Spring 6-1-2021

Implementation and Optimization of PEG Parsers for Use on FPGAs

Shikhar Sinha
Shikhar.Sinha.21@Dartmouth.edu

IMPLEMENTATION AND OPTIMIZATION OF PEG PARSERS FOR USE ON FPGAS

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Bachelor of Arts

in

Computer Science

by

Shikhar Sinha

Advised by Sean W. Smith

DARTMOUTH COLLEGE

Hanover, New Hampshire

June 1, 2021

Abstract

DARPA's Guaranteed Architecture for Physical Security (GAPS) project requires a device to provably enforce security policies. As part of a solution that GE and Dartmouth have proposed for the GAPS project, parsers for Parsing Expression Grammars (PEGs) are required to run on a Field-Programmable Gate Array (FPGA). There exist programs, like *Pegmatite*, which produce PEG parsers written in VHDL, but these parsers have not yet been run on FPGAs. They have been run in simulators where they have been tested for correctness, but they need to be adapted for execution on FPGAs (Lucas et al., 2021).

This thesis explores the process of modifying existing VHDL PEG parsers to run on FPGAs and optimizing their performance. We contribute two techniques to achieve performance improvements: (1) exploiting data parallelism, and (2) parsing the input packet as it arrives instead of waiting for the entire packet to be received. We were not able to execute our solution consistently on FPGAs, so we present an analysis of these techniques through simulations.

Acknowledgements

I would like to thank Professor Sean W. Smith for his support in advising not only this thesis but also the other projects I worked on in the Trust Lab over the past couple years. I would also like to thank Prashant Anantharaman for working with me throughout the year, helping me understand the new (to me) world of hardware design and the process of translating parsing algorithms into hardware.

Additionally, I thank Matthew Seidel for his development of the end-to-end FPGA test framework and guiding me through the process of development on FPGAs.

Thank you to all my family and friends for guiding and supporting me over the past four years, especially as I neared the completion of this project.

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0121. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

Contents

Chapter 1

Introduction

In critical network infrastructure like power grids, avionics, and in sensitive settings like healthcare and intelligence, security constraints require that only verified and permitted data traverses the network. To handle this situation, the Defense Advanced Research Projects Agency (DARPA) initiated the Guaranteed Architecture for Physical Security (GAPS) project (DARPA, 2019). GAPS aims to build a device to guarantee enforcement of security policies across multiple security domains. Additionally, critical infrastructure settings like the power grid and avionics also require low latency and high throughput.

1.1 Proposed Solution

General Electric Global Research (GE) has proposed a solution for this project called MIND (Monitoring and Inspection Device) (Pomerantz, 2019). Network traffic will flow through the MIND device, which will enforce the necessary security policies such that only those with appropriate credentials can see sensitive

data. The MIND device uses a Field-Programmable Gate Array (FPGA) to perform the necessary computation for enforcement of security policies.

This is where Dartmouth's Trust Lab comes in. We take a language-theoretic (LANGSEC) formal language approach to enforcing security policies (Falcon et al., 2016). In this approach, security rules are specified as a grammar that defines a language that meets security criteria. Enforcement of security rules entails parsing received and transmitted data and allowing only the packets that are accepted by the specified grammar. LANGSEC parsing and packet filtering as performed traditionally on a general-purpose CPU introduces undesirable latency (Lucas et al., 2021). We believe that executing parsers on FPGAs will reduce latency and increase throughput. We have chosen Parsing Expression Grammars (PEGs) as our formal language specification to define what packets should be allowed to pass through the MIND device. Compared to traditional formal language specifications, e.g., Context Free Grammars (CFGs), PEGs are more suitable for parsing network packets (Lucas et al., 2021). Lucas et al. (2021) have developed the *Pegmatite* tool to generate VHDL (VHSIC (Very High-Speed Integrated Circuit) Hardware Design Language) parsers for a given PEG specification.

Our task is to integrate *Pegmatite*-generated VHDL PEG parsers into an end-to-end framework on an FPGA to emulate the reception of a packet by a MIND device and its subsequent security verification. To help with this integration, GE has provided an end-to-end framework which simulates the GAPS environment. The framework allows us emulate sending and receiving packets on the FPGA and has hooks to invoke VHDL parsers to perform the necessary packet filtering, deciding whether or not a packet meets a given PEG specification.

Figure 1.1: Process for generating optimized parsers

1.2 Thesis Summary

This thesis explores the process of adapting *Pegmatite* parsers for use in the end-to-end framework and optimizing their performance. We introduce two techniques: (1) taking advantage of elements of the parsing process that can be performed independently in parallel, and (2) starting to parse the packet as it arrives instead of waiting to receive the entire packet. In addition, we look at the possibility of pipelining the parser to parse multiple packets at the same time, such that multiple packets are being handled by the parser in different stages of the parsing process.

The remainder of this chapter provides a background on LANGSEC, formal languages, in particular PEGs, FPGAs, and hardware acceleration techniques. Then, Chapter 2 describes the how we implement these optimizations. Chapter 3 describes the results. Chapter 4 concludes and examines avenues for future research.

1.3 Background

The following sections provide context on the key concepts and technologies that are used throughout the thesis. First, we discuss language-theoretic security, the basis of our approach to parsing. Then, we discuss Parsing Expression Gram-

mars, our chosen formal language specification. Lastly, we provide context on why FPGAs are well-suited for the GAPS project and previous use of FPGAs in parsing.

