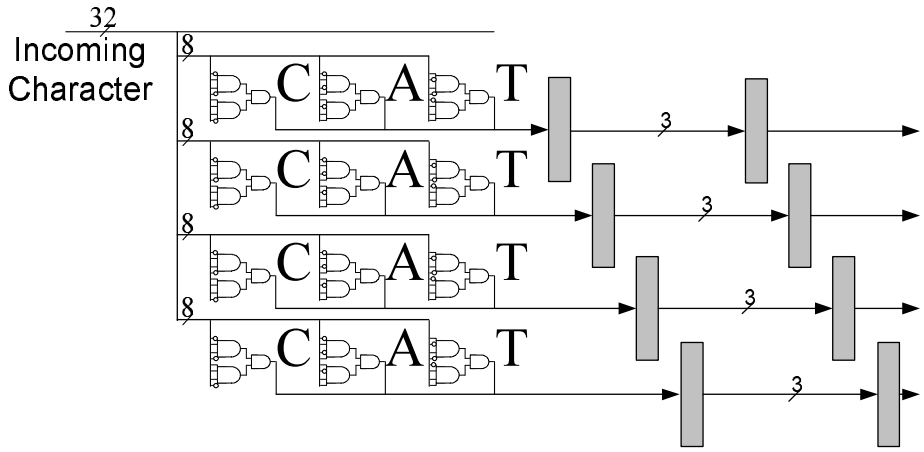

Figure 6: Illustration of 4-way front end. Character decoders are replicated to allow for four different beginning offsets.

	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 3: Performance results for 4 and 8-way architectures(32 and 64 bit datapaths, respectively)

that the increase in area is less than k times the 8-bit architecture, and the decline in clock frequency is acceptable.

4.3 Pre-filtering Architecture

In the high-throughput variation, accuracy in exact-string matching is assured through replication. A string that matches one of the desired patterns 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

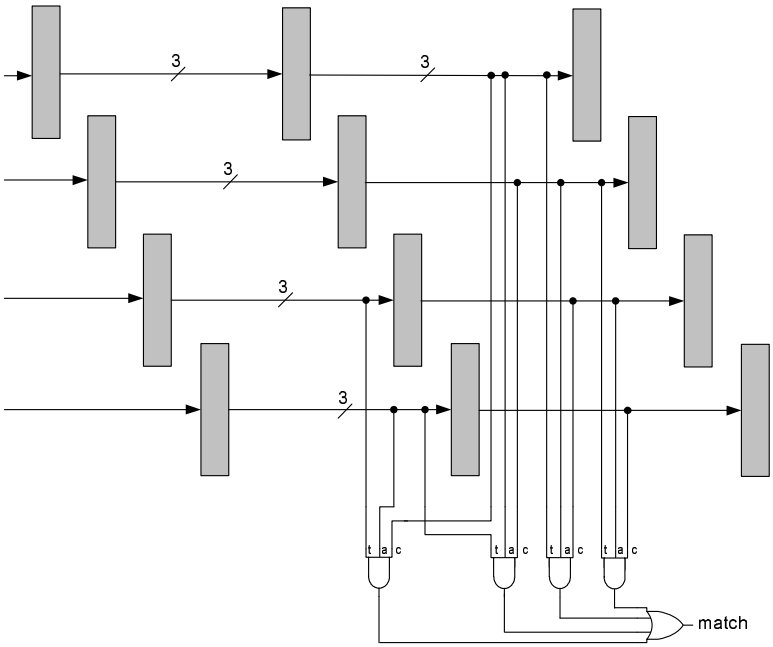

Figure 7: Illustration of 4-way back end matchers. The pipeline moves each block of decoded characters forward by four character positions, and pattern comparators select each decoded character line appropriately.

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 significantly. 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 data-path 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 system that previously only allowed one line to be active in any given pipeline stage. In this setup, each pattern can be properly matched at 4 differ-

ent 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 the right character in the wrong offset happen to be triggered. That is, "cat" and "cct" or "caa" will all trigger a match for the pattern "cat". 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 [15], 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 [15] as they both produce results that have some possibility of false positives. The Bloom filter provides matching against thousands of patterns at 2.4 Gb/s, at an 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.

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 ruleset total roughly 300kB in 9,000 lines, but synthesize to a minimum of 1200 slices. 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 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.

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. 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 [9, 10], creating the pattern-connection graphs and subsequently partitioning the graphs is an expensive operation, not suitable for runtime. representation, the Hogwash ruleset of roughly 7000 strings creates a graph occupying over 215MB. However, even with these large memory requirements, the process flow requires little time. ruleset of the

Hogwash database [10].

