Report Number: WUCSE-2006-36

2006-01-01

HAIL: An Algorithm for the Hardware Accelerated Identification of Languages, Master's Thesis, May 2006

Charles M. Kastner

This thesis examines in detail the Hardware-Accelerated Identification of Languages (HAIL) project. The goal of HAIL is to provide an accurate means to identify the language and encoding used in streaming content, such as documents passed over a high-speed network. HAIL has been implemented on the Field-programmable Port eXtender (FPX), an open hardware platform developed at Washington University in St. Louis. HAIL can accurately identify the primary languages and encodings used in text at rates much higher than what can be achieved by software algorithms running on microprocessors.

2006-36

HAIL: An Algorithm for the Hardware Accelerated Identification of Languages, Master's Thesis, May 2006

Authors: Charles M. Kastner

Corresponding Author: lockwood@arl.wustl.edu

Web Page: http://www.arl.wustl.edu/projects/fpx/reconfig.htm

Abstract: This thesis examines in detail the Hardware-Accelerated Identification of Languages (HAIL) project. The goal of HAIL is to provide an accurate means to identify the language and encoding used in streaming content, such as documents passed over a high-speed network. HAIL has been implemented on the Field-programmable Port eXtender (FPX), an open hardware platform developed at Washington University in St. Louis. HAIL can accurately identify the primary languages and encodings used in text at rates much higher than what can be achieved by software algorithms running on microprocessors.

Type of Report: Other

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

HAIL: AN ALGORITHM FOR THE HARDWARE-ACCELERATED IDENTIFICATION OF LANGUAGES

by

Charles M. Kastner, B.S.

Prepared under the direction of Professor John W. Lockwood

A thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

May 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

HAIL: AN ALGORITHM FOR THE HARDWARE-ACCELERATED IDENTIFICATION OF LANGUAGES

by
Charles M. Kastner

ADVISOR: Professor John W. Lockwood

May 2006
Saint Louis, Missouri

This thesis examines in detail the Hardware-Accelerated Identification of Languages (HAIL) project. The goal of HAIL is to provide an accurate means to identify the language and encoding used in streaming content, such as documents passed over a high-speed network. HAIL has been implemented on the Field-programmable Port eXtender (FPX), an open hardware platform developed at Washington University in St. Louis. HAIL can accurately identify the primary languages and encodings used in text at rates much higher than what can be achieved by software algorithms running on microprocessors.

To my parents,
for their love and
financial support;
To Bridget, for her
love and encouragement;
and to all my family
and friends, for
always supporting me.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my advisor, Dr. Lockwood, for his assistance in my graduate education. He supported my research and provided suggestions and constructive criticism.

Additionally, I would like to thank my research sponsor, Science Applications International Corporation (SAIC), for funding my research and providing me with experience I never would have gained without their support.

This work could not have been completed without the support and patience of current and former members of the Reconfigurable Network Group in the Applied Research Laboratory.

Thanks to James Moscola for his assistance with VHDL and the workings of the FPX platform; Todd Sproull for help with the NCHARGE software used for programming the FPX; Adam Covington and Andrew Levine for software tools that were used in the early stages of this project; Moshe Looks and Charlie Comstock for entertaining my many questions regarding probability and statistics; and finally the members of the group (current and former) whose advice has been very helpful: Mike Attig, Sarang Dharmapurikar, Phillip Jones, Jing Lu, Jack Meier, Jeff Mitchell, Shobana Padmanabhan, Abdel Rigumye, Dave Schuehler, Haoyu Song, and Chris Zuver.

Charles M. Kastner

Washington University in Saint Louis
May 2006

Chapter 1

Introduction

1.1 Motivation

In 2004, nearly two-thirds of the world's Internet users spoke a language other than English as their primary language (see Table 1.1), and nearly one-third of the pages available on the World Wide Web were written in a language other than English [40]. As the amount of multilingual content available on the Internet and in private storage systems increases, efficiently finding useful and relevant information becomes a growing problem.

Table 1.1: Internet Users by Language, 2004 [39]

Native Language	**Internet Users (Millions)**	**Percent of Internet Users**
English	287.5	35.64%
Chinese	102.6	12.71%
Japanese	69.7	8.64%
Spanish	65.6	8.13%
German	52.9	6.55%
Korean	29.9	3.70%
French	28.0	3.47%
Portuguese	25.7	3.18%
Italian	24.3	3.01%
Russian	18.5	2.29%
Malaysian	13.6	1.69%
Dutch	13.5	1.67%
Arabic	10.5	1.30%
Polish	9.5	1.18%
Other	55.3	6.85%
Total	807.1	

A system capable of quickly identifying the primary languages and character encodings used in text documents can ease many problems associated with the growing amounts of multilingual data. The primary use of this system is for language-based forwarding and data routing, such as technical support e-mail messages in a large international organization. Such a system could also be used as a preprocessor for document classification services that must know the language and encoding of characters in order to determine the semantic meaning of a text document. Moreover, the system could label documents in a large database or remove documents in unwanted languages from a data stream.

Although modern microprocessors continue to steadily increase in performance, they are not increasing as fast as the data rate of backbone networks. As the limits of Moore's Law are reached in the next ten years, the gap between the rate at which data needs to be processed and the rate at which a microprocessor can process data will increase further.

In contrast, reconfigurable logic can process network traffic at rates much faster than what is achievable with microprocessor-based systems. Systems created from Field-Programmable Gate Arrays (FPGAs) are flexible, since they can be easily modified to provide new functionality. They can be programmed with highly customized hardware circuits to drastically outperform microprocessor-based systems.

1.2 Thesis Objectives

This thesis set out to create an algorithm and hardware architecture to perform identification of languages[1] with the following objectives in mind:

- Create a language identification algorithm that is simple, fast and accurate.
- Enable the system to be trained on a minimal amount of labeled data.
- Create a software tool to demonstrate the functionality of this algorithm.
- Allow parameters within the software tool to be changed.
- Track changes in the algorithm's performance as these parameters are altered.
- Evaluate algorithmic optimizations for language identification.

[1]Unless otherwise specified, the terms "language identification," "identification of languages," and similar phrases will be used to describe the simultaneous identification of both a document's language and its character encoding.

- Create a hardware system that performs this language identification algorithm.
- Ensure that the system is synthesizable in a wide variety of reconfigurable hardware devices.
- Design the system so that languages and encodings can be added or removed without affecting the architecture itself.
- Create the architecture in a modular fashion so that it can be easily augmented for additional functionality.

1.3 Thesis Outline

This thesis begins with a background of character encodings, their history, and their use with various languages. A brief examination of existing algorithms used for language identification, along with their respective strengths and shortcomings, is provided.

The design of the HAIL system is presented and the design tradeoffs incorporated to implement it in reconfigurable hardware are described. The descriptions are accompanied by results of experiments that affect HAIL's performance and motivate design decisions.

The final section details an implementation of HAIL that was created for the Field-programmable Port eXtender (FPX) platform. A brief introduction to the platform and infrastructure components that were utilized in the design precedes a detailed description of the system's architecture and results obtained with this implementation.

Appendices are provided for reference. Appendix A identifies the location of HAIL's source files and instructions for creating an implementation. Appendix B describes the HAIL software tool and other software created to support HAIL. Appendix C discusses the laboratory configuration used to test the system. Appendix D provides additional figures. Appendix E contains a list of acronyms used throughout this thesis.

Chapter 2

Background

To understand the context of this project and the difficulties of the work, this section provides relevant background material. First, a description of character encodings and their use to represent text in different languages is presented. Second, a summary of related work on language and character encoding identification is provided with explanations regarding benefits and drawbacks. Third, relevant concepts from probability, statistics and machine learning are explained. Finally, a description of multiple implementation platforms is presented with their advantages and disadvantages.

2.1 Character Encodings

Character encodings assign numeric values to characters (letters, numbers, and symbols) so that computers can represent text in natural languages. Approximately 250 character encodings have been registered with the Internet Assigned Numbers Authority (IANA) [75], the organization that also regulates Internet Protocol (IP) addresses and top-level domains.

Identifying the character encoding of a document is a necessary part of language identification. Many encodings are in common usage, and most languages can be represented with more than one character encoding. While the character encoding of a document is sometimes stated within the document itself, there is no reliable standard between differing document formats for expressing this information. Therefore, explicit labels cannot be expected and will not be used for purposes of language and character set identification.

The purpose of this section is to clarify character encodings and to illustrate how the number of encodings complicates the issue of language identification.

2.1.1 ASCII

Many modern character encodings are extensions of the American Standard Code to Information Exchange (ASCII), which was a standard finalized in 1968 under the name ANSI X3.4 [31]. ASCII uses seven bits to encode 128 characters, 95 of which are printable and 33 of which are used primarily for control purposes.

Among the printable characters, only the uppercase and lowercase forms of the Latin alphabet's 26 letters can be represented with ASCII. As computers became more widespread and diverse, the need to represent more printable letters was realized.

2.1.2 Extended ASCII

"Extended ASCII" is a term loosely used to describe a wide variety of character encodings. Encodings referred to as extended ASCII use eight bits to encode 256 characters, the first 128 of which are the characters from the original ASCII standard. Many extended ASCII character encodings have been created over time.

ISO 8859 Character Encodings

One of the most popular extended ASCII character encodings is the International Organization for Standardization 8859 (ISO 8859) series [4]. ISO 8859 is a set of fifteen different eight-bit encodings that together can represent text in many different languages. Ten of these encodings are variations on the Latin alphabet; the other five include encodings for the Cyrillic, Arabic, Greek, Hebrew and Thai alphabets.

The most prevalent of these character encodings is ISO 8859-1, often referred to as *Latin 1.* It is used to represent characters in western European languages [55], including Albanian, Basque, Catalan, Danish, Dutch, English, Finnish, French, German, Greenlandic, Icelandic, Irish, Italian, Latin, Norwegian, Portuguese, Scottish, Spanish and Swedish.

Other Extended ASCII Character Encodings

When the first IBM PCs became available in the early 1980s, the ISO 8859 standard had not yet been created. IBM and Microsoft created their own character encoding, Code Page[1] 437 (CP437), for use on the operating system for IBM PCs (PC-DOS) [29]. This encoding added some characters used in non-English languages and a large number of basic symbols and graphics. It replaced many of the existing ASCII control characters with simple graphics.

As the use of IBM PCs spread, CP437 was augmented with over fifteen other code pages; some of these were updates of CP437 that were compatible with the Video Graphics Array (VGA) computer display standard. Other code pages replaced some of the characters in CP437 with those relevant to languages of other regions.

As Microsoft Windows began to replace PC-DOS (and its open-market companion, MS-DOS) on most personal computers, the use of IBM code pages became less common. Today, they are used primarily to provide legacy support. In place of the IBM code pages, Microsoft encouraged the use of its own line of character encodings [52] which includes ten extended ASCII code pages. While most of these encodings are very similar to various ISO 8859 encodings, they are not exactly the same. Each encoding adds printable symbols to a range of 32 characters that is unused by the ISO 8859 encodings. Most Microsoft encodings rearrange a few of the characters in a way that makes them incompatible with the corresponding ISO 8859 code pages [29].

A variety of other extended ASCII character encodings exist [29]. Apple, Adobe and Hewlett-Packard developed their own character encodings; although they are used less frequently than ISO 8859 and Windows code pages, the are still encountered on occasion. Other character encodings are targeted towards specific regions of the world. For instance, the set of *Kod dla Obmena i obrabotki Informacii* (KOI) character encodings [30] is used extensively in countries where the Cyrillic alphabet is prevalent. The Vietnamese Standard Code for Information Interchange (VISCII) encoding [33] is used for Vietnamese.

2.1.3 Double-Byte Character Encodings

While most of the world's languages can fit all of their symbols within a single byte, this is not the case for a few East Asian languages, namely Chinese, Japanese and Korean. These languages use thousands of different characters (called *ideographs*). A single byte can encode no more than 256 different combinations. On the other hand, double-byte character encodings allow for up to 65,536 combinations.

[1] "Code page" is used by some as a synonym for "character encoding."

The first of these East Asian character encodings, Japanese Industrial Standard (JIS) X 0208, was created in 1976 [32]. This encoding contains the three alphabets used in the Japanese language, as well as letters from the Latin, Cyrillic and Greek alphabets. It supports a maximum of 8,836 symbols, but has since been expanded by JIS X 0212 which added space to define 8,836 more symbols.