1.3.1 Language-Theoretic Security (LANGSEC)

When a program processes input without first verifying that the input is valid and meets the desired specifications, it opens itself up to reaching unpredictable states that can lead to confidential data being exposed (Bratus et al., 2011). In one such programming antipattern known as shotgun parsing, *if* statements are scattered throughout the processing code to check the inputs, without formal justification, hoping to prevent any problematic inputs from getting through (Falcon et al., 2016). However, this can put the program in unpredictable, potentially problematic states. The Heartbleed attack of OpenSSL's Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols, which help provide secure communications on a network, exploited unverified input to expose user data. In TLS/SSL, a *heartbeat* message is used to verify that another computer is still connected. In a heartbeat request, one computer sends a message with a length n, and encrypted data also of length n, and asks the other computer to send back that encrypted data. The problem is that the OpenSSL implementations do not verify that the encrypted data is as long as it is claimed to be. So, if the data is shorter than the claimed length, an attacker could gain access to user data stored in the allocated buffer beyond the length of the encrypted data (Fruhlinger, 2017). In this case, the set of checks was not sufficient, and this exposed user data. Ali et al. (2021) show how this problem is ongoing, pointing out recently discovered issues in the TCP/IP stack, iOS email, and Windows DNS, that all stem from failure to properly

validate inputs.

Each program implicitly has a set of expected, valid inputs. LANGSEC requires that this set be specified explicitly as a formal language, using a grammar. A formal language is a set of strings of characters from a finite alphabet. The grammar for a formal language provides a set of rules that can be used to form strings in the language. To combat exploits like the Heartbleed bug, it is necessary to construct a parser to recognize a program's input language. All untrusted, unverified input should be validated by this parser before being processed by the program. Ideally, the grammar should be as simple as possible, as categorized in the Chomsky hierarchy. If a grammar is too complex, it can be difficult to determine if it matches the implicit desired grammar of a program.

Parsing Expression Grammars (PEGs)

Parsing Expression Grammars (PEGs) are a class of formal languages that are well suited for binary packet specifications. PEGs, unlike more traditional Context Free Grammars (CFGs), do not have non-deterministic choice. This facilitates parsing of a PEG in linear time, a desirable property when trying to achieve high-throughput parsing (Ford, 2004). Section 2.1 details the mechanics of this parsing process.

1.3.2 Field-Programmable Gate Arrays (FPGAs)

As discussed earlier, parsing algorithms on general-purpose CPUs do not produce sufficient throughput and introduce additional latency, making them unsuitable for the GAPS project (Lucas et al., 2021). For this reason, a lower-level,

more targeted solution is desirable. Field-programmable gate arrays (FPGAs) have properties that make them a good choice for high-throughput, low-latency parsing. FPGAs have arrays of logic blocks that can be configured by a programmer to form various logic gates, like AND and XOR, or more complex functions. In addition, FPGAs contain memory in the form of flip-flops and RAM. Using a hardware description language (HDL), like VHDL or Verilog, a programmer can configure the logic blocks to form the desired circuit. An FPGA can take advantage of parallelization opportunities, and since the gates can be configured in a custom manner, the programmed circuit can perform a specific task more efficiently than a general-purpose CPU. In addition, they maintain advantages over custom hardware (Application-Specific Integrated Circuits) because they can be reconfigured relatively easily and often. ASICs, in contrast, have a relatively long design process and greater initial costs. Thus, a parser optimized to run on an FPGA could provide good performance and be reconfigurable to meet the changing and diverse needs of various security policies and domains. However, it is important that the parser be well-designed and optimized for an FPGA because while FPGAs have the advantages discussed above, they also have limited memory and slower clock rates than general-purpose CPUs. In our experiments, the FPGA we worked with used a clock frequency of 100 MHz while general-purpose CPUs have clock frequencies north of 1 GHz. So, a naive implementation of a parser on an FPGA could be slower than a CPU implementation.

GAPS-specific FPGA and the End-to-End Framework

The GAPS project target FPGA is a Xilinx Zynq UltraScale+ MPSoC (Multiprocessor System on a Chip) ZCU102. "MPSoC" indicates that this device has separate processing system (PS), a quad-core ARM Cortex A53 CPU, in addition to the programmable logic (PL) made up of logic blocks. In the end-to-end framework of our experiments, the PS is responsible for generating packets, sending them to the PL where our parsers reside, and reading responses from the PL to verify that the parsing and filtering has been performed correctly. The PS and PL communicate over an AXI-Stream interface. On the PL, processing has three key steps.

1. The packet is read in an stored from the CPU into RAM over the AXI interface, 1 byte at a time.

2. The packet is parsed and a decision is made about whether to pass the packet on, or reject it. As it is parsed, the packet is moved into a separate RAM module for step 3.

3. If the parser decides the packet is acceptable, the packet is written back out over an AXI interface 1 byte at a time to the A53 CPU.

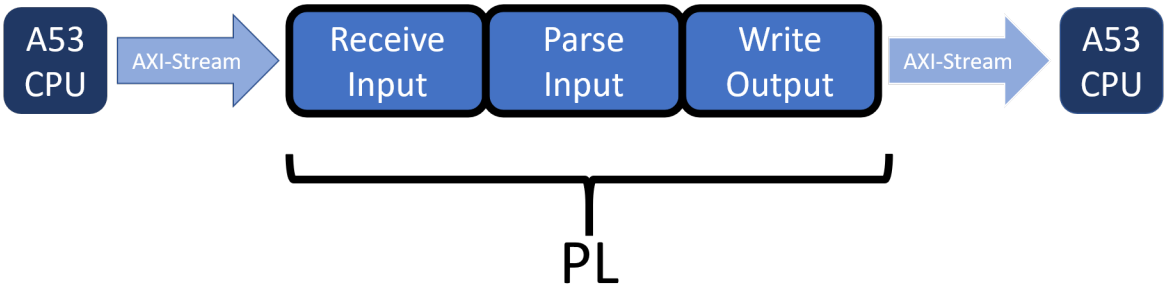

Figure 1.2: GAPS End-to-End Framework

Previous Work on Parsing on FPGAs

There have been previous efforts to introduce parsing on FPGAs using parsers designed in VHDL. Using a combination of a single FPGA and 5 CPU cores on a single server, Zhao et al. (2020) were able to perform 100 Gbps intrusion prevention on network traffic, looking at packet headers, and strings and regular expressions within the TCP bytestream and individual UDP packets. Using the FPGA as the primary processor and offloading to the CPU for more specific tasks, they exploit pipeline parallelism and data parallelism, acceleration techniques which will be discussed later, to achieve significant speed-ups while also reducing power consumption. For Natural Language Processing applications, Ciressan et al. (2001) used FPGAs to achieve a speed-up of over 240x over a pure software implementation. Also, Bordim et al. (2003) used an FPGA to achieve a 750x speed-up of Cocke-Kasami-Younger parsing of CFGs over a software implementation of the algorithm for general-purpose CPUs.

