NETWORK INTRUSION DETECTION LEVERAGING TARGETED REGULAR EXPRESSIONS ON FPGAS

by

Jack Gardel

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Electrical and Computer Engineering

Elmore Family School of Electrical and Computer Engineering

West Lafayette, Indiana

August 2025

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Sanjay Rao, Co-Chair

Elmore School of Electrical and Computer Engineering

Dr. Vishal Shrivastav, Co-Chair

Elmore School of Electrical and Computer Engineering

Dr. Mark Johnson

Elmore School of Electrical and Computer Engineering

Approved by:

Dr. Milind Kulkarni

To my Lord Jesus Christ who saves me and the Blessed Virgin Mary who intercedes for me

ACKNOWLEDGMENTS

Firstly, I am thankful for Dr Sanjay Rao for providing me with this research opportunity and I am thankful for both Dr Rao and Dr Vishal Shrivastav for their advice and funding; especially during the last months of the summer. I am also thankful for Dr. Mark Johnson who helped me with leading much of my computer architecture course as a TA.

I thank my peers who worked on this project with; Syed Usman Jafri who tested much of our designs and obtained countless results used in guiding this research as well as Weigao Su who provided assistance and expertise with realizing our designs in application.

I would finally like to thank those outside of academia who helped encourage me especially when I was most overwhelmed; my mother and father Julie and Robert Gardel, my parish priest Fr Peter Lewitzke, and the many friends who helped me along the way.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Current software-based IDS solutions for protecting networks do not scale well to meet the higher linerates of larger enterprise networks such as those at universities. Much of the bottleneck has to do with regular expressions which take the most amount of time to process. While hardware accelerators such as GRAPEFRUIT and HARE exist, they are either too slow or not dense enough to support large rulesets. Pigasus IDS uses an FPGA-based approach to filtering out packets before regex matching but suffers from trace-dependent performance. We present TRex; an FPGA-based IDS that has a filter similar to Pigasus and a custom regular-expression accelerator for this application. TRex only checks regular expressions that are relevant to each packet, allowing it to process packets in parallel on separate rules. Exploiting parallelism in this way allows TRex to support 2x throughput, lessening the demand for software to take over.

1. INTRODUCTION

Intrusion detection and prevention systems (IDS/IPS) are used as network monitors and firewalls that inspect incoming packets against a ruleset. Each rule contains multiple fields that all must be true for the rule to match. These fields can be exact match strings, TCP header checks, regular expressions, etc. Out of these fields, regular expressions take the most amount of time to check even with lots of research over the past two decades looking to accelerate this [1–5].

Many IDS's will typically employ a multi-stage approach to filtering [1, 6] where less computationally-expensive checks are performed first to hide the large overhead of regular expression processing - meaning fewer packets need to match against regular expressions. However, software-based IDS's are still shown to be inefficient and take many CPU cores to reach linerates comparable with that of large enterprise networks [6]. Our experience with campus network operations are consistent with this and require multiple servers dedicated to IDS.

Hardware solutions to accelerate regular expression matching aim to replace slower software solutions. Designs like GRAPEFRUIT [3] offer enough compactness to support full IDS rulesets but tend to suffer from throughput issues - only being able to support <10 Gbps of traffic. Unique ASIC designs such as HARE [7] offer throughput greater than 10 Gbps, but use techniques that only allow 20 or so regular expressions total in the design.

Pigasus IDS [6] does not put regular expression matching in hardware but instead does the opposite; creating an efficient multi-stage filter in hardware and pushing remaining packets into software for regular expression processing. This design claims to meet linerate demands of 100 Gbps. However, Pigasus will suffer if too many packets require regular expression processing (see §2.3.2) making Pigasus dependent on the network trace for its performance.

Considering these setbacks, we present TRex - an FPGA-based IDS that takes the filtering philosophies of Pigasus and the NFA matching of GRAPEFRUIT to process packets at linerate and be more robust to varying network traces. Since most packets only need to match against a few rules after filtering [6], TRex's "NFA match stage" will only check packets against rules that require those rules, reducing redundant rule checks. Since packets only need to check a few rules out of many, packets can be processed by TRex in parallel on separate regular expressions to increase throughput without replicating regular expressions.

Our results show that TRex can perform 2x better than Pigasus in most cases with heavy regular expression usage, supporting workloads at 100 Gbps for up to 20% of packets requiring regular expression processing §4.4. TRex can be configured to fit the performance demands of the network and the resource demands of the FPGA.

My primary contributions to this project include the design and implementation of the NFA matching stage §3.2 as well as the simulation of the NFA matching stage §4.1.

2. BACKGROUND AND MOTIVATION

2.1 FPGAs

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be programmed to function like any digital design (a CPU, an ethernet module, etc). FPGAs do this by enabling and connecting logic cells and other resources together to have the same behavior as the target design. FPGAs are commonly used to prototype ASICs before going into production, or they can be used in situations where having highly configurable hardware is preferred to doing the same thing in software.

FPGA logic cells typically contain flip-flops (FFs) and look-up tables (LUTs). Flip-flops are circuits that hold their output value constant until the incoming clock goes from voltage level 0 to voltage level 1 (also called a rising edge), at which the flip-flop will copy the input value to the output value. Look-up tables implement combinational logic such as as AND gates (F=AB) and are used in conjunction with the output of flip-flops to compute the output of the next clock-cycle, allowing for sequential logic that depends on a previous output state.

Compared to an equivalent ASIC, the FPGA design will tend to have a slower clock frequency. This is due to the standard cells being of a certain size and cannot be optimized to be smaller and closer together as an ASIC designer can achieve through different methods of laying out individual transistors. These longer connections have more parasitic effects and thus data takes longer to get from one part of the design to another, meaning the clock will be slower.

To gain advantage over software-based solutions for computation problems, FPGA engineers will employ techniques to maximize parallelism by having independent tasks run at the same time in different parts of the FPGA. If one functional block, such as an adder, needs to be used by many tasks at once, the block can be replicated to account for the higher throughput requirement. If we wanted to have the same parallelism using software, we would have to allocate another whole CPU core to carry out a duplicate task, often costing more hardware resources and taking away compute time normally given to other processes.

FPGAs may also contain various RAM resources. Xilinx's SRAM known as Block-RAM or BRAM tends to be used for high-speed applications and is typically in lower supply than dynamic-RAM or DRAM, which takes many clock cycles to access.

2.2 Finite Automaton

In discrete mathematics, finite automaton are memory-less models used to implement the behavior of a regular expression. They operate by keeping track of which state is active at any given point and transition between states when a new input character is introduced when reading a stream of characters.

The first type of finite automaton is non-deterministic finite automaton or NFAs. NFAs have typical transitions that can take a set of characters and decide whether or not to advance to the next state. However, when a regular expression branches into two possible match conditions, the NFA can have transitions that don't take any characters at all but instead "choose" the correct branch that would be matched against. We can also think of this as the NFA taking both transitions to find out which one is correct.

Another type of finite automaton is deterministic finite automaton or DFAs. DFAs do not have non-deterministic transitions and must always take one transition at a time based on the current input character. NFAs can be transformed into DFAs however the transformation into DFAs often adds complexity in the form of additional states. DFAs are technically a subset of NFAs as every DFA is a valid NFA just without any non-deterministic transitions.

2.3 IDS/IPS

Intrusion detection and prevention systems (IDS/IPS) work by checking network traffic against a ruleset. Each rule in the ruleset contains a few different fields; (i) header match which checks the TCP 5-tuple, (ii) exact match strings and (iii) regular expressions. Advanced options also exist such as depth of patterns, TCP flow state, etc [6, 8]. For any rule to match, all of it's fields must match the given traffic, resulting in an AND operation. For a packet or flow to be deemed malicious, only one rule must match the given traffic, resulting in an OR operation between rules.

Because each rule can be thought of as an AND operation, IDS systems will typically employ multi-stage filtering [6]. For instance, many IDS systems will have a stage where every exact match pattern is put into hash tables and those hash tables are used to determine if any exact-match string across all rules would match the given packet [1]. If the hash function returns false, then the packet can be flagged as benign without needing to check the rest of the header matches and regular expressions (again, due to the AND operation of rules).