JIS X 0208 was followed by a Chinese encoding called Guojia Biaozhun (GB) 2312. GB 2312 is structured similar to the original Japanese encoding in that it supports a maximum of 8,836 symbols.

2.1.4 Unicode

As previously suggested, there were many standards for character encodings that developed as computers gained popularity through the 1970s and 1980s. To reduce the problems caused by hundreds of incompatible character encodings, a non-profit organization called the Unicode Consortium [78] was formed in 1991. The consortium created the Unicode Standard which assigns a unique number to every character used in writing.

Unicode has been adopted by Apple, HP, IBM, Microsoft, Oracle, Sun and numerous other companies. It is also the underlying encoding of many standards such as XML and Java [78]. It can represent text in thousands of languages and dozens of scripts, both modern and historic [77].

The Unicode Standard in itself, however, is not a character encoding. In fact, the standard defines three different encodings, or Unicode Transformation Formats (UTFs) [76]: UTF-8, UTF-16, and UTF-32. All three can represent every character in the Unicode standard. The primary differences are detailed below:

- UTF-8 represents every ASCII character with a single byte; however, characters beyond ASCII require a variable number of bytes.
- UTF-16 stores most characters using two bytes, but represents a number of uncommon characters with four bytes.
- UTF-32 stores all Unicode characters in four bytes. The advantage is that all bytes are represented in a fixed amount of space. The disadvantage is that UTF-32 requires more space than the other two encodings in most situations.

Another type of encoding, UTF-7, is frequently used to transfer e-mail messages. UTF-7 is used to encode other Unicode formats into an ASCII text representation since many e-mail protocols forbid the transmission of data outside of the 7-bit ASCII range. However, it is rarely used by computers to store data internally.

2.2 Existing Language Identification Algorithms

To date, several mechanisms for computer-based language identification have been developed.

2.2.1 Dictionary-building

An obvious method of language identification is dictionary-building as outlined in a patent written by by Paulsen, et al. [44]. The process of dictionary-building extracts whole words from a set of training data. These words are stored in a data structure called a word frequency table (WFT). This table contains a list of the most common words found in the set of training documents and applies optional frequency-based weights to each word.

Once the system is trained with a sufficient amount of data, the algorithm can identify languages in documents by using table lookup operations. Words in a data stream are extracted and looked up in the table. If a word is found, a counter for the appropriate language(s) is incremented. Once the entire document is processed, the language can be discerned by examining the highest language counter.

The authors of the patent argue that their algorithm is superior to others because it requires fewer table lookups than other existing methods (see Section 2.2.2). They suggest that only the most common words in a particular language must be stored.

A limitation is that the algorithm is dependent upon knowledge of the underlying character encoding. For example, while the byte sequence *247 161* is interpreted as non-alphabetic symbols in many Windows and ISO 8859 encodings, it is a valid Chinese ideograph in the GB 2312 character encoding. Thus, the language identification engine must be aware of the encoding scheme before determining the language. Although some documents are labeled with their character encoding, this is not always the case. Different document formats identify their encoding at different points within the file and can use a variety of different labels to represent the encoding.

2.2.2 N-Gram Frequency Analysis

The analysis of patterns of exactly N successive characters, known as *n-grams*, is an effective method for language identification. N-gram based methods for the identification of languages have been developed by various authors, including Schmitt [66] and Huffman [42]. Damashek [34, 35] and Cavnar [23] broadened the use of n-grams by using them as the basis of topic identification algorithms.

Schmitt's language identification algorithm is a typical n-gram based classification method. Schmitt's method makes use of trigrams – sequences of three successive characters. This method extracts every trigram within a document and compares the extracted patterns with those in a pre-established library of trigrams.

Once all of the trigrams within a document have been looked up, the number of extracted trigrams that represent each language is divided by the total number of trigrams within the document. If this ratio exceeds a threshold for a particular language of interest, the document is classified as belonging to that language.

In general, using n-grams is more flexible than a dictionary-based method. While a word-counting algorithm is dependent on knowledge of the character encoding so that word boundaries can be determined, n-grams are not. N-grams can be sampled at every byte offset within a document, overlapping not only letters but also numbers and symbols. Due to this advantage, n-grams are the underlying language-identification method used by HAIL.

2.3 Probability and Statistics

Principles from probability and statistics are used in several areas throughout this paper. A brief overview of relevant probabilistic and statistical principles is outlined below.

2.3.1 Linear Regression

Linear regression [53] is a technique that fits a line to a set of observed data points. Once an equation representing the best-fitting line is determined, values for which no data is available can be estimated by solving the equation for a particular value of the independent variable. The following equations demonstrate the process of approximating a line to a set of data points. Derivations for the following equations can be found in [53].

$$b_1 = \frac{n\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2} \tag{2.1}$$

The slope b_1 of a line created to fit data points can be calculated with Equation 2.1. In the equation, x_i and y_i are respectively the independent and dependent coordinates of the observed data points. n is the number of observed data points used to estimate the regression line.

$$b_0 = \bar{y} - b_1\bar{x} \tag{2.2}$$

Once the slope is found, the vertical intercept b_0 can be calculated with Equation 2.2. b_1 is the slope obtained in Equation 2.1 while $\bar{x}$ and $\bar{y}$ are respectively the average independent and dependent coordinates of the data points.

$$\widehat{y} = b_0 + b_1 x \tag{2.3}$$

The slope b_1 and vertical intercept b_0 are used in Equation 2.3, the slope-intercept formula frequently used to represent linear equations. $\widehat{y}$ indicates that the value of the dependent variable is being estimated rather than calculated exactly.

2.3.2 Bayes' Theorem

Bayes' Theorem [6] is used to calculate the conditional probability of a variable, A, given another variable, B. This conditional probability is expressed mathematically as $P(A \mid B)$ and is typically referred to as the *posterior probability.*

The theorem assumes that one already knows the probability of B given A, $P(B \mid A)$, the probability of A, $P(A)$ (also known as the *prior probability*), and the probability of B, $P(B)$. Thus, it is typically applied when $P(B \mid A)$, $P(A)$, and $P(B)$ are easily observed, while $P(A \mid B)$ is difficult to observe. Conventionally, the theorem is represented with the following equation:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{2.4}$$

To exemplify Bayes' theorem, consider the following statement: "Thirty percent of dogs provided to American animal shelters are purebred" [65]. This can be expressed as "The probability of a dog being purebred given that it was provided to an American animal shelter is 0.3" or $P(B \mid A) = 0.3$. While this statistic can be confirmed easily by studying American animal shelters, it is much more difficult to measure $P(A \mid B)$ or "The probability that a dog was provided to an American animal shelter given that it is purebred". In addition to $P(B \mid A)$, one must also know $P(A)$, "The probability that a dog is purebred" and $P(B)$, "The probability that a dog is provided to an American animal shelter." Both of these values are also easily obtained through studying dog owners. Together with $P(B \mid A)$, the values can be used to calculate the probability that a dog is provided to an American animal shelter given that it is purebred.

2.3.3 Naive Bayes Classifiers

Statistical classification is a process in which items are labeled as belonging to one of numerous groups. This grouping is based upon statistical analysis of the features that appear in items to be classified and the features that appear in a *training set*, which contains already-labeled items. HAIL is a statistical classifier; it assigns labels (languages) to items (documents) based on a training set of documents for which the language is already known.

A naive Bayes classifier is a method for statistical classification. Such a classifier is "naive" because it assumes characteristics within an item are independent of one another. With some exceptions (see Section 3.2.3), HAIL operates under these assumptions and is therefore a naive Bayes classifier.

As an example, consider a naive Bayes, n-gram based language identification system. If the system were requested to identify the language of a document containing the word *there*, it might extract the n-grams *ther* and *here*. The presence of *here* given the presence of *ther* immediately before it may be a strong indicator that the document is written in English. However, a naive Bayes classifier would independently determine the probability that *ther* and *here* appear in various languages and give no weight to the fact that they occurred together.

2.4 Data Clustering

Data clustering ("clustering") [64] is a process in which items in a data set are partitioned into subsets, called clusters. Items judged to be similar are placed into the same clusters. The measure of similarity is based upon a *distance metric* which must be defined for the set of data in question. The difference between classification (defined in the previous section) and clustering is that during classification, items are compared to data obtained from a training set; during clustering, items are compared to one another.

Clustering is an extensively studied topic. Several subtypes of clustering have been defined, and many different algorithms have been developed. Details of clustering relevant to this thesis are outlined below.

2.4.1 Expectation Maximization

An expectation maximization (EM) algorithm [36] is an iterative approach designed to estimate optimal parameters in a system; frequently, EM is applied as as clustering algorithm. EM involves the manipulation and observation of known variables, while attempting to maximize a function of unknown variables, called *latent variables.*

The expectation maximization process involves two repeated stages: The expectation step, in which the present state of the system is evaluated, and the maximization step, in which the state is changed so that the function of the latent variables is increased.

In EM clustering, $P(B \mid A)$ represents the probability that the set of data, B, fits into the current set of cluster assignments, A. However, in a clustering problem, one wants to determine $P(A \mid B)$ or the probability that the current cluster assignments fit the set of data. While $P(B \mid A)$ is easily calculated from the values of the data points and the properties of the cluster assignments, $P(A \mid B)$ is not easily observed.

Recall that Bayes' Theorem (Equation 2.4) can be used to find the value of $P(A \mid B)$ when $P(B \mid A)$, $P(A)$, and $P(B)$ are known. Expectation maximization is linked to Bayes' Theorem because the probability $P(B \mid A)$ that the data fits into the cluster assignments is much simpler to observe than the probability $P(A \mid B)$ that the cluster assignments fit the data.

The formula representing Bayes' Theorem also contains the terms $P(A)$ and $P(B)$. Literally, $P(A)$ represents the probability that a particular feasible clustering assignment will occur, independent of the data set. $P(B)$ represents the probability that a particular feasible data set will occur. Given a set of possible clusterings and possible data points, $P(A)$ and $P(B)$ are constant. Thus, Bayes' Theorem could be expressed as

$$P(A \mid B) = K * P(B \mid A) \tag{2.5}$$

K is a positive constant equal to the quotient of $P(A)$ and $P(B)$. Based on this equation, $P(A \mid B)$ and $P(B \mid A)$ are proportional; an increase or decrease in the latter will respectively increase or decrease the value of the former term. Therefore, regardless of the value of K, maximizing the value of $P(B \mid A)$ will also maximize the value of $P(A \mid B)$.

Therefore, in order to maximize the function of the latent variables one needs only to maximize a function of the known variables. In clustering, maximization can be achieved by repeatedly changing the cluster assignment in a way that makes the data fit the clusters more closely and evaluating the properties of the clusters. In time, the clusters will reach a state in which no single change to the cluster assignment will increase the probability of the data fitting the clusters.

2.4.2 Simulated Annealing

Simulated annealing (SA) [45] is an approach utilized to find the optimal state of a system. The process is inspired by annealing, a metallurgical process in which a material is heated so that atoms are forced to move about randomly. As the temperature of the material decreases, the atoms have a better chance of configuring themselves to a lower-energy form than their initial state. Materials with lower internal energy are larger and have fewer defects than ones with higher energy.

In HAIL, simulated annealing is applied to the expectation maximization algorithm. Without simulated annealing, the EM algorithm can force the system into a state that is more optimal than nearby states, but less optimal than other possible configurations. Such a state is called a *local optimum*. SA increases the chance that the system will enter a globally optimal state.

Consider the discussion of EM and clustering outlined above. In the maximization step, changes to the clusters are allowed if *and only if* they increase the probability that the data fits the clusters. This will cause the overall cluster configuration to improve, but it may converge on a locally optimal clustering.

SA changes this rule by incorporating a parameter referred to as the "temperature." Initially, the temperature is set to a high value. When the temperature is high, changes in the clustering are allowed to can *worsen* the probability that the data fits the clusters. As the iteration count increases, the temperature is decreased. At lower temperatures, changes in the clustering that worsen the probability may still occur, but the amount by which the probability can be worsened is reduced. As time passes, the temperature is reduced to 0. At this point, only changes in the clustering that improve the probability are allowed.

Due to the random nature of changes in the probability when the temperature is high, SA provides a chance for the system to move past a local optimum and increases the chance that it converges on the global optimum. However, it is by no means a guarantee that an optimal solution will be found.

2.5 Implementation Medium

Language and character encoding identification can be performed on a variety of platforms, ranging from general-purpose processors to customized integrated circuits. The strengths and weaknesses of various platforms are discussed below.