Hardware Acceleration Techniques

Two of the ways that implementations on FPGAs can achieve performance gains over an implementation of an algorithm for general-purpose CPUs are pipeline parallelism and data parallelism. Both of these techniques are discussed in this thesis.

Pipeline parallelism involves breaking a monolithic process into discrete steps and processing different pieces of data in each step. In a monolithic process, some of the circuitry would be sitting inactive for the entire process, but pipelining allows us to use more of the logic at once. It provides increased throughput, usually

at the expense of some latency.

On the other hand, data parallelism performs the same operation on different pieces of data. In this case, the operation on a given piece of data is independent of other data, so multiple pieces of data can be operated on at the same time.

Chapter 2

Parsing Methods and Implementation

This chapter describes PEGs in greater detail and explains the approach taken by *Pegmatite* PEG parsers. Then, we consider three different optimization techniques:

1. Data parallelism.
2. Reversing the order in which input packets are consumed by the parser.
3. Pipeline parallelism.

2.1 *Pegmatite* and PEG Parsing

As discussed in the previous chapter, Parsing Expression Grammars were introduced by Ford as an alternative formal language specification that better lent itself to parsing machine-oriented languages (Ford, 2004). PEGs remove the ambiguity that make CFGs suitable for natural languages and instead introduce deterministic, prioritized choice.

2.1 *Pegmatite* and PEG Parsing

Formally, a parsing expression grammar G is a 4-tuple (V, Σ, S, R) where V is a set of non-terminal symbols, Σ is the alphabet of terminals over which the language is defined, $S \in V$ is the starting non-terminal, and R is a set of rules. Each rule r is a pair (v, e), written $v \leftarrow e$, where $v \in V$ and e is a parsing expression. Parsing expressions are listed in Table 2.1. For a given non-terminal $v \in V$, there is only one corresponding e such that $v \leftarrow e \in R$.

Rule	**Symbol**	**Explanation**
Epsilon	ϵ	empty string, always accepts and consumes 0 characters
Fail	f	always rejects
Character	a	consumes a character if it is a, rejects if else
Any	.	accepts and consumes 1 character, rejects if no characters to consume
And	$\&e_1$	accepts if parsing expression e_1 accepts, consumes 0 characters
Not	$!e_1$	accepts if parsing expression e_1 rejects, consumes 0 characters
Prioritized Choice	$e_1/e_2/\ldots/e_n$	accepts and consumes same number of characters as parsing expression e_1 if e_1 accepts. Follows behavior of parsing expression e_2 if e_1 rejects. If e_2 rejects, follows behavior of e_3, continuing to e_n
Concatenation	$e_1 \circ e_2 \circ \cdots \circ e_n$	accepts if $e_1, \ldots, e_n$ accept, each starting from where the previous finished. Consumes all characters consumed by $e_1, \ldots, e_n$
Kleene Plus	$(e_1)^+$	accepts and consumes e_1 parsing expressions until e_1 fails. Rejects if zero e_1's are consumed

Table 2.1: PEG Expressions as described in Lucas et al. (2021)

We now illustrate two key concepts of PEGs:

1. The prioritized choice parsing expression.

2. Recognizing and consuming a subset of a string.

Consider the grammar under alphabet $\Sigma = \{a, b\}$ with only the following rule: $S \rightarrow a/ab$. Now consider the input string $x = ab$. In a more common CFG, the rule would most closely correspond to $S \rightarrow a|ab$. Both options would succeed, and since CFGs allow for non-deterministic choice, we could select the second choice and recognize the entirety of x. However, PEGs are deterministic, so we cannot just choose the rule that leads to the consumption of the entire string. Since $S \rightarrow a$ succeeds, we must use that, and we cannot use $S \rightarrow ab$ even though that would consume all of x. Even though we were unable to recognize the entirety of x, this does not mean that we reject x. Instead the rule consumes just one character, a. Clearly no more characters can be consumed, so we would say that the PEG accepts and consumes the 1-character prefix of x, a.

The above example illustrates an important distinction between PEGs and other common formal language constructs: where other language specifications will either successfully recognize or reject a string, a PEG can reject a string or accept and consume a prefix of a string. A PEG recognizes a string only if it consumes the whole string.

In addition, PEG constructs are greedy. Operators like Kleene star and Kleene plus, which are not represented in Table 2.1 but can be easily constructed from those parsing expressions, consume as many repetitions as possible, eliminating the non-determinism attached to these operators in other formal language specifications. This means that a^+a could never be matched in PEGs.

The standard parser for PEGs, introduced by Ford, is known as the packrat parser (Ford, 2004). Packrat uses memoization and hashtables to achieve a linear

runtime. However, these more complex data structures do not transfer well to hardware. The *Pegmatite* tool, which produces VHDL parsers for PEGs, is based on the scaffolding automaton proposed in Loff et al. (2020). *Pegmatite* takes as input PEGs specified in text files in Backus Normal Form notation. The scaffolding automaton is a labeled, directed, acyclic graph of bounded degree. Consider a PEG G with k non-terminals operating on string s with n characters. The scaffolding automaton can be thought of as a $(n + 1) \times k$ two-dimensional array A. In the array, $A[i, j]$ indicates the number of characters non-terminal j will consume while recognizing the last i characters of s.