Later stages in these schemes will often more complex checks that can handle less throughput than prior stages. This way, we end up checking less than what would be checked if we blindly iterated through all rules.

The ruleset used in this paper is from Snort [8] - a software-based IDS that uses hash-tables as a filtering stage.

2.3.1 Software vs Hardware Accelerator Implementations

Software-based IDS systems have certain limitations. When running Snort at high line rates (e.g. 100 Gbps), hundreds of CPU cores are required to meet the throughput demand [6]. To combat this, accelerator-based implementations of IDS's can help us regain CPU resources. In particular, the Pigasus IDS [6] uses a cascading system of bit-wise filters to bring the CPU usage down to only a handful of cores in most cases.

In particular, Pigasus claims to give only 5% of packets to a software-based IDS and narrows the ruleset down to (number) of rules to check. It uses the multi-filtering approach from before but can do so more efficiently by replicating smaller units of hardware instead of using multiple CPU cores to achieve parallelism.

2.3.2 Filtering effectiveness

Although using hash tables and other light-weight checks as filters to later stages helps with meeting higher throughput, it is still sensitive to payload data and network traffic. In figure 2.1, different stratosphere [9] traces tested against an ideal exact-match filter with no false positives. Even for some of the "normal" traces that contain no malicious packets, 7% of the packets still require additional processing. Even if Pigasus or Snort were able to obtain this perfect filtering for 100 Gbps, 7 Gbps would still need additional processing, highlighting a need for regular expression acceleration.

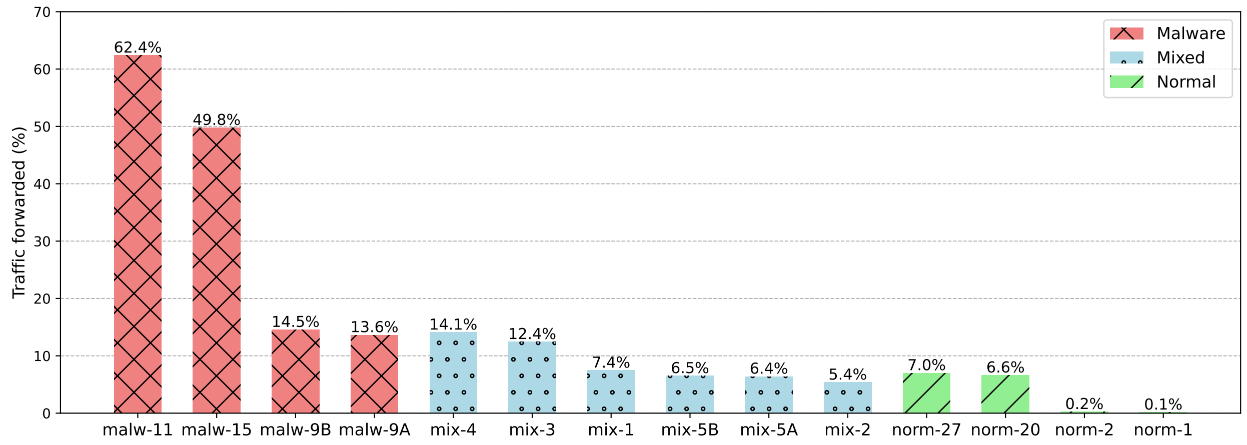


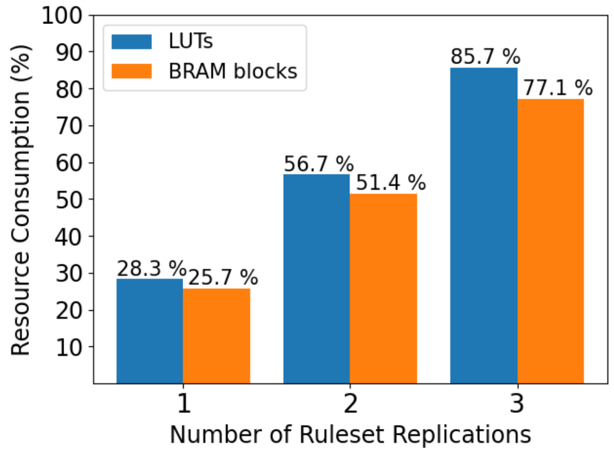
Figure 2.1. Traffic forwarded for regular expression matching assuming perfect exact-match filtering with no false positives

2.4 Regular Expression Processing on Hardware

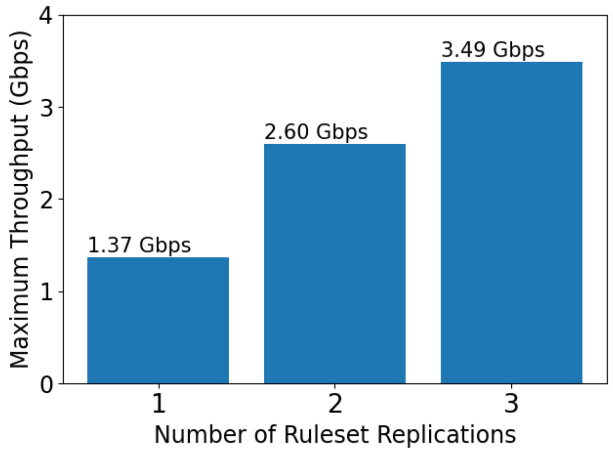
Hardware-based implementations of regular expressions can either use DFAs with one state active at a given time or NFAs that can have multiple states active. DFAs are implemented as state-transition tables, are easy to reconfigure, and hence easily applicable to ASIC implementations. However, these designs have more states and thus a larger memory footprint [7]. Further, they do not allow for as much parallelism. While NFAs are more compact and allow for parallelism, they must be re-synthesized as rulesets change with subsequent updates. NFAs are a good fit for FPGAs due to the hardware being reconfigurable [3, 10].

HARE [7] is an ASIC DFA-based design which takes a set of regular expressions and combines like prefixes to make a single large regular expression. HARE then creates state-transition tables and splits the regular expression into components. Since state-transition tables increase the number of elements to multiplex with each new state, multiplexing logic would become too slow to keep up with HARE's proposed clock rate for an entire IDS ruleset.

In practice, HARE is deployed on the order of 1-16 regular expressions, while IDS rulesets involve several thousands of regular expressions.



(a) Resource consumption of an NFA pipeline.



(b) Estimated throughput of NFA pipeline.

Figure 2.2. Throughput and resource usage of a Grapefruit-style NFA pipeline on an Alveo U200 FPGA for a 11K-rule Snort ruleset, as the degree of pipeline replication increases. The figure shows that there is only sufficient resource to support 3 replicas, which limits throughput to 3.5 Gbps.

For regular expression processing on FPGAs, solutions such as GRAPEFRUIT [3] and REAPR [10] exist, allowing for larger numbers of regular expressions to be processed due to more resource efficient NFA-based state machines. GRAPEFRUIT in particular, uses BRAM as a character-indexed lookup table for individual states in each regular expression, and uses logic cells to create the topology for each. Unfortunately, these FPGA designs do not offer nearly enough throughput to keep up with a 100 Gbps linerate. In figure 2.2, we create a simple pipeline of GRAPEFRUIT-style NFAs. If we seek to parallelize simply by duplicating this design, we quickly run out of resources (figure 2.2a). Using the synthesized clock rate and number of replications, the maximum throughput we can achieve with this method is shown in figure 2.2b at 3.5 Gbps, far below the linerate requirement.

Even from GRAPEFRUITs proposed figures, the best BRAM-based design will achieve 6.21 Gbps for a TCP ruleset. To achieve 100 Gbps, we would need to replicate GRAPEFRUIT 17 times. One replication of GRAPEFRUIT takes 73k LUTs, 106 FFs, and 581

BRAM blocks. Multiplying this 17 times goes well over the resource limits for virtually any FPGA.