2.5.1 General Purpose Processors

General purpose processors (GPPs) are the heart of servers, desktop, and notebook computers. The typical GPP is designed to integrate within a system of memory, disk drives, and I/O devices.

Although flexible, GPPs do not achieve high performance for language identification. This is due to the serial execution of tasks via a stream of instructions which does not make use of inherent parallelism within language identification algorithms.

GPPs are particularly ill-suited for the problem of language identification. Modern GPPs rely on a cache to improve memory throughput, and therefore, performance.

However, caches only offer a significant performance improvement when the program under execution exhibits *temporal locality*, a property in which individual memory locations are accessed repeatedly in a short amount of time, or *spatial locality*, in which memory locations that are close to one another are accessed within a short amount of time of each other.

When documents are processed by a language identification algorithm, the data within the documents is typically accessed sequentially; this is an example of spatial locality. However, once document data is processed, it is not revisited. Thus, there is little temporal locality within the documents. Furthermore, the data within a document must be looked up inside a table or other data structure to identify the language(s) associated with the data. These lookups are independent of one another, and therefore exhibit poor temporal and spatial locality.

A large cluster of GPPs could match the performance of other implementation platforms, but would be more costly, require more space, and consume more power.

2.5.2 Network Processors

Network processors, such as the Intel IXP 2400 and 2800 [43], contain a single GPP used for control purposes, as well as a number of small, specialized microprocessors, often referred to as *microengines*.

The IXP 2800 contains sixteen microengines. Each microengine has a large number of general-purpose and specialized registers and hardware support for switching between eight threads of execution. Engines share interfaces to external static random access memory (SRAM) and synchronous dynamic random access memory (SDRAM). The microengines in the IXP 2800 support a limited version of the C programming language and can efficiently execute programs of a few kilobytes in size.

Despite their advantages, network processors still perform tasks by a serial execution of instructions. While they do have some optimizations for hardware-accelerated processing of streaming data, they do not have explicit hardware support to accelerate the processing of data for language identification.

2.5.3 Application-Specific Integrated Circuits

An application-specific integrated circuit (ASIC) is a customized device designed to perform a specialized task at high rates and/or low power consumption. If a system's design specification is rigid and will never change, and if the product is targeted towards a very large number of consumers, an ASIC is the ideal solution.

However, an ASIC has drawbacks. First, the generation of the mask used to produce a set of ASICs has a large non-recurring expense (NRE) measured in millions of dollars [46]. In order to cover the NRE, the system must either have a large target audience or sell chips at excessively high prices. Second, the high price of manufacturing ASICs means that post-production changes are virtually impossible.

As a result, an ASIC is not the optimal choice of technology for performing language identification.

2.5.4 Field-Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) allow for the creation of customized, high-speed, and flexible hardware circuits. FPGAs contain a matrix of configurable logic blocks (CLBs) that consist of look-up tables (LUTs) for the implementation of combinational logic and small memory elements, and flip-flops for the creation of registers, sequential logic, and pipelines. Most FPGAs also contain blocks of low-latency random access memory (RAM).

When creating a circuit for implementation in an FPGA, a hardware designer can use a hardware-description language (HDL) such as VHDL or Verilog to create a representation of the circuit. He or she can then use tools to compile and simulate the design, synthesize the circuit into a low-level register-transfer level (RTL) description, and map the design into the CLBs on the target FPGA.

FPGAs achieve slower clock rates than ASICs because of extra routing delays in the reconfigurable fabric. However, they do have several important advantages. First, creating FPGA circuits does not require the hardware designer to pay large mask creation and fabrication costs. Second, new functions can easily be added to FPGAs, since their CLBs can be reprogrammed many times. FPGAs are ideal for designs with production targets up to tens of thousands of units, and systems that require flexibility and modifications in functionality. The HAIL algorithm presented in this paper has been implemented in FPGA technology, because it can take advantage of reconfigurability.

Chapter 3

The HAIL Algorithm

The crux of the HAIL project is a language-identification algorithm that can be implemented in reconfigurable hardware. This chapter describes the algorithm's objectives, an overview of the algorithm, various algorithmic parameters that can be altered, and experiments performed to achieve optimal performance by altering these parameters.

3.1 Introduction

At the highest level, HAIL is a variation of existing n-gram based language identification methods such as those developed by Schmitt [66] and Huffman [42]. HAIL contains many optimizations for implementation in reconfigurable hardware. Some of these optimizations, such as the use of only one n-gram size (Section 3.2.1), are straightforward. Others, such as the *trend register* (Section 3.2.3) and data clustering to simplify the layout of memory (Section 3.3), are novel. This section provides an overview of the objectives that drove HAIL's design and a high-level summary of the algorithm's functionality.

3.1.1 Algorithm Objectives

Several objectives were identified during the creation of the HAIL algorithm.

The first of these objectives is to provide high throughput. Numerous software algorithms have been developed for the purpose of language identification; most are capable of achieving high levels of accuracy for documents as small as a few dozen characters in size. While there is little room for improvement in accuracy, these methods typically require large amounts of processing time. Most existing algorithms cannot process more than a few megabytes of data per second on a modern personal computer or server. As stated earlier, there is a use for systems capable of performing language identification rapidly.

The second objective is accuracy. Most existing language identification algorithms are capable of highly accurate operation. If a language identification algorithm is extremely fast yet comparatively inaccurate, it is of limited interest to those interested in language identification. To be of significant use, a new algorithm should be as accurate as possible.

The third objective is latency, or the amount of data that must be analyzed in order to conclusively identify a document's language. This metric is particularly important when processing streaming network data, since documents are broken into packets which can be quite small. Ideally, the identification of a document's language should occur during the first packet so that the entire document can be handled uniformly by any further processing being performed.

The fourth objective is efficient implementation in hardware. For instance, although floating point mathematics are used in many existing language identification algorithms, units to perform floating point operations consume a large amount of FPGA resources. HAIL does not rely on floating point mathematics, and can therefore fit into lower-end FPGAs for which a floating point unit is not feasible. Furthermore, if placed into a modern FPGA, HAIL will require only a small percentage of FPGA resources and allow other components to be integrated in the same device.

The final objective of the algorithm is flexibility. HAIL has been implemented on the Field-Programmable Port eXtender (FPX), an open platform developed at Washington University in St. Louis. However, this chapter contains few platform-specific details; the algorithm can easily be implemented on a variety of systems. The only requirements for an implementation of HAIL are an FPGA of sufficient size and one or more banks of memory, such as SRAM, capable of high-throughput random access lookups. Optionally, one or more banks of SDRAM can be used to store the state of documents in the event that they are interleaved with one another as in streaming network data.

3.1.2 Algorithm Overview

Before the process of language identification can begin, HAIL must be trained. Training data is a set of documents in which the language and character encoding are already known. Once training data is obtained, the following steps are performed:

1. Extract fixed-length sequences of n bytes (n-grams) at every byte offset within the training documents.
2. Use a hash function to transform each n-gram into a value that will serve as a memory address.
3. Calculate the frequency at which each hashed n-gram appears in documents of a given language.
4. Cycle through each hashed n-gram. In each language, subtract the average frequency at which the hashed n-gram appears across all languages. This serves to eliminate tenuous affiliations between languages and n-grams.
5. Perform the slot clustering operations outlined in Section 3.3.
6. Load memory bank(s) with the results of slot clustering operations.

Once memory has been loaded, HAIL can begin identifying the languages of documents. As data enters the system, the circuit extracts n-grams from the data stream. The n-grams are hashed using the same hash function from step 2.

These hash values are used as memory addresses; the corresponding addresses are read and numerical language identifiers that represent language and character encoding pairs are output. These language identifiers use a counting scheme (Sections 3.2.3 and 3.3) to find the language or languages that best match the document.

3.2 Algorithmic Analysis

The following section evaluates various design choices made during the design of HAIL to create an algorithm that follows the constraints detailed in Section 3.1.1. The trade-offs and optimizations for hardware implementation were analyzed through experiments performed with software used to simulate the hardware implementation of HAIL.

In each section, a new trade-off or optimization is described and justified. Experiments used to test the validity of the design decision(s) are presented with figures and analysis. The primary metrics used to measure the effects of design decisions are accuracy (the percentage of documents correctly classified) and latency (the average number of bytes that must be sampled before the algorithm's final decision on a document's language is reached). Other metrics relating to the efficiency of a hardware implementation will be presented as needed.

During the experiments that were performed, it was assumed that certain algorithmic parameters were independent from one another. A change in a parameter does not alter the effects of parameters from which it is independent. Specifically, the parameters adjusted in each of the following sections are assumed to be independent from parameters in other sections, but *not* independent from parameters within the same section. Following this assumption, each of the following sections adjust one or two parameters while keeping all others constant. In each subsequent section, the optimal parameters determined in previous sections are applied. Table 3.1 lists the parameters applied in subsequent sections.

Table 3.1: Parameters applied during algorithmic analysis

Section	**N-Gram Length**	**Address Bits**	**Memory Width**	**Trend Depth**	**Permanent Counters**
3.2.1	*Varied*	19	Unlimited	No trend	Unlimited
3.2.2	5	*Varied*	*Varied*	No trend	Unlimited
3.2.3	5	19	4	*Varied*	*Varied*
Final	5	19	4	3	16

All experiments presented in this section were performed on data in 34 languages. Details on the data set are shown in Section A.2. Unless otherwise noted, results presented in this section are the averages of 50 runs which each use a different randomly-selected training set of 20 KB of data from each language. Documents used in the experiments averaged 500 bytes in length.

3.2.1 N-Gram Size

Software-based language identification algorithms typically attempt to process data as accurately as possible within a "reasonable" amount of time. Unlike HAIL, such algorithms do not attempt to operate at multiple Gigabits per second. They frequently extract and analyze multiple lengths of n-grams within a document. In order for HAIL to analyze multiple lengths of n-grams, a system must either contain numerous parallel memory banks (which requires a fairly complex platform) or must perform multiple lookups into the same memory bank at each byte offset (which requires a large amount of time). HAIL makes use of only one n-gram length for language identification. When the system processes data, it extracts n-grams of a predefined length, hashes them into a memory address and looks them up in the system's memory.

Table 3.2: Letter Frequencies in English, French, German, and Spanish [58]

Letter	**English Frequency**	**French Frequency**	**German Frequency**	**Spanish Frequency**
A	8.151%	8.147%	6.506%	12.529%
E	13.105%	17.564%	16.693%	13.676%
M	2.536%	2.990%	3.005%	3.150%
N	7.098%	7.322%	9.905%	6.712%
R	6.832%	6.291%	6.539%	6.873%
S	6.101%	8.013%	6.754%	7.98%
X	0.166%	0.350%	0.022%	0.221%

The choice of n-gram length can drastically impact the system's performance. Shorter n-grams typically appear frequently in multiple languages. Consider Table 3.2 which shows the frequency of occurrence for seven letters (n-grams of length one) in English, French, German, and Spanish. While some letters are useful for distinguishing these languages from one another, many others are not. For instance, the frequency of the letter *A* could be used to distinguish German from Spanish, English, and French. However, the frequency of occurrence of the letter *A* in English and French is nearly identical and not useful for distinguishing English documents from French documents.

The usefulness of single-byte n-grams grows when comparing unrelated languages, especially languages that are represented with different character encodings. As will be shown below, using these n-grams is much more accurate than randomly guessing the language; however, there are other lengths of n-grams that achieve much higher accuracy.

At the other extreme, long n-grams are also relatively inaccurate. As the n-gram length increases, the number of possible n-grams increases exponentially. In addition to this, the probability of occurrence for each individual n-gram decreases. Two consequences ensue. First, n-grams encountered during normal operation are less likely to have been seen during the training of the system. Second, many languages contain short words that occur frequently and are very useful for identifying the language in question. In English, examples include *the*, *and*, *for*, *this*, and *that*. Long n-grams will contain these words, but such n-grams also contain surrounding bytes. There are a large number of n-grams that can be formed with the addition of these surrounding bytes and reduce the frequency of any individual n-gram occurring in a language.