2.2 PEG Parsing on FPGAs

First, we examine how the *Pegmatite* tool produces VHDL parsers. Then, we can go about making the necessary changes to execute parsers in the MIND end-to-end framework and optimize their performance. The end-to-end framework operates in a finite-state-machine (FSM) model, transitioning across several states in each of the three steps described in Section 1.3.2. So, the critical work would be in modifying both the framework and the way the parsers are generated to make the two compatible.

Briefly, the VHDL parsers generated by *Pegmatite* are structured as follows: As input, a parser takes in the input string as an array of bytes, the length in bytes of the input, and a clock signal. It outputs a 1-bit signal indicating success (1) or failure (0). Internally, it maintains a scaffold which stores previous results. The actual processing work is performed by the *evaluate**i**()* processes where each *i* corresponds to a row of the scaffold automaton. A sample *evaluate* function similar to

those produced by *Pegmatite* is listed in Figure 2.1 for the language a^+. Within each *evaluate* process, there are k functions that correspond to and evaluate the spaces in the scaffold corresponding to each of the k rules in the grammar. These functions evaluate a given $A[i, j]$ using a combination of already computed results stored in the scaffold, the input itself, and specific properties corresponding to the rule and function they implement. The scaffold is then filled out with each process being repeated as more and more dependent results are computed. Importantly, the scaffold operates on the text from right to left, so it requires the entire packet before it can begin parsing. Line 5 of Figure 2.1 shows how the *counter* variable, which is decremented with each clock tick, ensures that the scaffold is evaluated from bottom to top. Section 2.2.2 explores an optimization we make to start parsing a packet as soon as it comes in.

```
1  -- "01100001" = "a" in ASCII
2  evaluatei: process(eval)
3  begin
4    if eval = 'i' then
5      if counter <= length - 1 then
6        if scaffold(i)(0) = -2 then
7          scaffold(i)(0) <= compute_aplus(
8                              i, scaffold, 0, 1, length);
9        end if;
10       if scaffold(i)(1) = -2 then
11         scaffold(i)(1) <= compute_terminal(
12                             to_rtype(input, length),
13                             i, "01100001");
14       end if;
15         end if;
16   end if;
17 end process evaluatei;
```

Figure 2.1: Initial Sequential Evaluate Function

In order to incorporate the *Pegmatite* parsers into the framework, we make several changes to both the framework code and the structure of the parser. The first of which is to introduce the scaffold as a signal that exists within the framework. This will make it available to the *evaluate* processes. Then, we add the supporting code for data structures (mostly various types of arrays) that are used by the *evaluate* functions throughout the parsing process. In addition, we incorporate the same parsing functions that are called within the *evaluate* functions.

As discussed earlier, a PEG differs from more traditional formal language specifications in that instead of being able to either recognize or reject some input string s, it can accept a *substring* of s, However, given the GAPS project setting, we do not want to accept a substring of s as that would leave some potential input packet unverified if it were passed along, or it would truncate and modify the input, another undesirable outcome. Fortunately, the structure of the scaffold automaton provides some assistance here. The structure of the scaffold parser and the greedy parsing process make it such that when the scaffold is completely filled out, $A[0,0]$ contains the maximum number of characters that can be consumed while accepting a given input. Thus, we can choose to recognize and pass along a packet only if $A[0,0]$ is equal to length of the input packet.

2.2.1 Parallelization

We are now ready to discuss the key changes we made to the a^+ *Pegmatite* parser and the framework to take advantage of data parallelism. There are a couple of existing sources of parallelism in the *Pegmatite* VHDL parsers:

1. Eventually, all the $evaluate$ processes operate simultaneously, so there are

$n + 1$ processes occurring at the same time. Each of these processes executes once every time anything in the sensitivity list updates, essentially operating on an infinite loop. Each time it runs, a process tries to fill in the assigned cells, relying on dependencies eventually being filled in.

2. All the functions within each *evaluate* process operate simultaneously, so each *evaluate* function completes k cells at a time.

One important observation about the previously generated VHDL parsers is that the sensitivity lists of the *evaluate* functions had only one element: the *eval* signal. This signal is modulated by the clock signal, so the process is only done once on every rising edge of the clock. The *if* condition checking the status of the *counter* variable also limits the extent to which these processes are actually active and updating values in the scaffold.

To allow the parser to better fit in the framework and take advantage of the independence of different elements in the scaffold, we first remove the dependency on the clock (by way of *eval*. Instead, we revise the sensitivity list to be *state* and *scaffold*, which are the current state of the framework FSM and the scaffold, respectively. This means that whenever either of those two parameters change, the *evaluate* process will be performed again. In addition, since we removed the *if* condition on *counter* (which also depended on the clock), we will not be held back by waiting on clock ticks. Now, when the FSM transitions into the parsing state and *state.eval* is set to 1, parsing can begin. Since we removed all reliance on the clock, the parser no longer has to wait for each clock cycle to update and operates entirely in combinational logic. This optimized function is shown in Figure 2.2.

Since the scaffold is included in the sensitivity list, each of the *evaluate* functions

will also update anytime a potential dependency is updated. Now, we can achieve gains by updating the scaffold multiple times per clock cycle. Additionally, independent items, that are not dependent on *counter*, can update immediately and in parallel instead of waiting for enough clock cycles to pass. Further, cells of the scaffold dependent on these items can also now be processed earlier. Now the parsing process takes as long as it takes to resolve the entire scaffold. The FSM waits until the parser has completely filled out the scaffold, which occurs when $A[0,0]$ changes from -2, indicating the cell has not yet been evaluated, to any other number. Then, if $A[0,0]$ is equal to the length of the input, the FSM moves on to the writing stage and transfers the packet back out. Otherwise, it drops the packet and moves on to read in the next packet.

```
1  evaluatei: process(state, scaffold)
2  begin
3    if state.eval = 'i' then
4      if scaffold(i)(0) = -2 then
5        scaffold(i)(0) <= compute_aplus(
6                            i, scaffold, 0, 1, state.length);
7      end if;
8      if scaffold(i)(1) = -2 then
9        scaffold(i)(1) <= compute_terminal(
10                           to_rtype(state.input, state.length),
11                           i, "01100001");
12     end if;
13     else
14       scaffold(i)(1) <= -2;
15       scaffold(i)(0) <= -2;
16     end if;
17 end process evaluatei;
```