3. DESIGN

3.1 Overview and Design Philosophy

For IDS designs similar to that of Pigasus, the output of exact-match filtering tends to be a much lower throughput than the input. Additionally, packets coming out of the exact-match filter typically require only 1 or 2 rules to be checked. Zhao et al. [6] proposes using designs such as GRAPEFRUIT [3] for regular expression processing on these filtered packets. However, GRAPEFRUIT alone still does not provide enough throughput for what Pigasus requires. A drawback from using GRAPEFRUIT in this application is that it will check every rule in the ruleset and not just the 1 or 2 rules that Pigasus tells it to check. It would be convenient if we could pick and choose which rules we would like to match within GRAPEFRUIT, and if two packets want to check different rules we can process those two rules in parallel. Even by processing just 7 packets in parallel, we would meet the "full matcher" requirement of Pigasus at 5 Gbps and completely remove the need to replicate GRAPEFRUIT.

There are still some things that GRAPEFRUIT does well that we would keep in such a multiplexed design. One such design feature is putting NFAs into "groups" that process in parallel on an input stream. While GRAPEFRUIT is pipelined, it doesn't pipeline all rules but instead pipelines groups of rules. The architecture of NFAs in GRAPEFRUIT prefer grouping, as they are able to share more BRAM this way and resource costs are minimized. Grouping also helps with multiplexing into rules as there would be fewer entries and therefore a potentially shorter critical path. These groups would also be convenient for grouping similar popular rules together and processing them in parallel on the same data (e.g. if multiple rules from a set of HTTP rules are popular, we can just group all HTTP rules together from this set).

In our design, we present a 2-stage structure; the first stage is an exact match stage similar to Pigasus's multi-string pattern matcher, designed to reduce throughput to subsequent stages and cut down the number of rules to check. The second stage is an NFA matching stage with NFAs similar to GRAPEFRUIT with the added bonus of multiplexing into NFAs using multiple channels of input, giving the illusion of replication without needing to replicate the entire ruleset.

The exact match stage (also known as stage 1) consists of multiple exact-match pipelines that operate around 400 MHz and are designed for high throughput and high filtering similar to Pigasus. A single pipeline will match whole packets in a byte-wise fashion using hash tables on all starting indices. At the end of these pipelines, a "Group ID store" or "GID store" will collect the NFA groups that need to be matched against later in the design.

The NFA matching stage (or "stage 2") is responsible for taking packets and their corresponding NFA groups from stage 1 to process them - checking all exact matches and regular expressions for the requested rules. Stage 2 is configured to maximize parallelism, and can skip over packets temporarily if there are too many packets currently processing on a requested rule. Throughput for particular groups can also be increased by statically replicating popular NFA groups at the time of synthesis. Stage 2 runs at 100 MHz and one group will process a packet at approximately 0.75 Gbps.

3.2 NFA Matching Stage

3.2.1 Overview

The exact match stage will hand any unfiltered packets and the respective rule groups over to the NFA Matching Stage (henceforth called stage 2). The goal of stage 2 is to extract the most amount of parallelism out of the incoming packets given the resources it has in its NFA table (the table containing NFAs that represent rules). Stage 2 operates on the notion of "channels". A channel is a state machine that reads packet data from it's corresponding buffer and feeds it to an NFA group. The number of channels corresponds to the maximum number of packets that can be processed in parallel. The channels will search their buffers for a packet that requests a free group that is not in use by any other channel. This leads to out-of-order execution with the goal of improving average throughput.

We can follow the path of execution for one 512-byte PDU (packet data unit) using figure 3.1. The PDU first arrives at an AXI interface for stage 2 where it is assigned a unique "data ID". The PDU then gets sent to one of several stage 2 interfaces in a round-robin fashion.

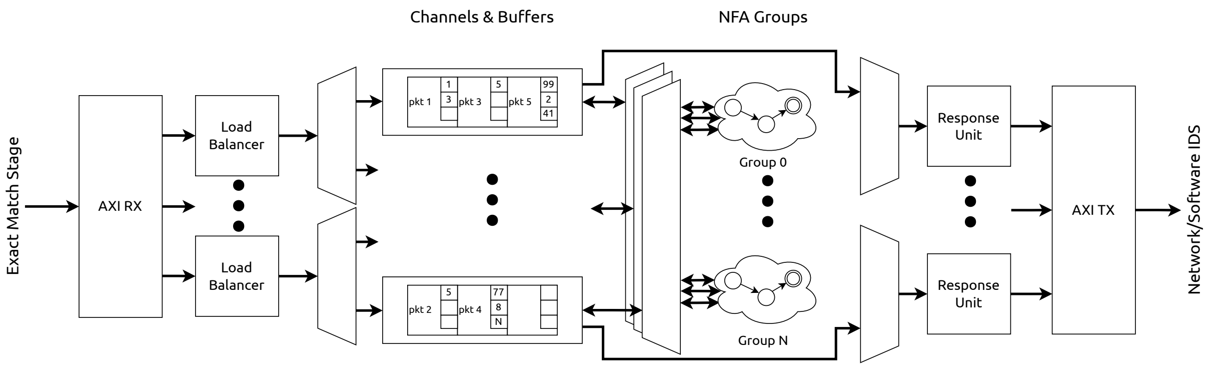
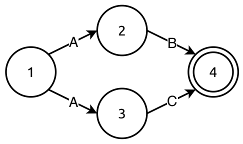


Figure 3.1. The NFA Matching Stage (stage 2)

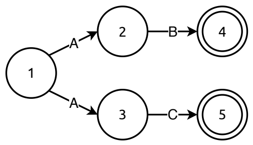
These interfaces each connect to a load balancer that is in charge of writing data to one of several buffers based on buffer capacity. Once the PDU is written to a buffer along with it's requested groups, it then waits for the channel to inspect it. Once the group that the PDU requests is available, the channel will select the PDU and start reading it's data to an NFA group. Once all groups are checked (or one group matches), the channel writes the result of the match as a boolean flag to the buffer. It also writes a flag to the PDU's buffer entry indicating that the PDU is done processing and is ready to be given back to the network via the response unit. Once the response unit reads the PDU to the AXI TX module, it is then sent back to the network where the trailer of the packet is modified to indicate the status of processing the PDU.

3.2.2 Implementing NFAs

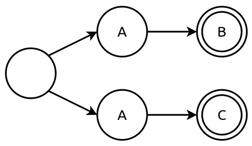
In our design, all regular expressions and exact-match strings for rules in stage 2 are represented as homogeneous NFAs. Homogeneous NFAs are NFAs in which each incoming transition for each state matches on the same set of characters. That is, if state S_i has incoming transitions $T_{\mathrm{i,j}}$ with corresponding characters $c(T_{\mathrm{i,j}})$, then the following property holds that $c(T_{\mathrm{i,j}}) = c(T_{\mathrm{i},k}) \forall \mathrm{j}, k \in transitions_\mathrm{i}$.



(a) Non-Homogeneous NFA



(b) Homogeneous NFA



(c) Homogeneous NFA reorganized

Figure 3.2. NFA representing the regular expression (AB)+(AC) being converted to a homogeneous NFA

Figure 3.2a, shows a non-homogeneous NFA for the regular expression "(AB)+(AC)" where state 4 breaks the homogenous property by having incoming transitions with different character sets. Figure 3.2b, splits state 4 into 4 and 5, creating a homogeneous NFA. Figure 3.2c, shows a common representation of homogeneous NFAs where the match characters for incoming transitions are closely tied to the state itself.

To construct this homogeneous NFA in hardware, we borrow much of the architecture from GRAPEFRUIT [3]. In figure 3.3, we implement the same regular expression of "(AB)+(AC)" in this fashion. The BRAM table is addressed by the current input character in the byte-stream. The output of BRAM is a bitvector where each bit position corresponds to a boolean value of whether the given character will match on an incoming transition for the corresponding state. States 1 and 2 share the same match function of "A", so we can share a column of BRAM between the two states, allowing for a more compact implementation.