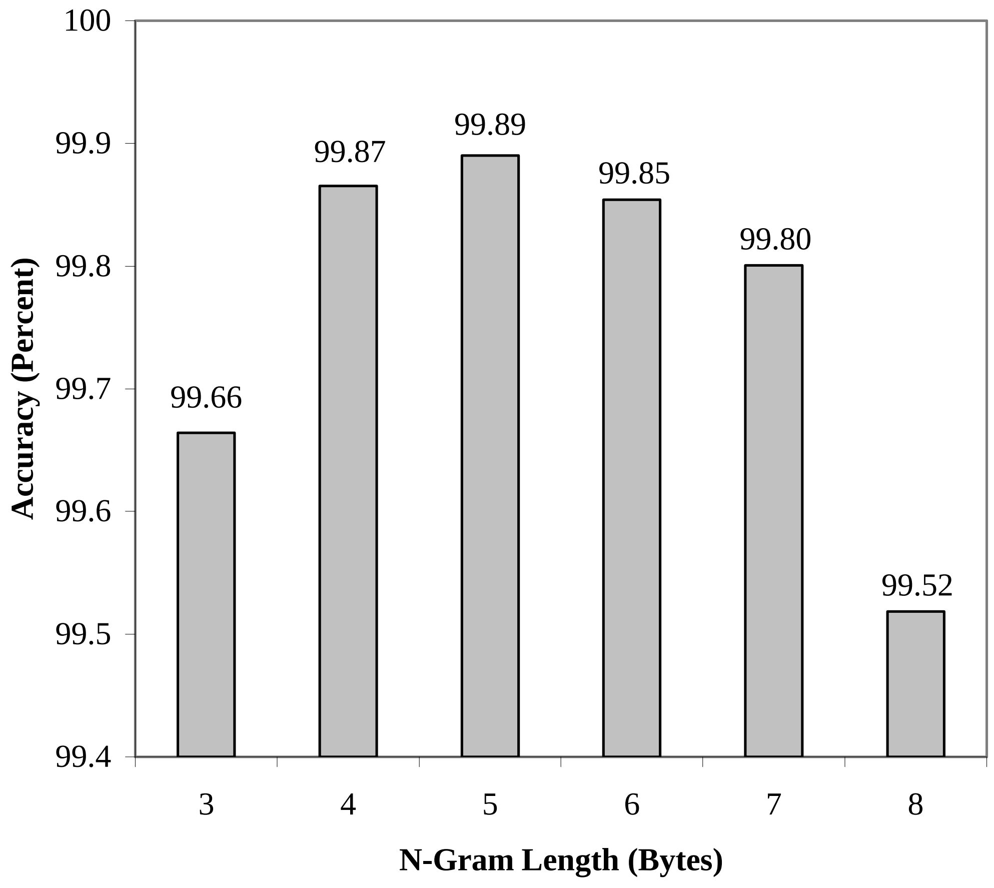


Figure 3.1: The effect of n-gram size on accuracy of language identification

Experiments were performed using HAIL simulation software to observe the effect of changing the n-gram length from one byte to ten bytes. Figures 3.1 and 3.2 present the most relevant portion of these results. Full results for the experiments are shown in Figures D.1 and D.2.

Figure 3.1 shows the change in accuracy as the n-gram size is altered from three to eight bytes. Accuracy jumps sharply between 3-grams and 4-grams, then increases slowly to a peak when 5-grams are used. The accuracy declines slowly when 6- and 7-grams are used, then plummets significantly when using 8-grams. This is consistent with prior analysis that short and long n-grams are inferior to those of a medium length.

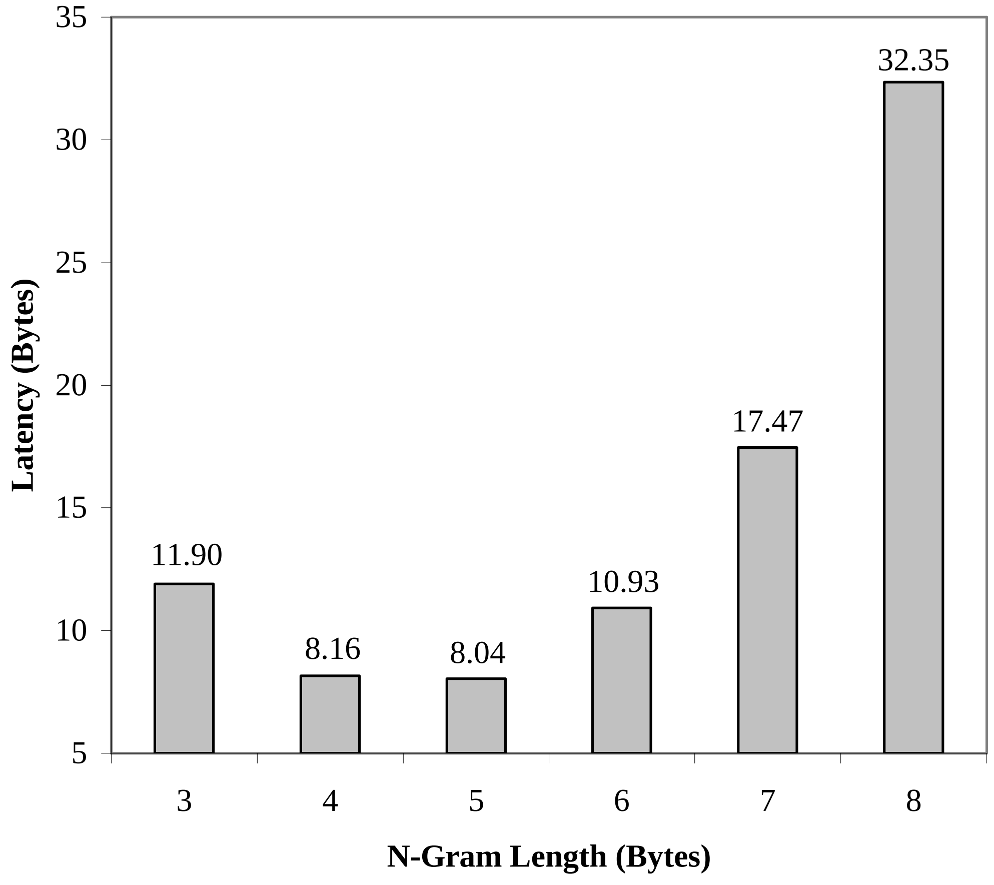


Figure 3.2: The effect of n-gram size on latency of language identification

Another measure of a design decision's effectiveness is latency, which was defined in Section 3.1.1. Figure 3.2 shows the effect of n-gram size on latency. The average latency, measured in bytes, is higher for large and small n-grams. This is to be expected; as stated earlier, when shorter n-grams are used, confusion can exist between similar languages. When longer n-grams are used, hash collisions and words not encountered during training increase and a good evaluation of the language is difficult to make.

Because of these reasons, it comes as no surprise that 5-grams, in addition to resulting in the highest accuracy, also result in the lowest latency. 4-grams have a marginally higher latency, although the differences in latency between other lengths of n-grams are more pronounced.

It should be pointed out that the differences in accuracy and latency between 4-grams and 5-grams are so small that they are probably insignificant. While it is likely that either length could be used, available data indicates that 5-grams offer a slight advantage; therefore, they will be used in the remainder of experiments during this chapter.

3.2.2 Memory Configuration

The FPGA implementation of HAIL makes use of low-latency random access memory such as SRAM to store the languages associated with n-grams found during training. Many varieties of SRAM can be accessed every clock cycle, allowing for high throughput when processing on high-speed streaming data. However, SRAM has a significantly lower capacity than higher-latency memory technologies such as SDRAM. Due to the small memory space, hash collisions can become a common occurrence and steps must be taken to mitigate the effects of the collisions.

Many software-based hash mechanisms resolve hash collisions through techniques such as *chaining* or *open addressing* [28]. In a chaining system, each location in a hash table references a linked list of all items that hash to the location in question. Collisions are resolved by stepping through the linked list until the relevant item is found or the end of the list is reached. In an open addressing system, collisions are resolved by searching through alternate locations in the hash table until the relevant item is found or an unused memory location is encountered. The alternate locations are determined by a probing sequence, which dictates the next address to search when a collision occurs.

A drawback of chaining is that memory size is not fixed; typically, memory for linked lists is dynamically allocated and can thusly be indefinitely large. This becomes a problem when using relatively small memory elements such as SRAM. A drawback of both chaining and probing is that the worst-case lookup time is $O(n)$, where n is the number of items stored in the hash table. When performing rapid processing on streaming data, it is desirable to perform tasks in $O(1)$, or a constant amount of time.

Pursuant to this constraint, HAIL does not use these collision resolution mechanisms in the hash table used to store the languages of n-grams. All n-gram language lookups are implemented by performing a single hash function over the n-gram and using the resulting hash to reference memory. Without a collision resolution mechanism, other measures must be taken to minimize the accuracy loss caused by hash collisions.

Another drawback of the reconfigurable hardware implementation is memory width. Experiments in the previous section assumed that memory was infinitely wide. While this assumption can be made in software by using a wide two-dimensional array to store all languages associated with a hashed n-gram, such an assumption cannot be made in an efficient hardware implementation. For purposes of throughput, each hashed n-gram can only be looked up a limited number of times in memory. Since physical memory banks are limited in width, only a limited amount number of language identifiers can be stored at each memory location.

For purposes of implementing HAIL on a wide variety of platforms, only a small number of memory banks are assumed to be available. Memory width could effectively be indefinite by incorporating a large number of parallel memory banks; however, this can be complex and limit HAIL's implementation to highly specialized platforms.

Experiments in the remainder of this section assume strict constraints on the amount of memory available to an implementation of HAIL in reconfigurable hardware. It is assumed that time and resources are limited to the extent that each hashed n-gram can be looked up only once in a bank of memory. The bank of memory simulated in the experiments is the SRAM available on the Field-Programmable Port eXtender (FPX), the platform on which HAIL is implemented in Chapter 4. This 2.25 Megabyte SRAM bank contains 19 address and 36 data lines, resulting in 524,288 memory locations that are 36 bits wide.

If language identifiers are 9 bits in size (allowing for 511 possible language/character encoding pairs while leaving zero to indicate "no language"), the width of SRAM allows four identifiers to be stored at a single memory location. An experiment was performed to establish which of the following three configurations would produce the highest accuracy and lowest latency:

1. Hash n-grams to a 19-bit address, and read four language identifiers from memory that correlate to the hash.
2. Hash n-grams to a 20-bit address. Use 19 bits to read up to four language identifiers from memory, and use the remaining bit to select two language identifiers that correlate to the hash.
3. Hash n-grams to a 21-bit address. Use 19 bits to read up to four language identifiers from memory, and use the remaining 2 bits to select the identifier that correlates to the hash.

The advantage of configuration 1 is increased tolerance of hash collisions and n-grams common to multiple languages. This configuration can accommodate up to four situations in which different n-grams used in different languages hash to the same value, or situations in which a particular n-gram is common in more than one language.

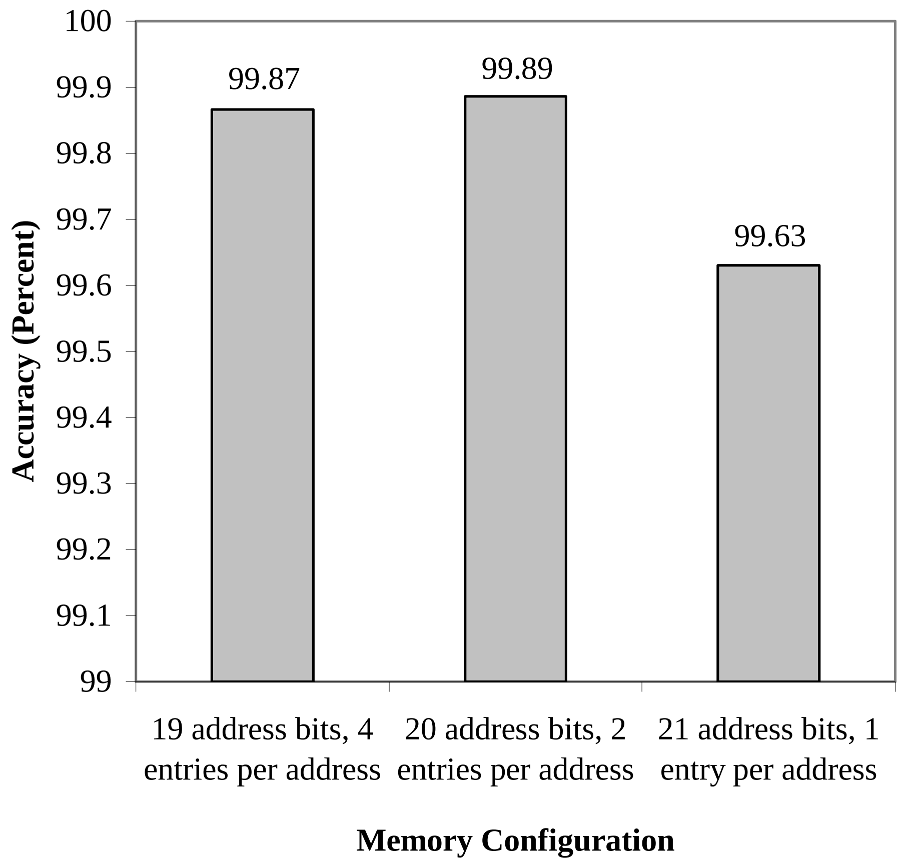


Figure 3.3: The effect of memory configuration on accuracy of language identification

The advantage of configuration 3 is a reduction in the total number of hash collisions, as it provides four times as many addresses as configuration 1. However, this configuration cannot resolve hash collisions or n-grams common to multiple languages; only the language most strongly associated with a hashed n-gram can be stored in a memory location.