Figure 2.2: Optimized Evaluate Function

2.2.2 Reversal Optimization

As discussed previously, the *Pegmatite* parsers parse the input from right to left, from the end to the beginning, so they cannot start until the entire input packet has arrived. In an environment where the entire input is already available, this is not a problem. But in a network setting where latency is important, and packets are arriving byte-by-byte, waiting for the entire packet is undesirable. In the end-to-end framework, when the first byte arrives, it must remain idle for $O(n)$ clock cycles until the rest of the input arrives before it can be processed. This leads us to another avenue to optimize parsing performance: we can try and flip the algorithm and instead operate on the input as it arrives, essentially filling the scaffold in reverse order. To do this, we make the following changes:

1. In the scaffold, $A[i, j]$ now represents how many characters non-terminal j will consume, starting at the beginning of the input and going up through the ith character. Since the length of the packet is unknown at the beginning of the parse, we fill the scaffold from the top down instead of bottom-up.

2. As a result of these changes, the functions that compute the values for each of the k rules needs to be modified to look at different elements of the scaffold.

3. In addition, since we are performing one set of operations (for each of the non-terminals) as every byte comes in, we do not need $n + 1$ evaluate functions. Instead, the values of the ith row in the scaffold can be computed as the ith byte arrives, so we can perform the parsing within the main FSM that is used to move data through the end-to-end pipeline. Figure 2.3 shows how the scaffold is filled out for the grammar a^+. *state.length* refers to the current

number of bytes seen so far, and *state.input* contains the input seen so far. Since each row is filled as its corresponding data comes in, the scaffold will be completely filled out immediately after the last byte arrives.

Then, with the new scaffold, we can instead check $A[n+1, 0]$ to see if the parser successfully consumed all the characters of the input. Now, there is limited overhead to parsing. Once the packet is completely read in, the scaffold should be complete, and all that is left to do is examine the scaffold to check for a successful parse. This method of acceleration has promise because the parsing overhead it incurs is independent of packet size. There are a few important things to note here. First, this reversal process is currently done manually, not automatically. Second, this process is will only work when the target language's reverse can be expressed in PEGs. The reversal optimization works for the languages we explore in Chapter 3, but it remains to be seen if it will translate to more complicated languages and how difficult it will be to reverse more complicated grammars.

```
1 scaffold(state.length)(1) <=
2     compute_terminal(
3     to_rtype(state.input, state.length),
4     state.length, "01100001");
5 scaffold(state.length)(0) <=
6     compute_aplus(state.length, scaffold,
7     0, 1, state.length);
```

Figure 2.3: Reversal Optimization: Filling Out Scaffold

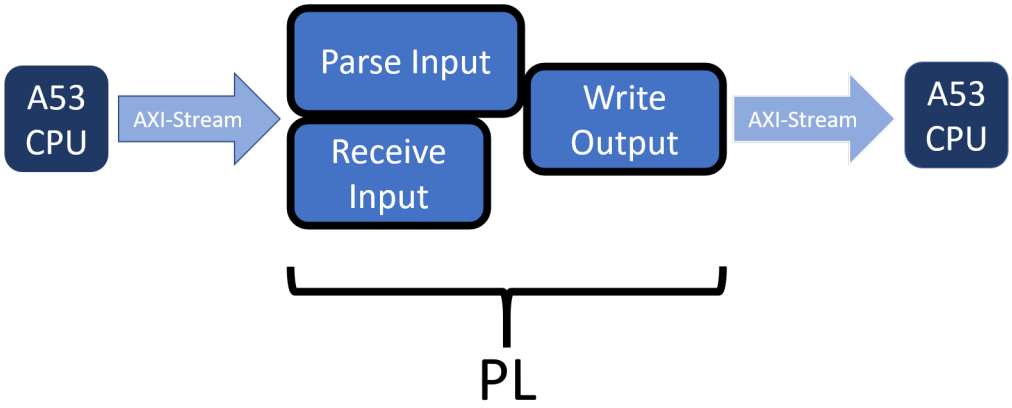

Figure 2.4: GAPS End-to-End Framework with Reversal Optimization

2.3 Further Pipelining

In addition to exploiting data parallelization in the parsing process, we initially also considered exploiting gains to further pipeline parallelization, performing different steps of the process on *different* input packets. In examining the process on the FPGA, we find three major steps:

1. Read in the packet
2. Parse the packet
3. Send out the packet, if it is accepted by the parser.

This pipelining will be bottlenecked by the slowest stage. In our experiments, we find that the reading and writing are the most significant stages, both taking $O(n)$ time to read in the packets. So, further pipelining would only introduce a limited speedup and would introduce significant additional complexity, as well as perhaps introducing some extra latency around transferring packets between

stages. For these reasons, we decided not to implement additional pipelining in this thesis.

Chapter 3

Results and Analysis

In this chapter, we discuss performance measurements of implementations of PEG parsers for two grammars. We present data from two types of measurements:

1. Running the end-to-end framework in simulation.

2. Running the end-to-end framework on the FPGA.

The reason for doing simulations is that running data through the parser on the FPGA does not provide details on clock cycles spent parsing. The simulations are performed in a Docker container using GHDL, an open-source VHDL simulator. The GHDL simulations give an estimate of the number of clock cycles spent in the parsing stage. After analyzing the simulations, we discuss our experiences and results running the framework on the FPGA.