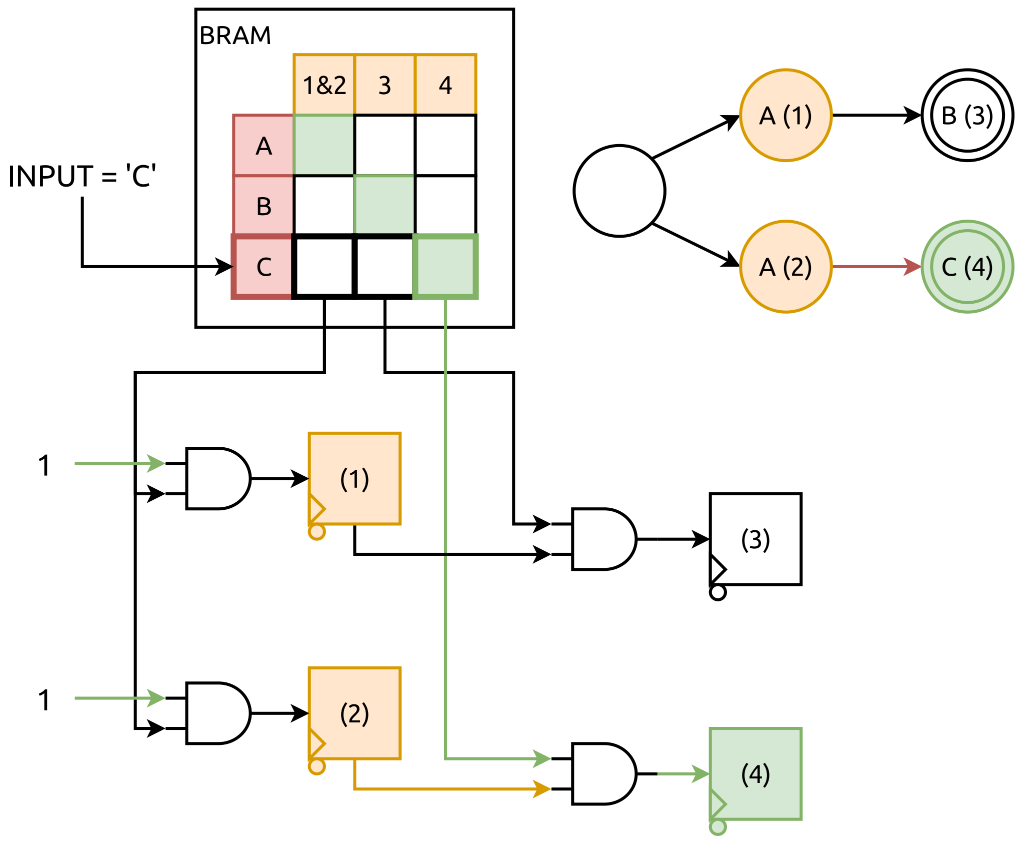


Figure 3.3. GRAPEFRUIT NFA for the regular expression (AB)+(AC)

Figure 3.3 shows a moment in time where states 1 and 2 are currently active and the NFA is receiving an input symbol "C". On the column for state 4, the character C yields a 1. This bit gets ANDed with the register for state 2 because state 2 has a transition into state 4. The result of this AND will activate state 4 on the next cycle, matching the regular expression.

The next step in the process is to organize these NFAs into a group of NFAs that occupy the same BRAM blocks. Our Vivado BRAM IP supports an address width of 9 bits and an output vector of 36 bits. If we wish to process multiple NFAs in parallel, we can follow

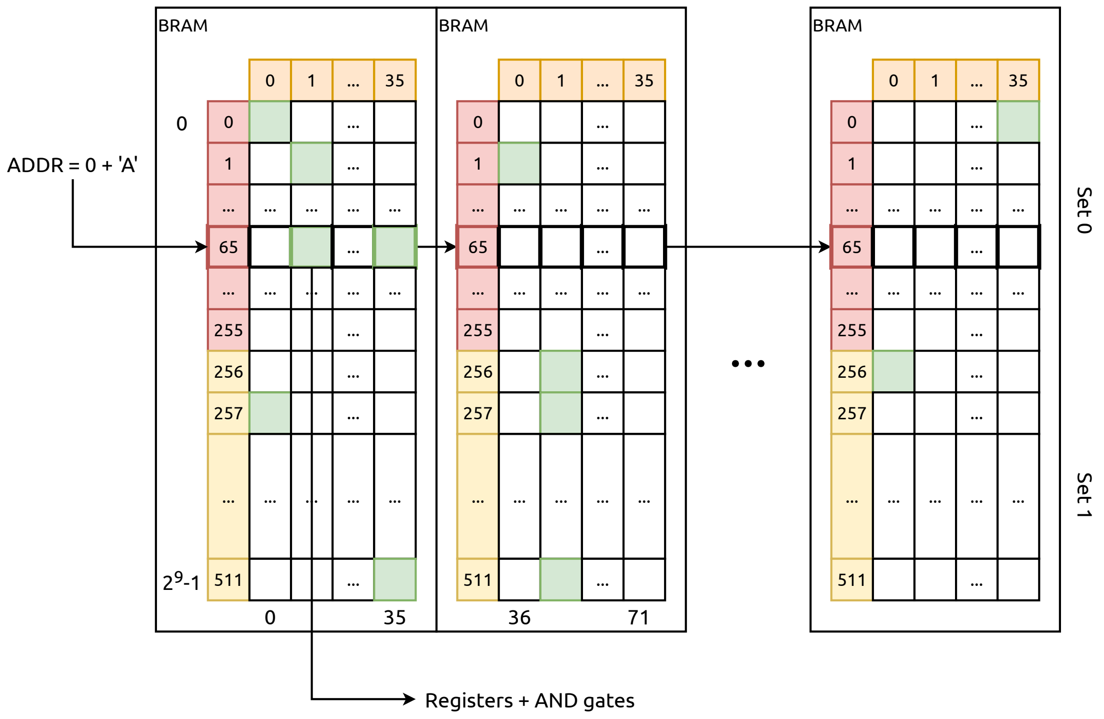


Figure 3.4. NFA group showing how to add multiple BRAM blocks horizontally and how to populate both halves of each BRAM block to avoid wastage

the architecture of figure 3.4 by using more bits in the output vector and sharing columns between states with identical match characters.

Once we run out of space and we need more than 36 bits at each address, we can simply had more BRAM blocks horizontally, where each new BRAM block takes the same address of the input symbol. We can now increase the number of states supported linearly in increments of 36 bitlines per BRAM block.

We can continue to add NFAs and states to this architecture forever. However, we are always wasting half of our BRAM. Vivado's BRAM IP has a 9-bit address field in its smallest configuration and ASCII characters are only 8-bits in length, meaning there are 2^8 possible input combinations but 2^9 addresses to access in each BRAM block. If we hard-wire the most-significant bit of the address to 0, we end up not utilizing the upper half of all BRAM blocks. To get more use out of our BRAM, we divide NFA groups into "NFA sets", where each set occupies one of these halves of BRAM. The most significant bit in the address field is the "set bit" that can be toggled to target different NFAs. For ease of implementation, we allocate the same number of rules to each set in each group (16), but this can be changed or optimized to get even tighter utilization. This structure of 32 rules in 16 rules per set, is referred to as an "NFA group".

3.2.3 AXI Interface

The AXI interface modules shown in figure 3.1 connect stage 2 to Xilinx's AXI bus protocol to communicate with stage 1 and the network. However, these interface blocks are responsible for more than just interfaces. They also handle packet re-ordering and bypassing of packets into software.