Configuration 2 is a compromise, as it can resolve two collisions or n-grams common to multiple languages while providing twice as many addresses as configuration 1.

In accordance with Table 3.1, the experiments to investigate the effect of these configurations used an n-gram length of five bytes, no trend depth and an unlimited amount of permanent counters (the latter two terms are explained in Section 3.2.3).

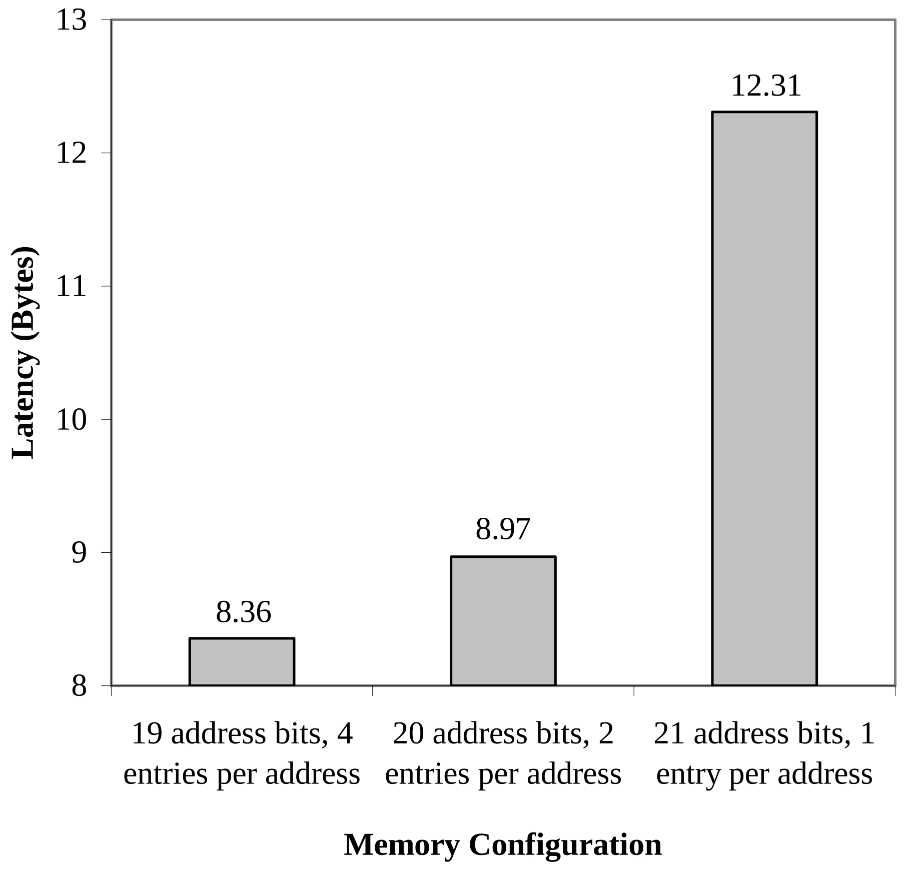


Figure 3.4: The effect of memory configuration on latency of language identification

Figure 3.3 shows the effect of memory configuration on the accuracy of language identification. As seen in the chart, the greatest accuracy is achieved using configuration 2. However, this result is so close to that of configuration 1 that differences between the two are likely insignificant. Configuration 3, on the other hand, is significantly inferior in accuracy.

Figure 3.4 shows the effect of memory configuration on latency. Lowest latency is achieved with configuration 1, while the highest latency is achieved with configuration 3. The difference between configurations 1 and 2 is fairly small. However, this difference is much more pronounced than that shown between the two configurations in Figure 3.3.

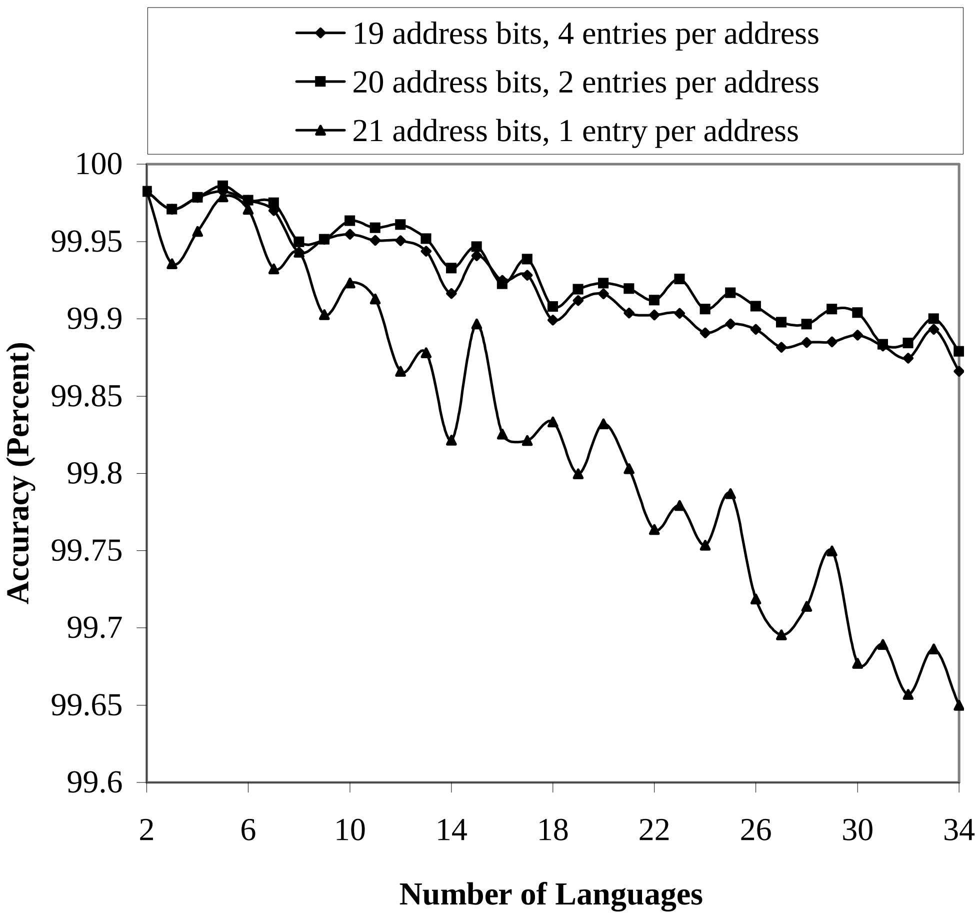


Figure 3.5: The effect of number of languages and memory configuration on accuracy of language identification

The relatively poor performance of configuration 3 makes the case that, for this particular application, added tolerance for hash collisions is more important than a somewhat larger memory space. If more languages were introduced to the system, the amount of training data to be stored in the hash table would increase. This would naturally lead to more hash collisions. The similar performances of configurations 1 and 2 may be caused because hash collisions are not prevalent enough to warrant tolerance for more than two collisions to a particular location.

Figure 3.5 partly verifies this speculation. Configuration 3 mirrors the other two configurations for approximately eight languages or fewer. At this point, its accuracy falls off significantly from the previous two configurations. While the amount of data available does not show one of the other two configurations falling away from the other, it is not unreasonable to speculate that such an occurrence may happen when more languages are added.

The experiment that generated Figure 3.5 involved 33 experimental runs. Each run differed in the number of languages used, a value that was varied from two to 34. Languages in each run were chosen at random and training was performed. The HAIL software was then run three times, each time using a different memory configuration. The process was repeated 100 times for each differing number of languages, resulting in 9,900 total runs of the HAIL software. Results from each combination of trend depth and number of languages were averaged, producing 99 data points; three different memory configurations, each analyzed from two to 34 languages.

Based on results obtained in this section and speculation based on Figure 3.5, it was assumed that configuration 1 holds an advantage over other configurations. Thus, configuration 1 will be applied during the remainder of this chapter.

3.2.3 Counting and Trend Establishment

A language identification system could require one counter for each language recognized by the system. If such a system recognized several hundred languages, it would require several hundred counters. In order to sort or to find the largest value among these counters, a number of comparisons no less than the total number of counters must be performed.

In a hardware implementation, a large number of comparisons can be performed in parallel; however, this can consume a significant portion of resources on an FPGA. Alternatively, a hardware implementation can perform the required comparisons serially. This alternative is inappropriate for hardware intended to operate as quickly as possible on streaming data, since comparing hundreds of counters can require several hundred to several thousand clock cycles.

Another difficulty that arises from maintaining hundreds of counters is the amount of context information that must be associated with each document. When performing language identification on a medium such as streaming network data, it can be expected that packets containing data from multiple documents will be interleaved with one another. In order to analyze entire documents, information on the state of each document must be maintained when processing the packets of other documents. Maintaining several hundred counters implies that several hundred counters must be stored when processing other documents. Not only does this require a large amount of memory, but the time spent reading from and writing to memory can be significant.

An optimization has been created that drastically reduces the number of counters needed to track languages. It was observed that only a handful of counters are needed to classify a document into one of many languages.

This process is carried out by a *trend register*, which is used to track successive n-grams that represent the same language. As hashed n-grams are looked up in memory and the corresponding language identifiers are output, the trend register counts consecutive appearances of the same language. Once this counter reaches a particular threshold (referred to as the *trend depth*), the language in question is granted access to one of several "permanent" counters. From that point onward, all occurrences of that language will increment the corresponding permanent counter. The relatively small number of permanent counters drastically reduces the amount of hardware resources required.

It should be noted that the addition of this trend register system implies that HAIL is not a purely naive Bayes classifier, which was described in Section 2.3.3. Since the trend register is based on the assumption that n-grams closely associated with a given language tend to occur close together, it adds an element to HAIL that is not naive Bayesian in nature. However, once a language creates a sufficiently large trend, all future occurrences of the language in question are treated in the fashion of a naive Bayes classifier.

The trend depth is a design optimization that was determined experimentally. If the trend register threshold is set too low, the permanent counters can be affected by noise or the results of hash collisions. If the threshold is set too high, the true language of a document may not ever be granted access to a permanent counter.

The number of permanent counters is another experimentally-determined design optimization. If too few permanent counters are used and and large amounts of noise and/or hash collisions are present, the true language of a document may not gain access to a permanent counter. If too many permanent counters are used, accuracy and latency do not suffer; however, the sort time and resource reductions granted by the trend register are depreciated.

Due to the close relationship between trend depth and the number of permanent counters, it was not assumed that these two parameters are independent. In fact, the number of permanent counters required is dependent on the trend depth; as the trend depth is increased, the threshold for granting access to a permanent counter increases and the number of required permanent counters decreases.

The nature of the trend register system requires a different experimental configuration from that of the previous two sections. The benefit of a trend register system is not fully realized without a large number of languages. However, the corpus of data available for the experiments performed in this chapter contains only 34 languages. While a certain number of permanent counters may be suitable for 34 languages and a particular trend depth, that number of permanent counters may be insufficient for 511 languages. Therefore, the change in the number of permanent counters required as the number of languages is altered must be recorded and used to predict the number of permanent counters required at 511 languages.

To achieve this, 33 experimental runs were performed. Each run differed in the number of languages used, a value that was varied from two to 34. Languages in each run were chosen at random and training was performed. The HAIL software was then run five times, each time using a different trend depth (which was varied from one to five consecutive n-grams). The process was repeated 100 times for each differing number of languages, resulting in 16,500 total runs of the HAIL software. Results from each combination of trend depth and number of languages were averaged, producing 165 data points: five different trend depths, each analyzed from two to 34 languages.