3.1 Language: a^+

First, we look at a basic language: a^+, also known as Kleene plus. While the Kleene plus operator is not listed among the primitives in Table 2.1, it can easily be derived from them as seen in Table 3.1.

$S \rightarrow aA$
$A \rightarrow aA/\epsilon$

Table 3.1: Building a^+ from basic parsing expressions

The terminals of our languages are limited to 1-byte symbols, corresponding to the binary nature of packet specifications that will likely be common in the GAPS project setting. In the language a^+, only strings of a's are acceptable, so strings like $aaaaa$ would be accepted by the grammar while strings like $aaab$ or $ababa$ would be rejected. We follow the process described in Section 2.2.1 to generate an appropriate parser, implementing the parallelization optimization technique. Then, we perform simulations using GHDL. These simulations are used to determine the correctness of the parser as well as to estimate performance. GHDL does not accurately reflect the timing delays that occur as signals propagate in the combinational logic of the $evaluate$ functions. Without accounting for these propagation delays, the simulations show that an entire packet could be parsed in a constant number of cycles, regardless of the size of the packet. This is unrealistic given the structure of the parser, as it would require all the cells of the scaffold and all their dependencies to be filled out within that constant number of cycles, regardless of the length the packet. Xilinx's synthesis tools translate the VHDL code into instructions for the FPGA, configuring the various logic blocks and RAM to interact with each other

to implement the behavior specified by the VHDL. In the process, the tools report the time taken by the critical path, which is the path between an input and output with the longest delay. This delay provides an indicator of the maximum time it could take to complete assignment of various cells in the scaffold. Through synthesis of our parsers, we find critical paths in the range of 1-6 nanoseconds, so we conservatively account for these in our simulations by delaying the assignment of any cell in the scaffold by six nanoseconds. It is important to note that is not the same as adding a constant number of clock cycles to each simulation. Instead, this delay occurs at every assignment in the scaffold, so a cell x with a dependency on cell y will have to wait the six nanoseconds of simulated propagation delay before it can read an updated value from cell y. Similarly, a cell z dependent on x would have to wait an additional six nanoseconds. So, the longer a packet is and the more dependencies there are, the more impact the propagation delays will have.

3.1.1 Parallelization Optimization

Figure 3.1 shows the cycles spent in the parsing stage for packets of varying lengths from 8 to 128 bits. The "sequential" line is an estimate for a parser implemented without any of the optimizations we made in 2.2.1 that takes one cycle per byte of input to complete the parse. Even with the six nanosecond delay, the simulations show improvements over a sequential implementation, consistently spending less time in the parsing state than the initial sequential implementation.

Figure 3.2 shows estimated cycles when accounting for the reading and writing stages of the framework in addition to the parsing stage. We can see that our parallelization optimization is still better than a sequential implementation, but the

gains are held back by the time spent on receiving and sending the packet. Table 3.2 shows estimated throughput of just the parsing stage as well as the throughput of the entire PL section of the end-to-end framework. Calculations are based on a clock frequency of 100 MHz. At smaller packet sizes, the parallelized implementation provides similar performance to a sequential implementation, but at larger packet sizes, the parallelized implementation does have improved throughput. The gains in end-to-end throughput are more limited because of the significant cost of reading and writing the packet back out. Also, these improvements are such that the time spent parsing is is still linear in the size of the input, which is not ideal and leads us to try our second optimization technique.

3.1.2 Reversal Optimization

Using the reversal optimization as described in Section 2.2.2, we see significant reductions in parsing overhead. The simulations show a constant one cycle spent exclusively parsing the packet, regardless of the delay or size of the packet. This is because we start parsing as soon as the packet comes in, as explained in Section 2.2.2 and Figure 2.4. The parsing itself still takes $O(n)$ time, but it is overlapped with input part of the framework. Figure 3.1 illustrates how regardless of the size of the packet, the parser spends a constant number of cycles exclusively parsing the packet and not doing any other task. Figure 3.2 shows how in comparison to a framework that implements no parsing and instead just reads the packet in and writes it out, this optimization provides very similar performance. This shows a minimal additional impact to parsing. With the reversal optimization, parsing becomes essentially free; I/O is the bottleneck. We conjecture this may hold true

for many of the grammars we are able to reverse. Table 3.2 also shows how this optimization introduces minimal impact to estimated throughput versus a baseline of not parsing a packet at all.