In the 361 pattern, 8263 character system, the design automation system can generate the character graph, partition, and create the final synthesizeable, optimized VHDL in less than 10 seconds on a desktop-class Pentium III 800MHz with 256 MB RAM. The 1000 pattern, 19584 character ruleset requires about 30 seconds.

All of the code code except the partitioning tool is written in Perl, a runtime language. While Perl provides powerful text processing capabilities useful for processing the rulesets, it is not known as a high performance language. A production version of this tool would not be written in a scripting 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 to FPGA takes much longer, roughly ten minutes to complete for the 361 pattern, 8263 character design. 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 hard-wired comparator modules is in the requirement for a full place-and-route to make any change, no matter how small, to the design. Because of the exceptional area and time efficiency possible with this customized design paradigm, this issue has been largely ignored.

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 pre-existing prefixes. The partition least modified by the addition of the new rule is determined by comparing the pre-decoded bits already within the partition.

We formalize this operation as follows, where S_{p^*} is as defined in Equation 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.

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

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

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

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 tree 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 re-synthesizes the modified modules. Because of the previously defined area constraints, each pipline 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 alloted sufficient space on the device. The Xilinx PACE tool estimates how many slices are required, and we find it useful to provide a 20% 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 Figure 8 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.

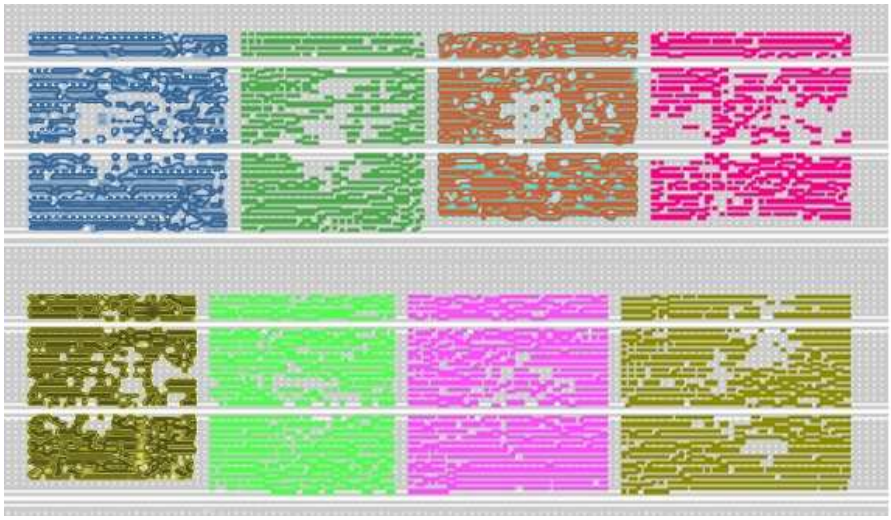

Figure 8: Area constraints and placed modules on Virtex II Pro 50 device (image is cropped)

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% of the total PAR time). However, without the use of incremental place and route, the system would require a completely new place-and-route, or p times additional time. Certainly a lower-level incremental change controller, possibly implemented in an embedded microprocessor core as in [31, 32], making the required changes directly within a partition without resorting to extensive re-processing(as in [33]) would be a more effective solution.

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 ruleset of 204, 361, 602

and 1000 patterns, subsets of the Nikto ruleset of the Hogwash database [10].

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 [34]. We have done performance verification on the Xilinx ML-300 platform [35]. 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 execute correctly at the speeds documented in Table 4.

We utilized the tool set to generated architectural descriptions for various numbers of partitions. Table 4 contains the system characteristics for partitioning-only unary designs, and Table 2 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 ruleset; this tends toward 2-3 below 400 patterns and toward 8 partitions for the 1000 pattern ruleset. The clock speed gained through partitioning can be as much as 20%, 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 ruleset shows the most dramatic improvements when using the tree approach, reducing area by almost 50% in some cases; the general improvement is roughly 30%. 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 250MHz for all designs. The system area increases as the number of partitions increases, but the clock frequency reaches a maximum at 3 and 4 partitions for sets under 400 rules and at 8 partitions for larger rulesets. Our clock speed, for an entire system, is in line with the fastest single-comparator designs of other research groups.

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

Table 4: Partitioning-only Unary Architecture: Clock period (ns) and area (slices) for various numbers of partitions and patterns sets

Table 5 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.