Once these data points were obtained, linear regression (see Section 2.3.1) was used to fit a line to the 33 data points representing each trend depth. The equation for each line was used to predict data points for each trend depth when the number of languages is increased to 511.

A new metric is presented in Figure 3.6. It is an estimate of the number of permanent counters required for a particular configuration when 511 languages are used. The metric is obtained by observing the permanent counters and noting when, on average, the actual language of a document is granted access to a permanent counter. A value of one indicates that the actual language is, on average, the first to be granted a permanent counter within a document; likewise, a value of five indicates that the actual language is typically the fifth to be granted access to a permanent counter. The metric can therefore provide a good estimate of the number of permanent counters required for a particular configuration.

Figure 3.6 shows a clear difference between different configurations. While a trend depth of one n-gram (effectively no trend register) would on average require over 18

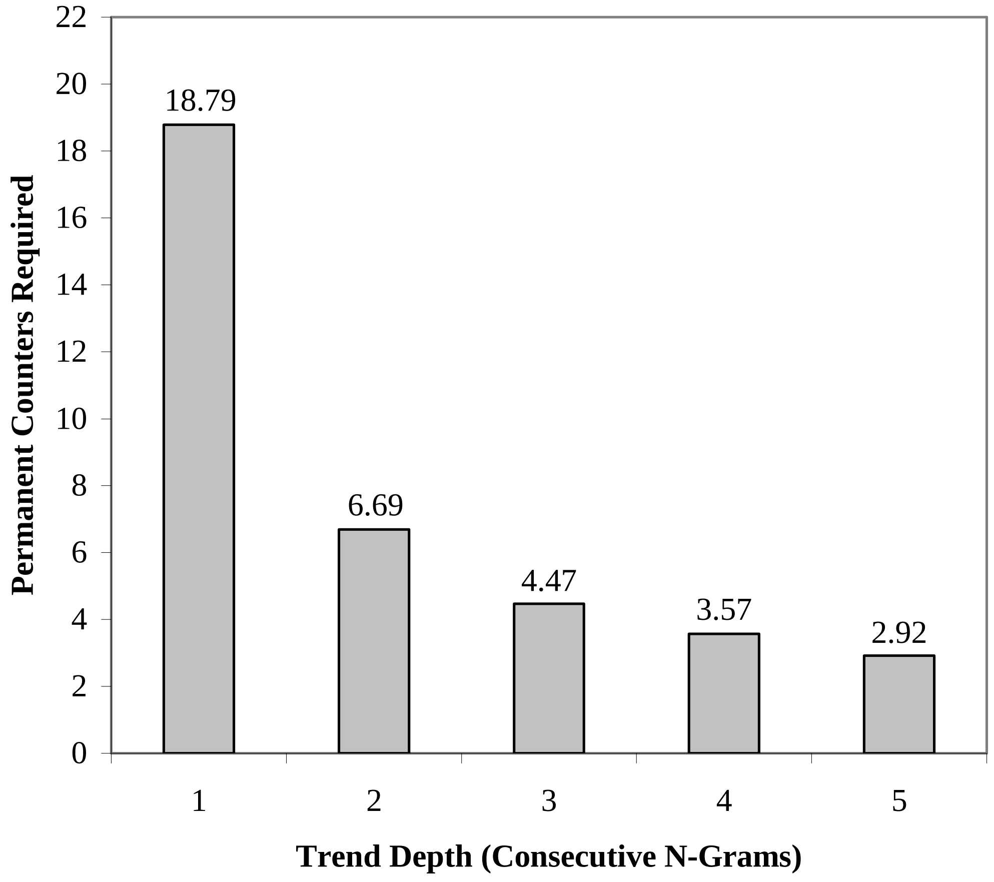


Figure 3.6: The effect of trend depth on permanent counters required, estimated at 511 languages

permanent counters, this number drops off sharply as the trend depth is increased. A trend depth of five consecutive n-grams requires under four permanent counters.

While a particular trend depth may lend itself to a small number of permanent counters, lower resource utilization is fairly meaningless if accuracy and latency are significantly compromised. Therefore, linear regression was also used to predict these two metrics for different trend depths when 511 languages are used.

Figure 3.7 shows the accuracy of each trend depth from one to five, based on the estimation of data points at 511 languages. Note that a trend depth of five, while requiring a very small number of permanent counters in Figure 3.6, is less accurate than shorter trend depths by a relatively large margin.

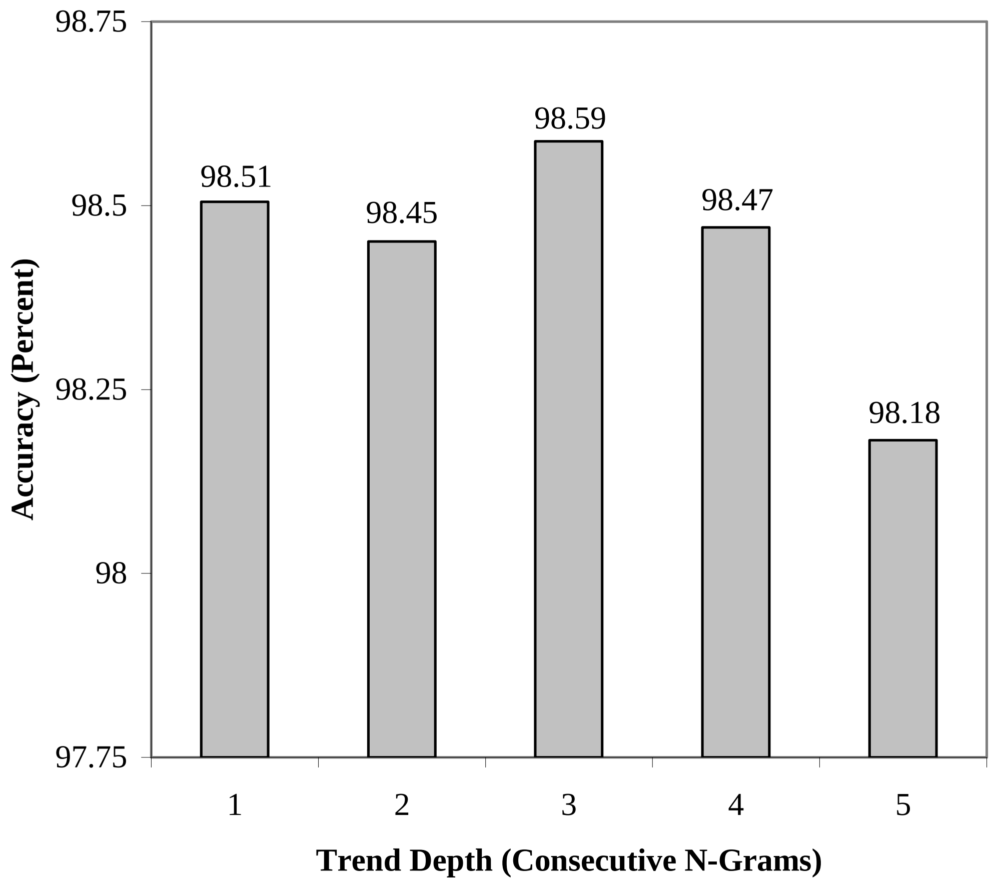


Figure 3.7: The effect of trend depth on accuracy of language identification, estimated at 511 languages

As seen in the figure, a configuration using a trend depth of three consecutive n-grams provides the highest accuracy. However, like several other charts throughout this chapter, this result is very close to other observations and is not significantly higher than trend depths of one, two or four consecutive n-grams.

Figure 3.8 illustrates the latency, projected at 511 languages, as the trend depth is altered from one to five consecutive n-grams. The latency follows a general upward trend, with the exception of a trend depth of one; this is due to the threshold being set so low that many languages are granted access to permanent counters, making it simpler for other languages to gain leads in their counts. For larger trend depths, latency increases because of the higher threshold needed for any language to be granted a permanent counter.

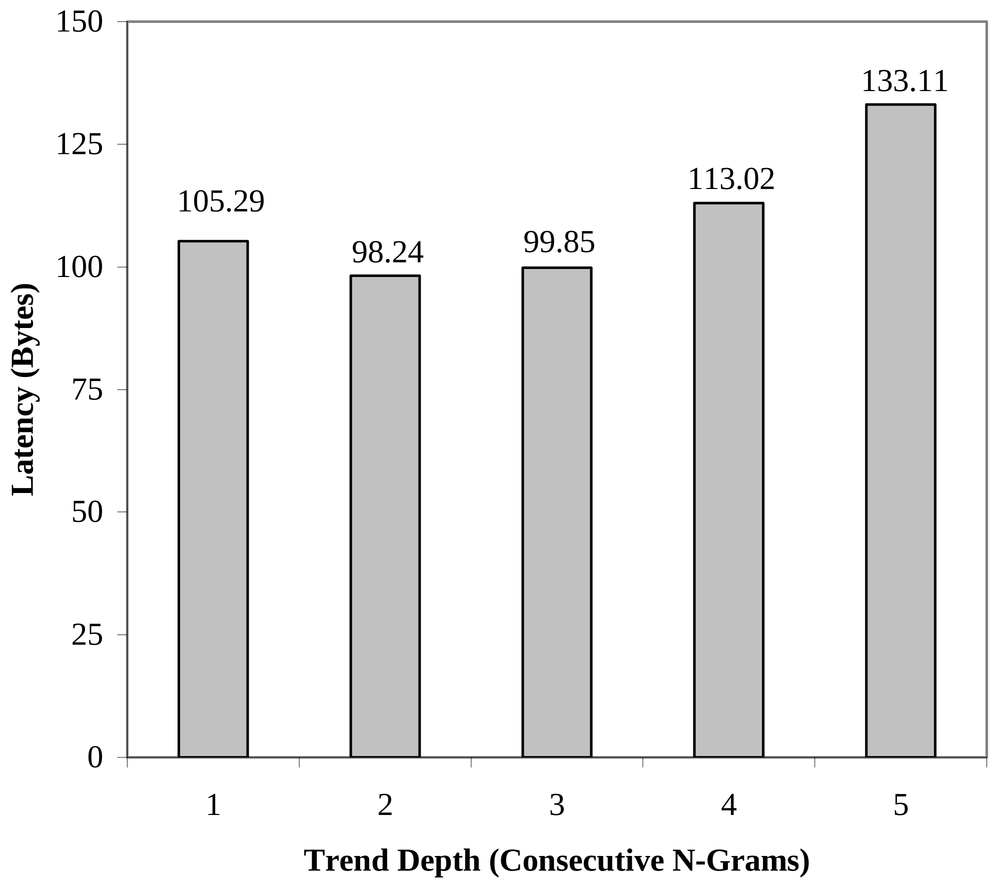


Figure 3.8: The effect of different HAIL configurations on latency of language identification, estimated at 511 languages

Trend depths of two and three consecutive n-grams are clearly superior in terms of latency. Based on the results, a depth of two maintains a slight advantage. Again, this is a trivial difference and the respective latencies are obviously very similar.

Choosing an optimal trend depth based on these results is not an entirely clear decision. However, it is worth noting that a trend depth of three was, in the experiments, the most accurate; it required under two permanent counters more than a trend depth of five (the optimal depth in terms of permanent counters); and it was virtually tied for the lowest latency. Based on these observations, a trend depth of three will be used in subsequent sections of this paper.

Choosing a number of permanent counters to use is also not a simple conclusion that can be derived from these results. While Figure 3.6 showed that five permanent counters (rounded up from 4.47) is sufficient in the average case, it is obvious that some documents will require more counters. In subsequent sections, 16 permanent counters will be used. This value is significantly larger than the average of 4.47 and can be expected to accommodate a wide variety of documents. Furthermore, comparing the counters to determine the largest can be done with only fifteen comparisons. This can be performed with only four levels of comparators by comparing adjacent counters and propagating the larger counter to the next level of comparators until the largest is found. Conceptually, this is very similar to a single-elimination tournament bracket.

Notes on Linear Regression

Linear regression, applied throughout this section to estimate data points when 511 languages are used, works best when a series of data points is roughly linear. When a set of data follows a nonlinear distribution, linear regression cannot be expected to provide accurate results.