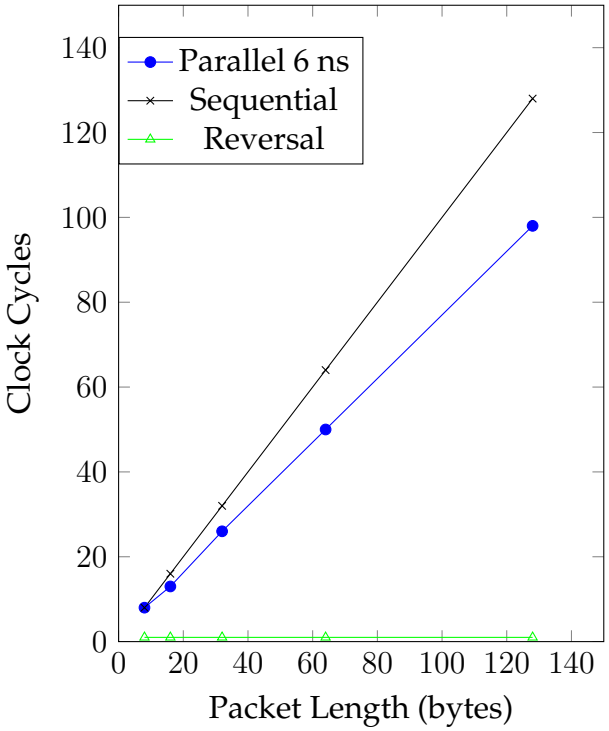

Figure 3.1: Estimated cycles spent in parsing stage

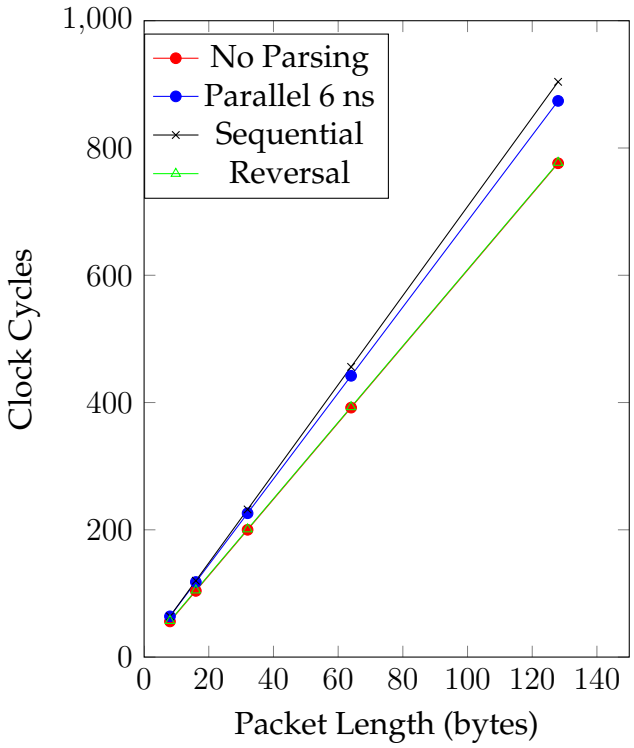

Figure 3.2: Estimated cycles to complete PL section of end-to-end framework

Type	Packet Size (bytes)	Parsing (Gbps)	Complete End-to-End (Gbps)
No Grammar	8	-	0.114
Sequential	8	0.800	0.100
Parallel 6 ns	8	0.800	0.100
Reversal	8	-	0.112
No Grammar	128	-	0.132
Sequential	128	0.800	0.113
Parallel 6 ns	128	1.045	0.117
Reversal	128	-	0.131

Table 3.2: Estimated Throughput from Simulations for Grammar a^+. Parsing throughput estimates for the reversal optimization are left blank because parsing does not occur as a separate stage and is overlapped with the receiving input stage

3.2 Language: $(a^5/a^3/a^2)^+$

This section explores a language which shows the impact of PEGs' prioritized choice versus the non-deterministic choice of CFGs. $(a^5/a^3/a^2)$ indicates a prioritized choice across the three rules. For each of the three rules, a^n indicates consuming a string of n a's. Consider the string $aaaaaaaaa$ (9 a's). This string would be able to be parsed by a CFG, in multiple ways, either using the a^3 rule three times, or the a^5 followed by the a^2 twice. However, with PEGs, the a^5 rule has to be used first, consuming five characters. Then, with four a's left, the a^5 rule will fail, and the next rule, a^3 will succeed, leaving only one a left. Then, none of the rules will succeed, and the parse fails. However, a string with twelve a's would pass both the CFG and the PEG. We explore this grammar because it involves more complex

rules in the scaffold and should therefore have more complex circuitry such that it takes more time to evaluate the cells of the scaffold. We do not explore parallel optimization because of the attractiveness of the constant one cycle extra time spent on parsing in the reversal optimization of a^+.

3.2.1 Reversal Optimization

Even with a more complex grammar, we are successfully able to evaluate each row within a single clock cycle (ten nanoseconds), so the impact of parsing versus the baseline of no parsing is again a single cycle. Xilinx's synthesis tools shows a maximum propagation delay of 5.43 nanoseconds which is less than the maximum propagation delay of a^+, so even with a more complex parsing process, we are able to complete the parsing process within a single cycle of completely receiving the packet.

3.3 Performance on the FPGA

Thus far, we have seen results from simulations. We focus on these for a couple reasons:

1. The end-to-end framework spends time generating packets, processing them, and verifying the output from the PL to ensure that the parser is operating correctly. In addition, the process for receiving packets introduces a sleep as we wait for a packet to be processed by the PL. We can only calculate time on the CPU, so it is hard to nail down how much of the time is actually spent by the PL to perform the parse versus how much is time spent on other functions

mainly performed by the CPU.

2. We had trouble consistently getting the framework to run correctly on the FPGA. There were problems with seemingly correct packets getting dropped, and packets timing out with no indication that they were dropped. As of the writing of this thesis, we have been unable to ascertain the cause of these results.

With all that said, we present some of our limited results here. We estimate performance by running 100,000 64-byte packets through the framework, minimizing variance as much as possible by precomputing packets and limiting verification as we have previously verified the correctness of these runs. We chose 64 bytes as the packet length because it performed consistently on the FPGA for grammar a^+. However, we were unable to get consistent performance for any packet length using the grammar $(a^5/a^3/a^2)^+$. Throughput is roughly estimated by measuring the total data processed and calculating the time taken to complete processing of 100,000 packets. Our baseline throughput estimate comes from the default settings of the end-to-end framework, which simply add 32 to each byte of the packet (bytes are in the range [0,223] to prevent overflow after adding 32). This should incur insignificant overhead as this addition occurs as the data is transferred to the output RAM to be sent back out to the PS. Table 3.3 shows our measured throughputs. It is encouraging that the difference between the baseline and the a^+ implementation is minimal and lines up with our expectation that with the reversal, parsing should introduce a small impact on throughput. However, it is hard to verify the exact single cycle impact because of the significant variability of results from run to run.

Grammar	**Throughput** (Mbps)
Baseline	46.77
a^+ Reversal	45.86

Table 3.3: Measured Throughput of 64-byte Packets on the FPGA on the PL section of the End-to-End Framework

3.4 Analysis Summary

The reversal-optimized parsers spend only one additional clock cycle on parsing beyond the time spent on receiving and sending out the packet. This is valuable because it reduces the latency impact of parsing, especially over our alternative method of optimization. Whereas the first method of optimization spends $O(n)$ cycles exclusively parsing the packet, a parser utilizing the reversal optimization exclusively parses a packet for only $O(1)$ additional cycles beyond the time required to receive the packet.