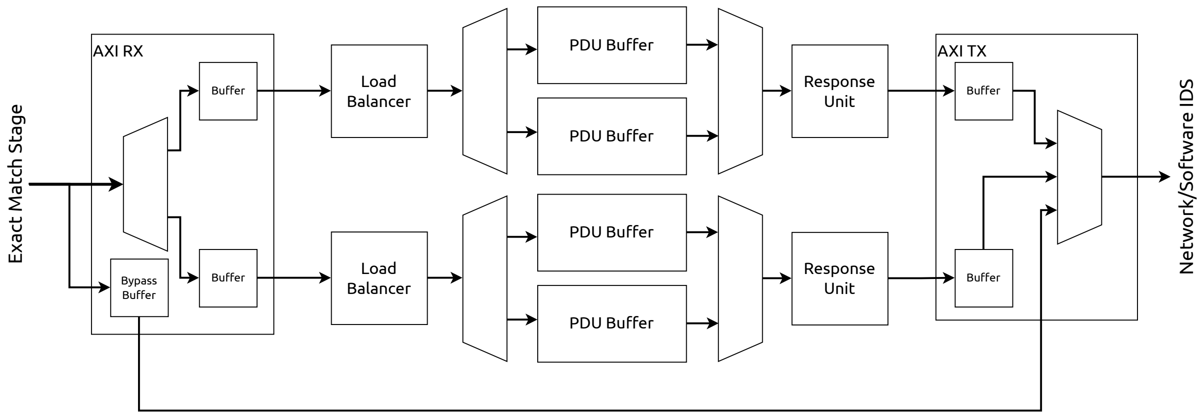


Figure 3.5. Stage 2 with the AXI modules shown in more detail. In this configuration, there are 2 load balancer - response unit pairs each in charge of 2 PDU buffers. NFA groups not shown.

AXI RX Interface

The AXI RX interface module receives packet data as well as group ID's on the AXI protocol's data lines. It temporarily stores the latest incoming packet in a "bypass buffer" in case the packet needs to be sent to software or if the packet is a control packet for the

NIC [11]. Upon filling the bypass buffer, it also fills one of several buffers corresponding to interfaces for the primary components of stage 2. These interfaces are connected to load balancers as discussed in §3.2.4.

The AXI RX module will fill these buffers in a mostly round-robin fashion. However, if all interfaces indicate that they cannot take any more packets, the packet will by passed to the AXI TX interface to be sent to a software IDS.

AXI TX Interface

Similar to the AXI RX interface module, the AXI TX module has several buffers that it rotates between. The number of buffers in the AXI TX module corresponds to the number of response units in stage 2. In figure 3.5, the AXI TX module can multiplex between any response unit as well as the bypass buffer coming in from AXI RX.

3.2.4 Load Balancer

When a PDU arrives in stage 2 via the AXI RX module, it then gets sent to a module called a "load balancer" (seen in the second column of figure 3.1). The load balancer is responsible for writing data and group ID's to a small collection of PDU buffers, where the packet data will wait to be processed. The load balancer peeks at the capacity of each buffer and writes data to the buffer with the highest current capacity. Once written, the load balancer will trigger the entry holding the PDU to become "valid" so that the channel is aware of packet data occupying the now full entry.

If all buffers are full, a signal is sent out to the RX module telling it to not send any further packets to the load balancer. In this case, the RX module will either select another load balancer or forward the packet to be processed in software.

While the load balancer is writing to the buffers, it communicates with the response unit the buffer number and entry number it is writing to. The response unit will eventually use this information to re-order packets.

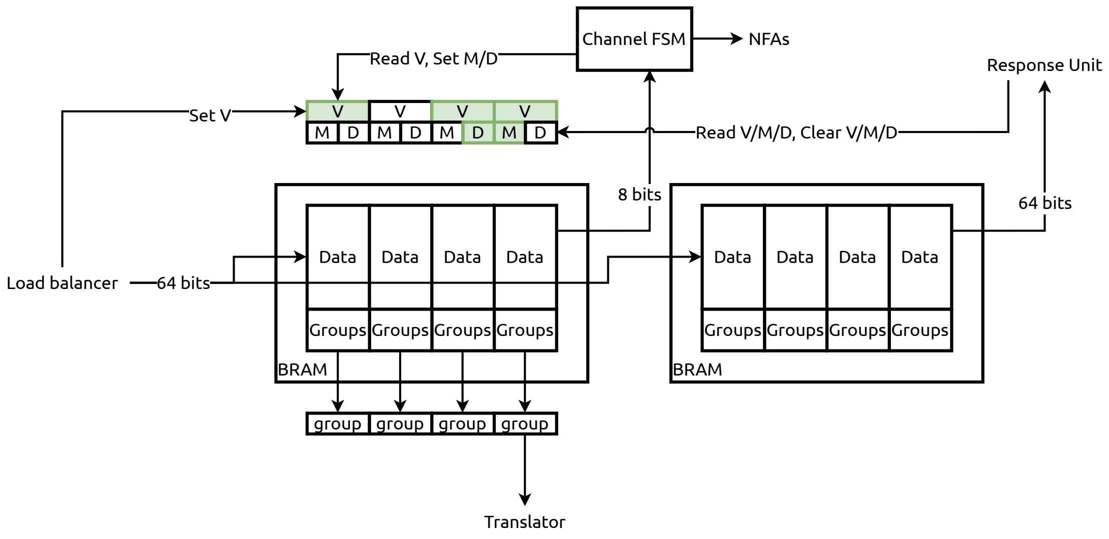


Figure 3.6. PDU buffer with corresponding interfaces and channel state machine

3.2.5 PDU Buffers and Channels

Each buffer in stage 2 corresponds to a channel - a collection of hardware objects such as state-machines that takes stored data and streams it into an NFA group. These buffers store the data used for matching and eventually de-allocate this data or send it out on the stage 2 egress path.

PDU buffers have three ports, each requiring their own independent address bus; a port for the load balancer to write to an entry, a port for the channel to read data character-by-character, and a port for the egress or "response unit" to read out data after it has been matched against an NFA. Due to Xilinx's BRAM IP only having at most two independent address busses, stage 2 buffers incorporate duplicate BRAM arrays inside each buffer to achieve this as seen in figure 3.6. The load balancer will write to both BRAM blocks simultaneously. However, the channel will read characters (8 bits at a time) from block 1 and the egress port will read 64 bits at a time from block 2 independently. This way, we effectively get a BRAM block with 3 address busses by allocating a block for each read port.

Each slot in the buffer has 3 status bits; valid, match, and done. Valid gets set high when the BRAM is written to by the load balancer on an empty entry. When the channel sees that the valid bit is high but the done bit isn't, it is responsible for matching that entry against an NFA group. Once the channel is done processing the packet, it sets the "done" bit high and sets the "match" bit high whenever it encounters a match. The response unit (egress) will look for entries that are both valid and done. These entries will get sent out of stage 2, after which all status bits get cleared and the slot is empty again.

The channel FSM will look for a buffer entry in the SEARCH state. When it finds a packet taking up a slot in the buffer, it enters the CHECK state where the channel checks to see if the corresponding NFA group is available and not being used by any other channel. If the group isn't available it goes back to the search state. If the group can be acquired, the channel moves to the PROC state and reads characters from the selected entry. It then moves to the DONE state, where the channel flips the "match" bit for the entry if the NFA group matched against the packet. The channel may also set the "done" bit high if it is done matching the packet, signaling the egress to take a look at the result. If there are more groups to check, the buffer will take a few cycles to fetch the next group ID from BRAM and inform the channel that there are more groups to analyze.

3.2.6 Response Unit

In stage 2, PDUs have the ability to finish processing in a different order to how they arrive in stage 2. If these PDUs are allowed to be forwarded to the network out-of-order, this can put additional pressure on the TCP protocol to re-order these packets [12]. Even though packets may complete matching out-of-order, they should be sent out in the order in which they arrived.

Part of this responsibility lies with the response unit, which keeps a queue of packet locations within a set of packet buffers. This queue tells the response unit which PDU should be allowed next to exit stage 2. It continually looks at the next entry until the "done" bit is set high and then reads out the data in 64 bit chunks.