The smallest of designs in the published literature providing individual match signals is in [14], in which a state machine implements a Non-deterministic Finite Automata in hardware. That design occupies roughly one slice per 2.5 characters. Our tree design occupies roughly one slice per 5.5-7.1 characters, making it significantly more effective. While this approach is somewhat limited by only accepting 8 bits per cycle, the area efficiency allows smaller sets of patterns to be replicated on the device, as in Section 4.2.

Design	Throughput	Unit Size	Performance
USC Unary	2.07 Gb/s	7.3	283
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
USC Unary (Tree)	2.00 Gb/s	6.6	303
Los Alamos (FPL '03)[7]	2.2 Gb/s	243	9.1
UCLA (FPL '02)[5]	2.88 Gb/s	160	18.0
UCLA w/Reuse (FCCM '04)[11]	3.2 Gb/s	11.4	280
U/Crete (FPL '03) [6]	10.8 Gb/s	269	40.1
U/Crete (FCCM '04) [28]	9.7 Gb/s	57	170
GATech (FCCM '04) [14]	7.0 Gb/s	50	140

Table 5: Pattern size, average unit size for a 16 character 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.

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 2x 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.

We release the collection of tools used in this paper to the community at

`http://halcyon.usc.edu/~zbaker/idstools`

References

[1] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the Slammer worm," *IEEE Security & Privacy Magazine*, vol. 1, no. 4, July-Aug 2003.

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

[3] Z. Chen, L. Gao, and K. Kwiat, "Modeling the spread of active worms," *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, April 2003.

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

[5] Y. Cho, S. Navab, and W. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," in *Proceedings the Tenth ACM/SIGDA International Conference on Field-Programmable Logic and Applications (FPL '02)*, 2002.

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

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

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

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

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

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

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

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

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

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

[16] P. Moisset, J. Park, and P. Diniz, "Very High-Level Synthesis of Control and Datapath Structure for Reconfigurable Logic Devices," in *Proceedings of the Second Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)*, Oct 1999.

[17] B. So, M. W. Hall, and P. C. Diniz, "A Compiler Approach to Fast Design Space Exploration in FPGA-based Systems," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'02)*, June 2002.

[18] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, and P. Banerjee, "FPGA Hardware Synthesis from MATLAB," in *Proceedings of VLSI Design Conference*, Jan 2001.

[19] P. Bellows and B. Hutchings, "JHDL: An HDL for Reconfigurable Systems," in *Proceedings of the Sixth Annual IEEE Symposium on Field Programmable Custom Computing Machines 1998 (FCCM '98)*, 1998.

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

[21] P. Jones, S. Padmanabhan, D. Rymarz, J. Maschmeyer, D. Schuehler, J. Lockwood, and R. Cytron, "Liquid Architecture," in *Proceedings of the 18th Annual IEEE International Parallel and Distributed Processing Symposium (IPDPS '04)*, 2004.

[22] Y. Ha, P. Schaumont, M. Engles, S. Vernalde, F. Patargent, L. Rijnders, and H. D. Man, "A Hardware Virtual Machine for Networked Reconfiguration," in *Proceedings of the IEEE Conference on Rapid System Prototyping (RSP 2000)*, June 2000.

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

[24] Robert S. Boyer and J. Strother Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[25] A. V. Aho and M. J. Corasick, "Efficient String Matching: an Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

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

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

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

[29] D. Knuth, J. Morris, and V. Pratt, "Fast Pattern Matching in Strings," in *SIAM Journal on Computing*, 1977.

[30] G. Karypis, R. Aggarwal, K. Schloegel, V. Kumar, and S. Shekhar, "METIS Family of Multilevel Partitioning Algorithms," 2004, http://www-users.cs.umn.edu/~karypis/metis/.

[31] P. James-Roxby, G. Brebner, and D. Bemmann, "Time-Critical Software Deceleration in an FCCM," in *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.

[32] B. Blodget, S. McMillan, and P. Lysaght, "A Lightweight Approach for Embedded Reconfiguration of FPGAs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, 2003.

[33] P. Suaris, L. Liu, Y. Ding, and N. Chou, "Incremental Physical Resynthesis for Timing Optimization," in *The Twelfth Annual ACM International Symposium on Field-Programmable Gate Arrays (FPGA '04)*, 2004.

[34] The Xilinx Corporation, "Virtex II Pro Series FPGA Devices," 2004, http://www.xilinx.com/xlnx/xil˙prodcat˙landingpage.jsp?title=Virtex-II+%Pro+FPGAs.

[35] ——, "ML-300 Development Board," 2004, http://www.xilinx.com/ml300.