However, PEGs can be significantly more complex than the ones we explored in this paper. It is important to find out if a more complex language is more likely to have a longer propagation delay that exceeds the clock cycle. Such a language would not be able to parsed in $O(1)$ additional cycles. Anecdotally, the more complex language, $(a^5/a^3/a^2)^+$ had a propagation delay of 5.43 nanoseconds, which was less than the propagation delay of the simpler language, (a^+) (5.83 nanoseconds). The relationship between complexity of the language and propagation delay needs to be examined further.

Chapter 4

Conclusion

4.1 Summary

The major contributions of this thesis are:

1. The parallelization optimization, which allows us exploit independence in computation to start elements of the parsing process earlier and finish parsing faster.

2. The reversal optimization, which allows us to overlap parsing time with time spent receiving the input packet, reducing the impact of parsing.

4.2 Future Work

As discussed in Section 3.3, we would like to understand and solve the inconsistent performance of our parsers on the FPGA, so we can complete the integration into the end-to-end framework. Also, we would like to augment *Pegmatite*

to support automatic generation of our optimized parsers. In addition, we would like to expand the applications of the reversal optimization to more completely cover PEGs. Right now, we do not cover the entire set of PEG constructs, but we would like to do so. It would also be useful to formally prove that our reversal optimization produces a parser that accepts the same language as the original.

The length field occurs frequently in network packets and other binary formats that could be common in the GAPS project setting. At this point, it is not known whether PEGs can express the length field, so it would be a natural extension of PEGs to support a length field. Lucks et al. (2017) extended regular languages with a length field to introduce what they call calc-regular languages. Similarly, Lucas et al. (2021) introduced calc-PEGs, an extension of PEGs with explicit support for the length field. In addition, Lucas et al. (2021) introduce an extension of the scaffold automaton designed to handle parsing, essentially creating a scaffold for each character of the input. This too could be parallelized on an FPGA by filling out these scaffolds simultaneously just as we did here. The extension of VHDL parsers to support calc-PEGs on FPGAs could prove especially useful for the GAPS project.

As discussed in Section 2.3, we did not pipeline our framework because it appeared to introduce significant complexity to the VHDL code for somewhat limited gain. However, with more time, it could be worth exploring extracting additional throughput by simultaneously receiving a packet, parsing another packet, and sending a third packet back out, all at the same time

Another aspect of the GAPS project is that for some security policies, it may be that some of the data in a packet may need to be passed along, but some of the more specific information in the packet may be too sensitive. In a medical setting,

consider patient records and scientific research. These patient records contain potentially valuable medical data, but they also contain private and identifying data about patients. A security policy may be that any patient data traveling to scientists doing research must be stripped of any identifying information that could tie the data to the patient. A GAPS parser could potentially be used to scrub data traveling in this direction but let the medical data pass through. So, a future step could be adding the ability to not just parse and recognize a packet as being part of a language, but also modify it according to some security policy. However, this would introduce added complexity surrounding recognizing what needs to get replaced, and different formal specifications may be required.

Our reversal optimization significantly reduces the impact of parsing such that I/O to the FPGA is now the bottleneck. As discussed in Section 1.3.2, the end-to-end framework currently sends and receives a single byte at a time. The FPGA can support increasing this up to 32 bytes, so, while I/O is capped, there is room for improvement.

Bibliography

Ali, S., Anantharaman, P., Lucas, Z., and Smith, S. W. (2021). What we have here is failure to validate: Summer of LangSec. *IEEE Security Privacy*, 19:17–23.

Bordim, J. L., Ito, Y., and Nakano, K. (2003). Accelerating the CKY parsing using FPGAs. *IEICE Transactions on Information and Systems*, 86(5):803–810.

Bratus, S., Locasto, M. E., Patterson, M. L., Sassaman, L., and Shubina, A. (2011). Exploit programming: from buffer overflows to weird machines and theory of computation. *USENIX ;login:*, pages 13–21.

Ciressan, C., Sanchez, E., Rajman, M., and Chappelier, J.-C. (2001). An FPGA-based syntactic parser for real-life almost unrestricted context-free grammars. *Field-Programmable Logic and Applications*, pages 590–594.

DARPA (2019). DARPA explores new computing architectures to deliver verifiable data assurances. `https://www.darpa.mil/news-events/2019-01-16`.

Falcon, M. D., Bratus, S., Hallberg, S. M., and Patterson, M. L. (2016). The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. *IEEE SecDev*.

Ford, B. (2004). Parsing expression grammars: A recognition-based syntactic foundation. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 111–122.

Fruhlinger, J. (2017). What is the heartbleed bug, how does it work and how was it fixed? `https://www.csoonline.com/article/3223203/what-is-the-heartbleed-bug-how-does-it-work-and-how-was-it-fixed.html`.

Loff, B., Moreira, N., and Reis, R. (2020). The computational power of parsing expression grammars. *Journal of Computer and System Sciences*, 111:1–21.

Lucas, Z. S., Liu, J. Y., Anantharaman, P., and Smith, S. W. (2021). Research report PEGs with length in software and hardware. *IEEE Security & Privacy: LangSec Workshop*.

Lucks, S., Grosch, N. M., and König, J. (2017). Taming the length field in binary data: Calc-regular languages. *2017 IEEE Security and Privacy Workshops (SPW)*, pages 66–79.

Pomerantz, D. (2019). MIND the gap: DARPA funds new research to make information more secure. `https://www.ge.com/news/reports/mind-the-gap-darpa-funds-new-research-to-make-information-more-secure`.

Zhao, Z., Sadok, H., Atre, N., Hoe, J. C., Sekar, V., and Sherry, J. (2020). Achieving

100Gbps intrusion prevention on a single server. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 1083–1100.