3.2.7 Packet Re-ordering

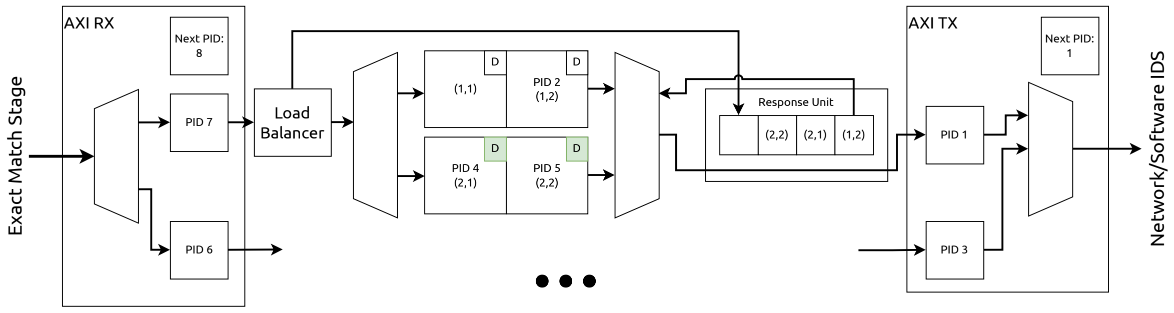


Figure 3.7. Packet re-ordering is handled on a per-location basis in the response unit and on a per-data ID (PID) basis in the AXI TX module

Although stage 2 may process packets in a different order than how they arrive, it always commits them in the same order to not add any more work for TCP drivers. It does this through the use of "data ID's" (or PID's). A data ID is a unique number assigned to each packet as it enters stage 2 and is how stage 2 generally keeps track of what to send to the network next.

As packets come into stage 2, the AXI RX module will assign packets to data ID's in a sequential order, only incrementing the next data ID after a successful store to a PDU buffer. This ensures that every PDU in stage 2 can be placed in a sequential order with no gaps in the data ID's (e.g. if PID 2 and PID 5 exist in stage 2, PID's 3 and 4 must also exist in stage 2).

After the PDU with its data ID is successfully handed to a load balancer, the load balancer will place the PDU in an empty buffer entry and inform the corresponding response unit where the PDU was placed. The response unit uses this (channel,entry) pair to keep track of which entry to read from the buffers next so that the PDU's are read out in order. In figure 3.7, we can observe that PID's 2,4 and 5 can be reached by the response unit, which keeps a queue of the location of these packets. In the same figure, the response unit shown is waiting for the PDU at (1,2) to finish (as indicated by the "done" bit).

Once the response unit sees that the next PDU has finished, it forwards this PDU to it's buffer in the AXI TX module with the corresponding PID. Since each PID in stage 2 exists in sequential order, the AXI TX module doesn't keep a queue but rather a counter indicating which PID needs to be read next. Since each response unit forwards PDU's in the order which the load balancer receives them, the next PID will always be given by one of the response units. Once the AXI TX module has been given the next PID, it forwards the PDU to the network and then increments it's PID counter to search for the next packet.

3.2.8 NFA Replication

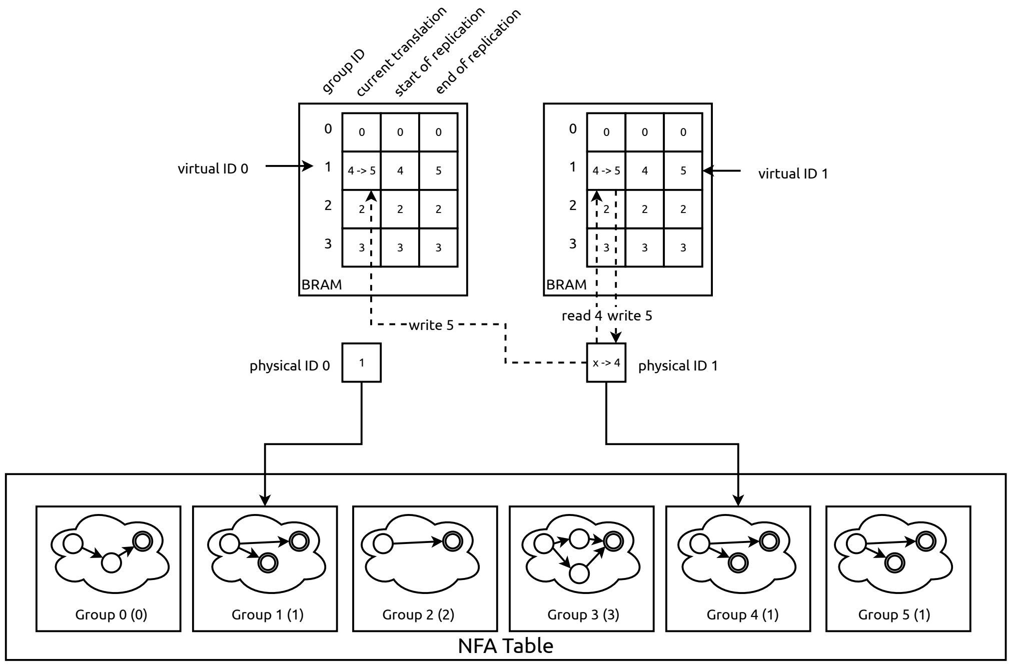


Figure 3.8. Group ID translation and NFA group replication. This diagram shows channel 0 currently has group 1 while channel 1 also requests group ID 1 but gets access to group 4.

Stage 2 offers the ability to replicate certain NFA groups at the time of synthesis. In the NFA table, this is handled simply by placing these replicated groups at new indices in the table as seen with group 1 in figure 3.8.

In between the channel and the NFA table sits the "translator block" which takes the original GID requested and transforms it to select one of several replicas. The channel's requested GID is called the "virtual GID". In the table shown in figure 3.8, this virtual ID is used as an index to obtain the current corresponding "physical ID" pertaining to one of several replications of the same NFA group. When a channel wants to request a new group, it accesses the translation table and writes the "current translation" to a register that is seen by the NFA table. The translator then looks at the "start of replication" and "end of replication" values to determine where the next translation will be located. It then writes the new translation for the GID back to each channel's BRAM so that the next channel to request the same GID gets a new physical GID.

Channels can still clash on the same NFA groups and there are not enough resources to replicate all groups by the number of channels. That's why the NFA table itself keeps track of which channel has access to which group and ignores requests from channels that do not have access.

4. RESULTS

4.1 FPGA Evaluation and Python Simulation

To test the validity of stage 2 beyond simulations, stage 2 is placed within Corundum [11] - an FPGA-based NIC. This design is then placed on the Alveo U200 FPGA board to verify correctness and theorized performance numbers. This verification is also used to validate results obtained by stage 2's corresponding Python simulation which allows for parameters to be easily modified and results to be quickly gathered without needing to re-synthesize the whole design.

Stage 2's Python simulation is behaviorally accurate to the RTL implementation. The Python simulation emulates multiple PDU buffers all operating on an NFA table, keeping track of which channel has access to which NFA group. If any packets cannot fit into the simulator, they are written to the output trace indicating the need for the software-based IDS. The Python simulation ignores overhead cycles due to collisions on the same NFA group, cycles for accessing the NFA table, and cycles for reading/writing to the PDU buffers from the AXI interface. The Python simulator also does not perform packet reordering as discussed in §3.2.7.

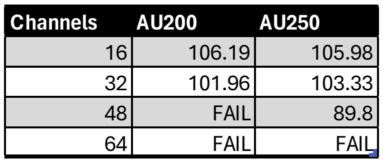
4.2 Theoretical Limits

In it's default configuration, stage 2 is designed to run at 100 MHz. With each NFA group being able to process a single 8-bit character per cycle, each channel operates at a throughput of around 0.75 Gbps. If the assumption is made temporarily that each packet only requests one NFA group, then 0.75 Gbps is the minimal performance of stage 2.

This worst-case performance of 0.75 Gbps happens when all packets in stage 2 request to match against one group. Since each group is atomic and can be ran independently, only one channel can access a group at any given time resulting in the serial execution of all packets. This minimum throughput can be avoided through the replication of popular NFA groups as discussed in §3.2.8.

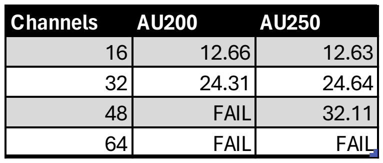
In contrast, the maximum theoretical performance for stage 2 happens when all channels are able to process packets simultaneously. Each channel must be accessing a different group, making the maximum throughput $0.75N$ Gbps where N is the number of channels. In practice, the maximum throughput will be somewhat less than this to account for the time it takes to select a new packet and start processing it, in addition to other miscellaneous latching overhead.