Figures D.3, D.4 and D.5 show the changes in permanent counters required, accuracy and latency as the number of languages is adjusted from two to 34. As seen in Figure D.3, the number of permanent counters required appears to grow in a less-than-linear fashion. This is evidenced by the thick straight lines placed on the chart; a line constructed to closely fit the first half of the data points moves above the second half of data points. (This does not appear true for trend depths of four and five, although the curves may be only locally linear.) If these less-than-linear trends continue through 511 languages, the estimates shown in Figure 3.6 are not entirely accurate. Rather, they would be upper bounds on the average number of permanent counters required.

Figure D.5 is very similar. The curves, other than that for a trend depth of one, appear to be less-than-linear. This would imply that latency figures in Figure **??** are, again, upper bounds. Figure D.4 is an exception to the previous two graphs. The curves appear to oscillate evenly in a small range about the straight line plotted on the graph. If this linear tendency holds, the results in Figure 3.7 can be taken as reasonably accurate.

3.3 Assignment of Memory Slots

Using the trend register system, problems arise when a trend depth of three consecutive n-grams is coupled with four language identifiers per memory location.[1] This requires a significant amount of parallel and sequential logic, such as:

- **Stage 1:** Compare each of the language identifiers exiting memory to the languages in each of the 16 permanent counters to see if the languages are already represented.
- **Stage 2:** Increment permanent counters corresponding to languages that are already represented. Send languages that are not currently represented to the trend register.
- **Stage 3:** Perform 32 comparisons of language identifiers between memory slots at different stages of the trend register.
- **Stage 4:** Perform 64 comparisons between the outputs of **Stage 3** to identify trends of three n-grams representing the same language.
- **Stage 5:** Identify and select the 0 to 4 languages from **Stage 4** that formed a trend.
- **Stage 6:** Compare each language from **Stage 5** to the languages in each of the 16 permanent counters to see if any counters are unused. Also check if any of these languages have become represented in the permanent counters during the time that has elapsed since **Stage 1** was performed.
- **Stage 7:** If space is available in the permanent counters and the languages have not become represented in the permanent counters during the time that has elapsed since **Stage 1** was performed, populate the permanent counters with the newly-discovered trends.

Figure 3.9 illustrates Stage 3 of the aforementioned process and demonstrates the number of comparisons that must be made between different languages within the trend register system. Language identifiers in the first and last stages of the trend register must be compared to four other language identifiers, while those in the second stage must be compared to eight other language identifiers.

[1]Each memory location in the SRAM device being simulated is 36 bits wide. Up to four 9-bit language identifiers are packed into every memory location. For clarity, a 9-bit section within a memory location will be referred to as an *memory slot*.

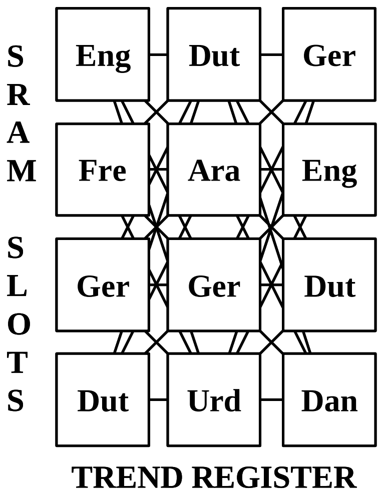


Figure 3.9: Searching for a trend when language identifiers can appear in any memory position and n-grams are processed one at a time

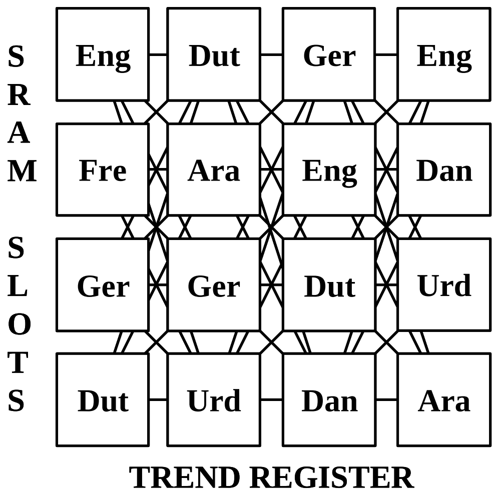


Figure 3.10: Searching for a trend when language identifiers can appear in any memory position and two n-grams are processed simultaneously

The figure does not illustrate the extra logic needed to shift each language identifier through different stages of the trend register as additional n-grams are processed. It also does not illustrate the comparisons in Stage 4 needed to identify complete trends (such as the trends formed by German and Dutch language identifiers in the figure) or the many comparisons to permanent counters carried out in Stages 1 and 6.

Within an FPGA, there are a limited number of logic components as well as a finite amount of interconnect available to transmit information between components. Performing the seven-stage process described above requires a significant amount of both logic and interconnect and in modern FPGAs would be infeasible to carry out in a small pipeline, much less a single clock cycle.

This problem is exacerbated if more than one n-gram is processed in any given clock cycle (see Section 4.3.2). Figure 3.10 illustrates a scenario in which two n-grams are processed simultaneously. Since the language identifiers associated with each n-gram must be compared with the preceding two sets of language identifiers, the number of comparisons required in the aforementioned process is increased significantly.

3.3.1 Slot Mapping

Figure 3.11 illustrates a situation in which language identifiers, if they are associated with a hash value, always appear in the same memory slot. As shown in the figure, the *Eng* (English) language identifier only appears in the first memory slot, the *Dut* (Dutch) language identifier only appears in the fourth memory slot, and so forth. In this situation, searching for a trend requires only a comparison of language identifiers originating from the same memory slot.

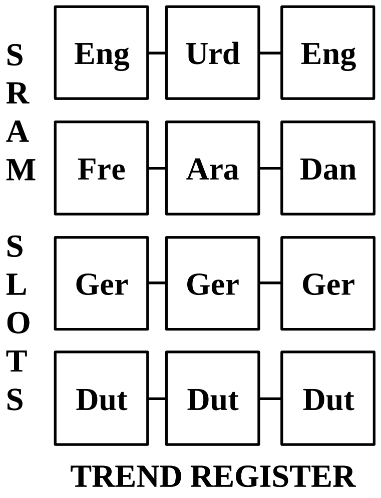


Figure 3.11: Searching for a trend when language identifiers can appear in only one memory position, with no data loss

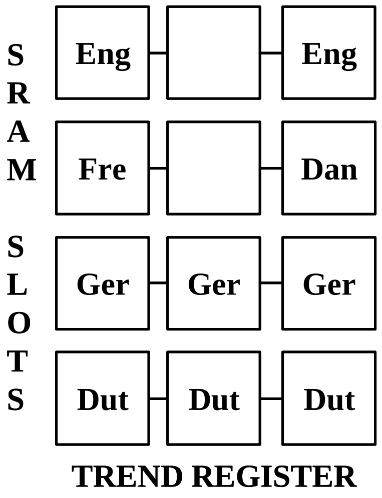


Figure 3.12: Searching for a trend when language identifiers can appear in only one memory position, and data is lost

To simplify the process of identifying trends, it was decided that a system such as this must be enacted; that is, whenever a language identifier appears in a memory location, it will *always* appear in the same memory slot. To this end, a mapping of languages to memory slots must be created. However, such a system will invariably cause some data loss. Figure 3.12 illustrates the same situation as Figure 3.11, except *Urd* (Urdu) and *Ara* (Arabic) were mapped to the same memory slot as *Ger* (German) and *Dut* (Dutch). In cases such as this, the language identifiers more closely associated with a particular hash value will be stored in the corresponding memory location, while those less closely associated will not appear.

Due to the inevitable data loss, the mapping of language identifiers to slots must be carefully created. For instance, the Spanish and Portuguese languages are closely related [62] and will likely be associated with many of the same hash values. If they were mapped to the same memory slot, a large amount of loss would occur. Thus, the mapping should work to minimize the amount of data loss.

Figure 3.13 shows a model of memory if a mapping algorithm were not applied. In this simplified example, overlap between any two languages can be calculated as the number of times that a particular language identifier appears whenever a second

6	**Danish**	**Swedish**		
5	**French**	**English**	**German**	**Swedish**
4	**Spanish**	**Italian**	**Russian**	
3	**Danish**	**Dutch**	**English**	
2	**German**	**Danish**	**Russian**	
1	**Italian**	**Spanish**	**French**	**English**
0	**English**	**German**	**Dutch**	**Russian**

Figure 3.13: Layout of memory before language slot mapping

language language identifier appears in the same memory location. The layout of memory in the figure creates three categories of language pairings:

1. **Language pairs which overlap twice:** Dutch and English, English and French, English and German, German and Russian, Italian and Spanish.

2. **Language pairs which overlap once:** Danish and Dutch, Danish and English, Danish and German, Danish and Russian, Danish and Swedish, Dutch and German, Dutch and Russian, English and Italian, English and Russian, English and Spanish, English and Swedish, French and German, French and Italian, French and Spanish, French and Swedish, German and Swedish, Italian and Russian, Russian and Spanish.

3. **Language pairs which never overlap:** Danish and French, Danish and Italian, Danish and Spanish, Dutch and French, Dutch and Italian, Dutch and Spanish, Dutch and Swedish, French and Russian, German and Italian, German and Spanish, Italian and Swedish, Russian and Swedish, Spanish and Swedish.

The optimal assignment of languages to memory slots would ensure that language pairs in categories 1 and 2 are **never** assigned to the same memory slot, while language pairs in category 3 are freely assigned to the same memory slots. If this is not possible, language pairs in category 2 could be assigned to the same memory slots, but as infrequently as possible. Language pairs in category 1 should be assigned to the same memory slot as a last resort.

6		**Swedish**	**Danish**	
5	**German**	**Swedish**	**French**	**English**
4	**Italian**	**Spanish**		**Russian**
3		**Dutch**	**Danish**	**English**
2	**German**		**Danish**	**Russian**
1	**Italian**	**Spanish**	**French**	**English**
0	**German**	**Dutch**		English

Figure 3.14: Layout of memory after language slot mapping

Figure 3.14 shows a possible mapping of languages to memory slots. As shown, most assignments (German and Italian, Dutch and Spanish, Dutch and Swedish, Spanish and Swedish, Danish and French) are chosen from category 3 presented above. However, there was no feasible position in which to place Russian without choosing at least one language pair from category 2. In Figure 3.14, Russian and English are assigned to the same memory slot, despite the fact that they both appeared in memory address 0 in Figure 3.13. In the figure, English was placed in this slot (note the slot containing the word *English* that is not written in boldfaced text). In practice, the decision as to which language is placed in a conflicting slot must be made based on the language more strongly associated with the particular memory address.

While a good assignment of languages to memory slots was found easily in the aforementioned example, the situation is more complex in practice. The SRAM device utilized in Chapter 4 contains 524,288 memory locations, and the current implementation of HAIL supports as many as 511 languages. Some overlap between every language is all but assured, and some data loss will occur. An exhaustive method to find the best assignment of languages to memory slots is not feasible.

The number of assignments of languages to memory slots can be calculated as Stirling numbers of the second kind [24]. The formula for a Stirling number of the second kind, which calculates the number of ways n elements can be divided into k sets, is given in Equation 3.1.

$$S(n,k) = \frac{1}{k!}\sum_{i=0}^{k-1}(-1)^i \binom{k}{i} (k-i)^n \tag{3.1}$$

For this particular problem, there can be multiple values of k. That is, the 511 possible languages can be split into one set (in which all of the languages are assigned to one memory slot) to as many as four sets. Thus, to calculate all possible sets, Equation 3.1 must be calculated four times, varying k from one to four. In practice, however, any feasible solution (for four or more languages) will achieve the least amount of overlap by spreading languages across all four memory slots.

If 34 languages (the number used in experiments throughout this chapter) are used, there are approximately $1.23 * 10^{19}$ ways to assign the languages to four memory slots. If 511 languages are used, there are approximately $1.87 * 10^{306}$ ways to assign the languages to memory slots. Even if one billion assignments of 34 languages to four memory slots could be examined per second, it would require approximately 390 years to evaluate all of them. This figure grows to $5.94 * 10^{289}$ years when 511 languages are used.

Therefore, if slot mapping is going to be performed, a non-exhaustive approach must be used.

3.3.2 Slot Clustering

A solution that makes use of data clustering (see Section 2.4) was used to create low-overlap memory slot mappings. The clustering process takes place during training and uses the principle of expectation maximization to arrive at an memory slot mapping that provides low overlap in a reasonable amount of time.