4.3 Resource Consumption



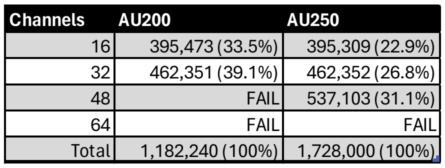
Channels	AU200	AU250
16	106.19	105.98
32	101.96	103.33
48	FAIL	89.8
64	FAIL	FAIL

(a) Max frequency (MHz) for a differing number of channels



Channels	AU200	AU250
16	12.66	12.63
32	24.31	24.64
48	FAIL	32.11
64	FAIL	FAIL

(b) Maximum estimated throughput (Gbps) for a differing number of channels



Channels	AU200	AU250
16	395,473 (33.5%)	395,309 (22.9%)
32	462,351 (39.1%)	462,352 (26.8%)
48	FAIL	537,103 (31.1%)
64	FAIL	FAIL
Total	1,182,240 (100%)	1,728,000 (100%)

(c) LUT count for a differing number of channels

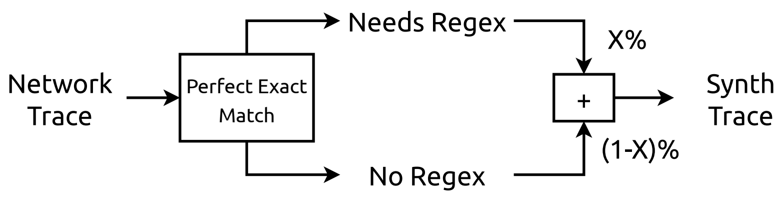
Figure 4.1. Metrics comparing the Alveo U200 FPGA to the larger Alveo U250 FPGA when synthesizing stage 2, sweeping the number of channels, the Alveo U200 fails to synthesize after 32 channels and the Alveo U250 fails after 48

Figure 4.1 presents synthesis results from the Alveo U200 FPGA platform (our default platform) and the larger Alveo U250 FPGA platform. We vary the number of channels between the two platforms until the design is too large to fit on the board. Each channel contains a buffer that contains 4 entries for packets.

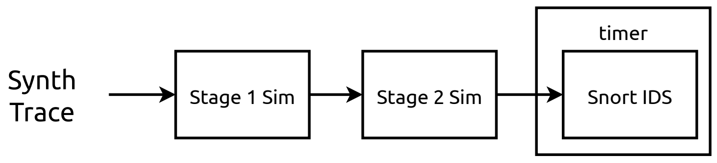
In figure 4.1a, we notice that the maximum frequency stays roughly the same for both boards with the exception of 48 channels on the AU250 platform. Figure 4.1b shows our estimated maximum throughput calculated using the number of channels and the maximum frequency. Changing the number of channels from 32 to 48 shows that with larger FPGAs, TRex can scale to meet larger throughput demands.

Additionally, figure 4.1c shows the LUT usage across both boards. LUTs for synthesizable versions of stage 2 typically use less than half of the FPGA's LUTs. Any more than this and stage 2 creates too much routing congestion to synthesize.

4.4 Simulation-Based Throughput Experiments



(a) Process for creating synthetic traces where X% of packets need to be checked against regular expression matching.



(b) Gathering throughput results for a synthetic trace using a timer on the Snort IDS

Figure 4.2. Python-based evaluation platform for gathering throughput results on TRex

Using stage 2's Python simulator, we are able to get an estimate of throughput in various circumstances. In figure 4.2, we take stratosphere [9] traces and split the packets into two groups depending on whether the packets would require the last bit of regular expression processing. Looking at the result in figure 4.3; even when 20% of packets require regular

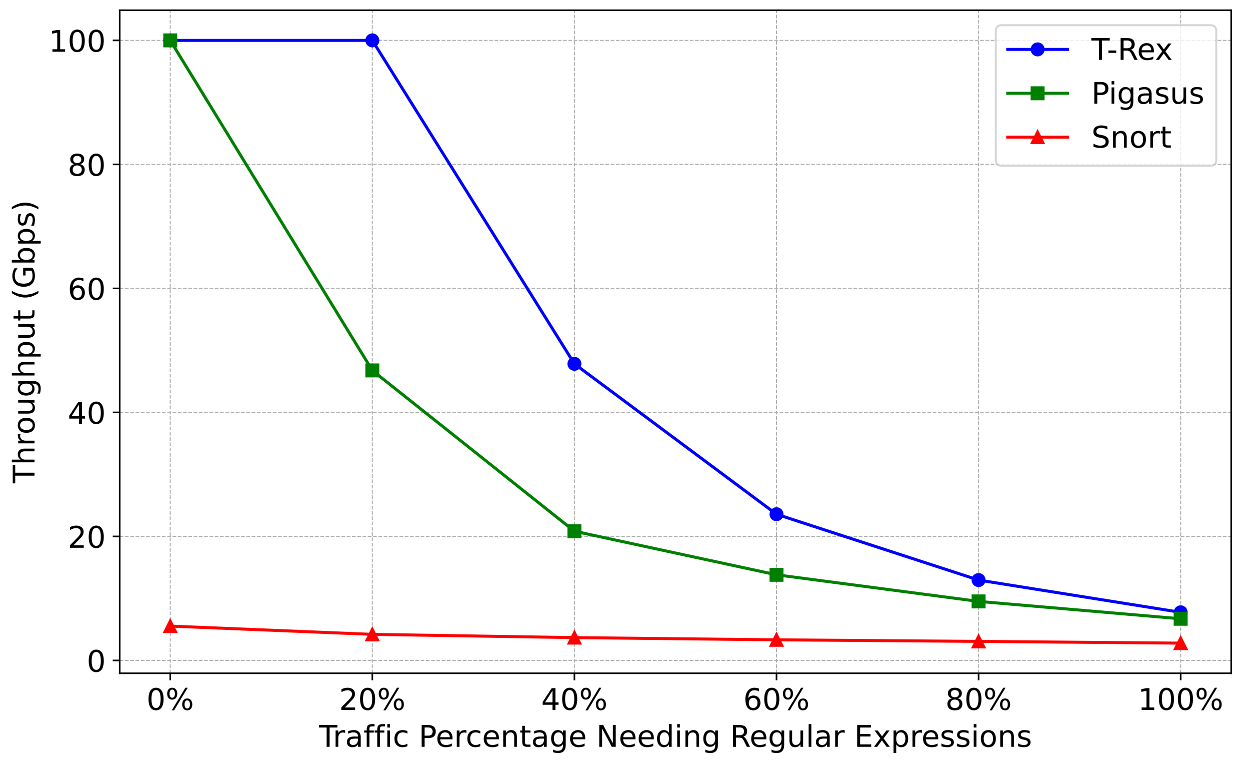


Figure 4.3. 10-core throughput of different IDS's while varying the percentage of packets requiring regular expression matching (out-of-order response unit)

expression processing, Pigasus' throughput is cut in half compared to 0%. The TRex model is still performing at 100 Gbps when 20% of packets require regular expression matching.

From 4.3, we can see that the curve for TRex is similar to the Pigasus curve shifted to the right by 20%, showing that stage 2 of TRex can handle roughly 20% of all network traffic before needing software support. In theory, TRex should only perform worse than Pigasus if Pigasus outperforms TRex's filtering stage enough to cancel out any benefits of stage 2.

If we scale the original stratosphere [9] traces to 100 Gbps as in figure 4.4, we observe that TRex doesn't always offer a significant improvement in throughput. Part of this phenomenon is due to packets with a high "load" where many groups are requested and single packets take a long time to process. Current work is being done to discover strategies of grouping and replicating rules in order to widen the gap with Pigasus [6].

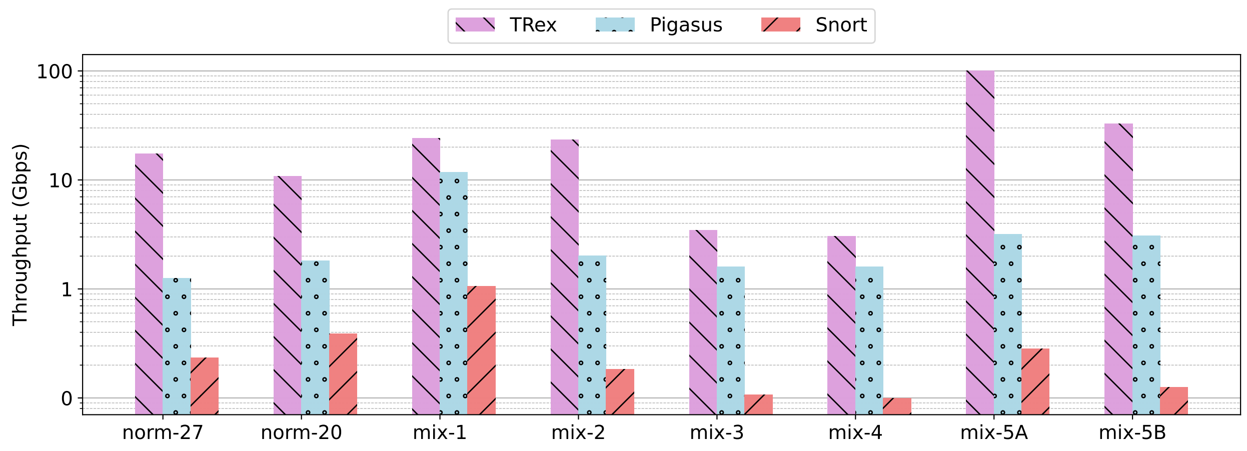


Figure 4.4. single-core throughput of different IDS's on scaled stratosphere [9] traces (out-of-order response unit)

5. FUTURE WORK

5.1 FPGA Testbench

Ideally, we would like to use our FPGA implementation of stage 2 to test throughput. However, we are currently running into packet loss when the throughput in stage 2 exceeds 1 Gbps. Current efforts are aimed at finding the source of this packet drop whether it be inside stage 2 or on the path to stage 2. Once this issue gets resolved, we can obtain more accurate throughput results.

5.2 Grouping Strategies

In the default configuration for TRex, rules are taken from the given Snort ruleset 32 rules at a time and put into groups naively. A lot of adjacent rules are similar and will come from the same file so this approach is still somewhat performative. While the initial reason for putting rules into groups was to decrease multiplexing and resource utilization, it’s still worthwhile to see if the way rules are grouped reduces the "load" of stage 2 where load is the product of the groups requested per packet and the incoming throughput. We are currently looking into multiple grouping strategies to decrease load.

While debugging and looking at individual packets, we notice that there are certain popular rules that attract a lot of packets after the filtering stage. Some of these rules appear together frequently. Therefore, the ideal grouping strategy would be to group popular rules together that appear together frequently, decreasing the average number of groups requested by each packet. With this approach, we would need to be careful to not group popular rules that do not appear together as this would lead to more conflicts for the same group unnecessarily. Once the popular groups have been formed, they can be selectively replicated before synthesizing TRex.

Analyzing popular rules has a caveat; measuring "popular" rules is a per-packet trace metric and can be different between traffic patterns. This would be effective only so far as there are "globally" popular rules that are popular across many different and diverse traces. It may be more ideal to group together similar rules - rules that have like strings and fields between them. For example, a rule that contains the string "dog" and another rule that contains "doggy" would be grouped together in this scheme.

5.3 Runtime-Configurable State Machines

Although TRex stage 2 can be configured to contain multiple replicas of certain groups, this does not happen during runtime and must be accomplished by determining the best configuration. The configuration must then be synthesized before it is flashed to the FPGA.

If TRex encounters a large burst of malicious traffic, it may overflow on an NFA group that does not have sufficient resources in terms of the number of replications. For this scenario, it would be useful to have some module that can reprogram itself at runtime to simulate the behavior any NFA in the design.

With the current architecture for NFAs, this cannot be done because the topology for the different states is in the flip-flops and look-up tables for the FPGA which are not re-programmable at runtime. As noted from comparing HARE [7] and GRAPEFRUIT [3], we have to use an NFA-based approach to fit all rules on the FPGA. However, we may use HARE's DFA-based approach to create a small collection generic modules that do not require re-synthesizing to reconfigure.

These modules are outlined in figure 5.1. The idea is to store the state-transition tables for all rules in DRAM. If a rule suddenly receives a large amount of traffic, the corresponding table from DRAM can be written to a generic DFA module in BRAM to allocate more resources to specific groups or rules. These DFA modules can then start matching against more packets parallel in addition to what is already implemented as NFAs to improve the lower-bound performance of stage 2.

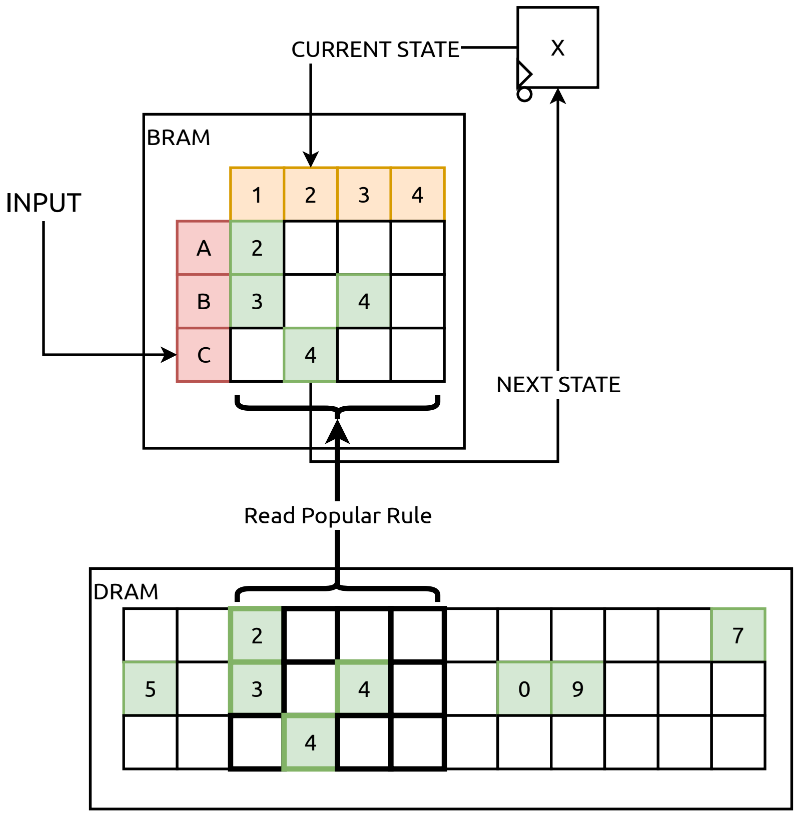


Figure 5.1. DFA based modules that can be reprogrammed at runtime for popular rules that were not accounted for at the time of synthesis

REFERENCES

[1] X. Wang *et al.*, "Hyperscan: A fast multi-pattern regex matcher for modern CPUs," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 631–648, ISBN: 978-1-931971-49-2. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang.

[2] R. Sidhu and V. Prasanna, "Fast regular expression matching using fpgas," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, 2001, pp. 227–238.

[3] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An open-source, full-stack, and customizable automata processing on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 138–147. DOI: 10.1109/FCCM48280.2020.00027.

[4] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, 2007, pp. 1064–1072. DOI: 10.1109/INFCOM.2007.128.

[5] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS '06, San Jose, California, USA: Association for Computing Machinery, 2006, pp. 93–102, ISBN: 1595935800. DOI: 10.1145/1185347.1185360. [Online]. Available: https://doi.org/10.1145/1185347.1185360.

[6] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 1083–1100, ISBN: 978-1-939133-19-9. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng.

[7] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[8] *Snort*. [Online]. Available: https://www.snort.org.

[9] *Stratosphere ips*. [Online]. Available: https://www.stratosphereips.org.

[10] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "Reapr: Reconfigurable engine for automata processing," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056759.

[11] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen, “Corundum: An open-source 100-gbps nic,” in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 38–46. DOI: 10.1109/FCCM48280.2020.00015.

[12] M. Laor and L. Gendel, “The effect of packet reordering in a backbone link on application throughput,” *IEEE Network*, vol. 16, no. 5, pp. 28–36, 2002. DOI: 10.1109/MNET.2002.1035115.