During the training process (outlined in Section 3.1.2), a two-dimensional array is built containing the frequency at which every hashed n-gram appears in each language. For every hashed n-gram, the average frequency is subtracted from each language's respective frequency for the n-gram. This serves to eliminate weak associations between languages and n-grams.

The overlap between all languages in this two-dimensional array is then calculated. Overlap between two languages is calculated as the sum of frequencies, in both languages, across all hashed n-grams where both languages have a presence. Figure 3.15

524,287	**English** .015	**French** .018	**Danish** .001	**Russian** .000
⋮				
5	**English** .007	**French** .029	**Danish** .028	**Russian** .002
4	**English** .000	**French** .000	**Danish** .013	**Russian** .025
3	**English** .000	**French** .004	**Danish** .003	**Russian** .009
2	**English** .022	**French** .001	**Danish** .008	**Russian** .000
1	**English** .001	**French** .000	**Danish** .000	**Russian** .012
0	**English** .010	**French** .032	**Danish** .000	**Russian** .000

English-French: .134	**French-Danish:** .092
English-Danish: .081	**French-Russian:** .044
English-Russian: .022	**Danish-Russian:** .080

Figure 3.15: Calculating overlap between languages

shows a brief example in which overlap is calculated among four languages. (The examples shown in Figures 3.13 and 3.14 presented a simplified version of this process.) To calculate the overlap between English and French, for example, the frequencies for English and French are summed in all locations where both frequencies are greater than zero. In this case, frequencies at memory locations 1, 3, and 4 are **not** added because one or both of the frequencies are zero.

Once overlap between all languages is computed, the clustering process begins. This is done by expectation maximization (see Section 2.4.1), a repeated process of "expectation" (evaluating the present state of the system) and "maximization" (changing the state of the system in order to move it closer to its optimal state).

Initially, languages are assigned at random to the memory slots. The total overlap among all slots is calculated by summing overlap between all languages in each slot, then summing the results from each slot. This is the "expectation" step as it is an evaluation of total overlap, which is the parameter to be optimized using expectation maximization. The lowest possible total overlap is preferable, as it will reduce the data loss when languages are assigned to memory slots.

This is followed by the "maximization" step. One language identifier moved at random to a different memory slot. The expectation step is performed again to determine if the change was acceptable. If not, the system is returned to its previous state; if the change was acceptable, the new state replaces the previous state. The cycle of expectation and maximization steps is repeated for a fixed number of iterations.

The acceptability of a change, as alluded to in the previous paragraph, is determined through the process of simulated annealing (see Section 2.4.2). While a standard expectation maximization process will always converge to the nearest local optimum by only allowing beneficial changes during the maximization step, simulated annealing allows changes during early stages of the process that move the system away from an optimal state. This is performed in order to provide a chance to bypass a local optimum and converge on a global optimum.

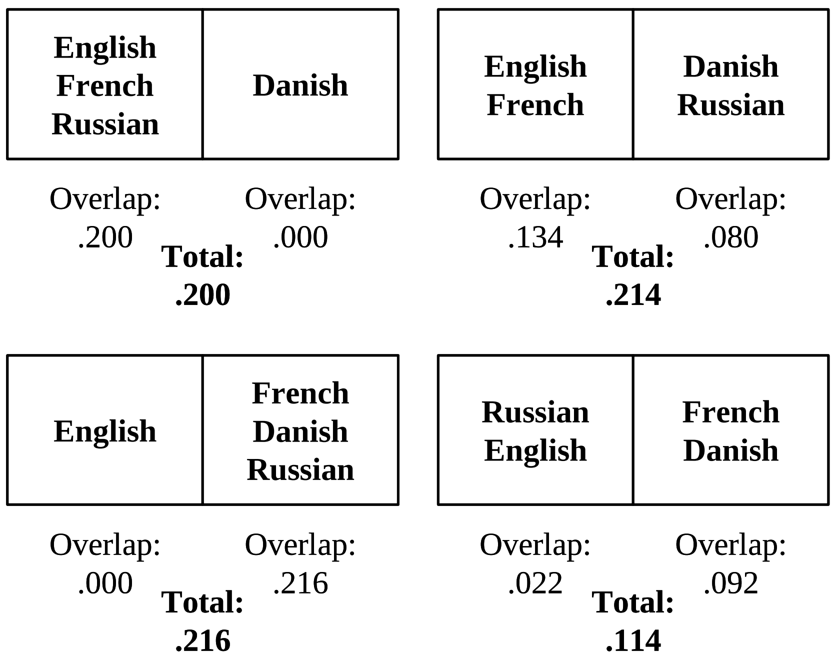


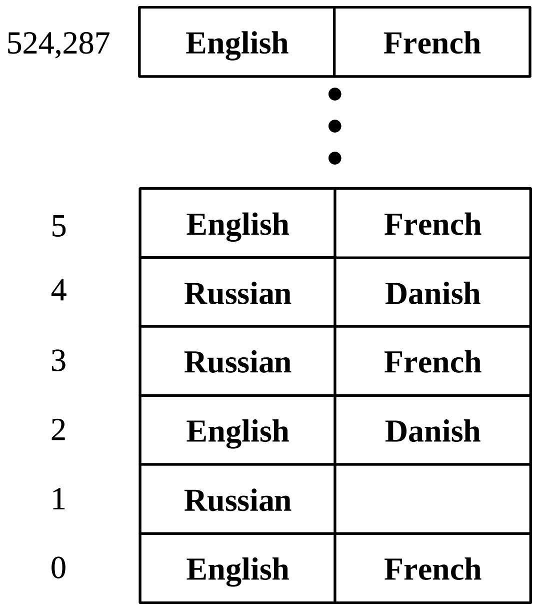
Figure 3.16: An example of the clustering process

Figure 3.16 illustrates a simple example of the clustering operations described above. The figure should be read left-to-right, then top-to-bottom. This example uses the language data from Figure 3.15. In the first step, the languages are randomly assigned to memory slots (the example uses only two slots) and the overlap between languages in each slot is calculated.

In the second step, Danish was chosen at random and moved to another memory slot. The presence of Danish and Russian in the same slot increases the overlap. However, such changes are allowed to some extent during the clustering process because of simulated annealing.

In the third step, French was randomly chosen and moved to the second memory slot. The three languages in the second slot provide the worst total overlap score yet. However, the overlap is only marginally worse than in the second step. Again, such a change can be allowed because of simulated annealing.

In the final step, Russian was randomly chosen to be moved to the first memory slot. This is the best total score achieved at this point and in this case, it is the global optimum. While this state could have been reached in only one step by moving French from its original assignment to the second memory slot, the random nature of expectation maximization does not guarantee a timely convergence towards a local optimum. Furthermore, the addition of simulated annealing does not guarantee that the global optimum will be found; it merely improves the chance.



524,287	**English**	**French**
	● ● ●	
5	**English**	**French**
4	**Russian**	**Danish**
3	**Russian**	**French**
2	**English**	**Danish**
1	**Russian**	
0	**English**	**French**

Figure 3.17: Loading memory after clustering is performed

Figure 3.17 provides a continuation of Figures 3.15 and 3.16. Once languages are assigned to particular memory slots, memory can be loaded with the appropriate language identifiers. At each memory location, a memory slot is populated with the language that has been assigned to the slot and is associated more strongly with the memory location than other languages in its slot.

Not only does this cause some data loss, but it also means that the languages stored at every memory location are not necessarily the languages that are associated most strongly with the memory location; rather, they are the languages *from their memory slot* that are most strongly associated with the memory location.

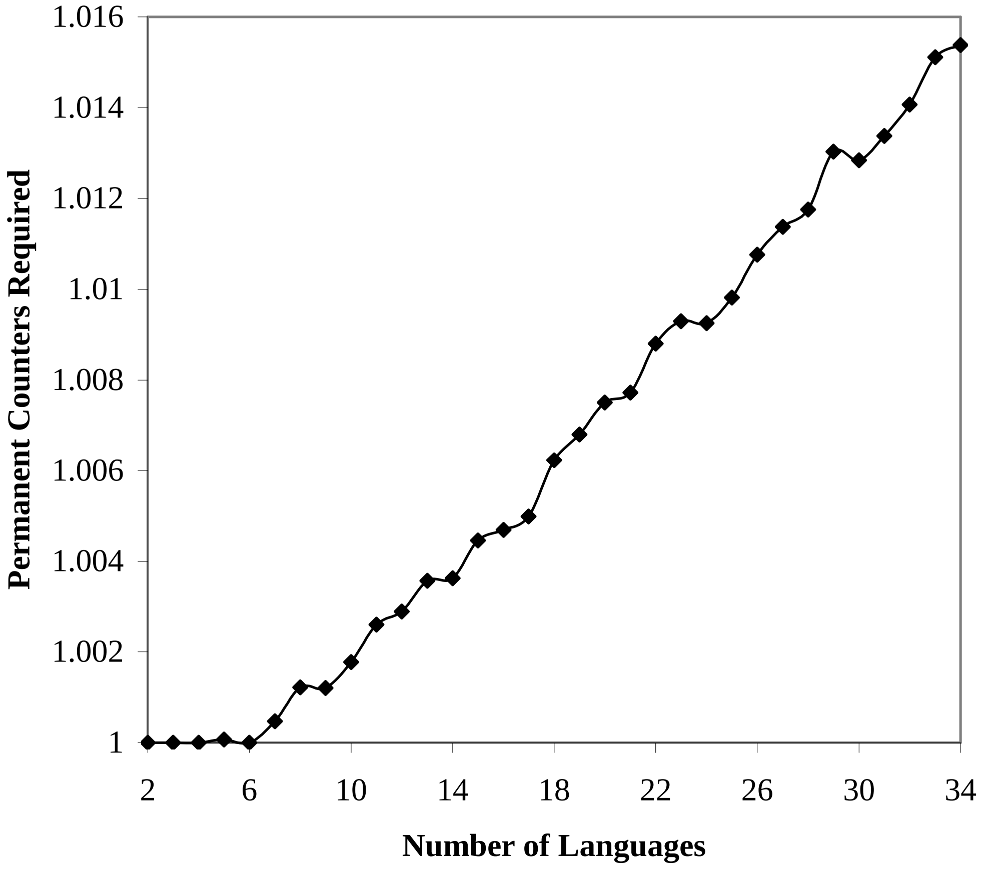


Figure 3.18: The number of permanent counters required as the number of languages is adjusted from two to 34

The nature of language slot clustering changes the definition of "permanent counters" discussed in Section 3.2.3. In practice, comparing four parallel trend registers to the languages associated with sixteen different counters is logic-intensive and not suitable for a high-speed implementation in FPGAs, most notably in older FPGA devices.

To reduce the amount of required logic further, the implementation using slot clustering still uses sixteen permanent counters, which was decided in Section 3.2.3. However, these counters are divided so that the trend register associated with each memory slot receives its own four "private" permanent counters. These counters can be populated only with language identifiers from the corresponding memory slot.

This still appears to provide enough counters. Figure 3.18 estimates the number of permanent counters required *per memory slot* as the number of languages is increased from two to 34. Once six or more languages are used, the number of permanent counters required increases, albeit very slowly. Through linear regression to 511 languages, it was estimated that, on average, 1.26 permanent counters are required per SRAM slot in order to ensure that the true language is granted access to a permanent counter. Maintaining four permanent counters per slot may be excessive for most documents, but it is still implemented easily and useful for accurately identifying outlier documents which may require more permanent counters.

The compromises presented throughout this section are of course less than optimal. However, after slot clustering is performed, a memory configuration with 19 address bits and up to four entries per address remains highly accurate and low in latency. Results for accuracy and latency are presented below.

3.3.3 Summary

Figure 3.19 compares the accuracy of different HAIL configurations presented in this chapter when 34 languages are used. The highest accuracy was achieved in Section 3.2.1, in which memory was unlimited in width. However, adjusting to the real-world constraint of a fixed memory width in Section 3.2.2 decreased accuracy noticeably.

Curiously, accuracy increased in Section 3.2.3. This may be due to unexpected benefits conferred by the trend register. For instance, without a trend register, words not encountered during training can hash to a populated memory location and cause inaccurate data to be counted. This can skew results. The trend register system guarantees that hash collisions such as these are not counted unless the same language identifier appears in several successive hashed n-grams.

Another benefit of the trend register is to eliminate the effect of words and phrases that have been adopted outside their native language. For instance, the phrase *coup d'état* is occasionally encountered in the English language despite being French in

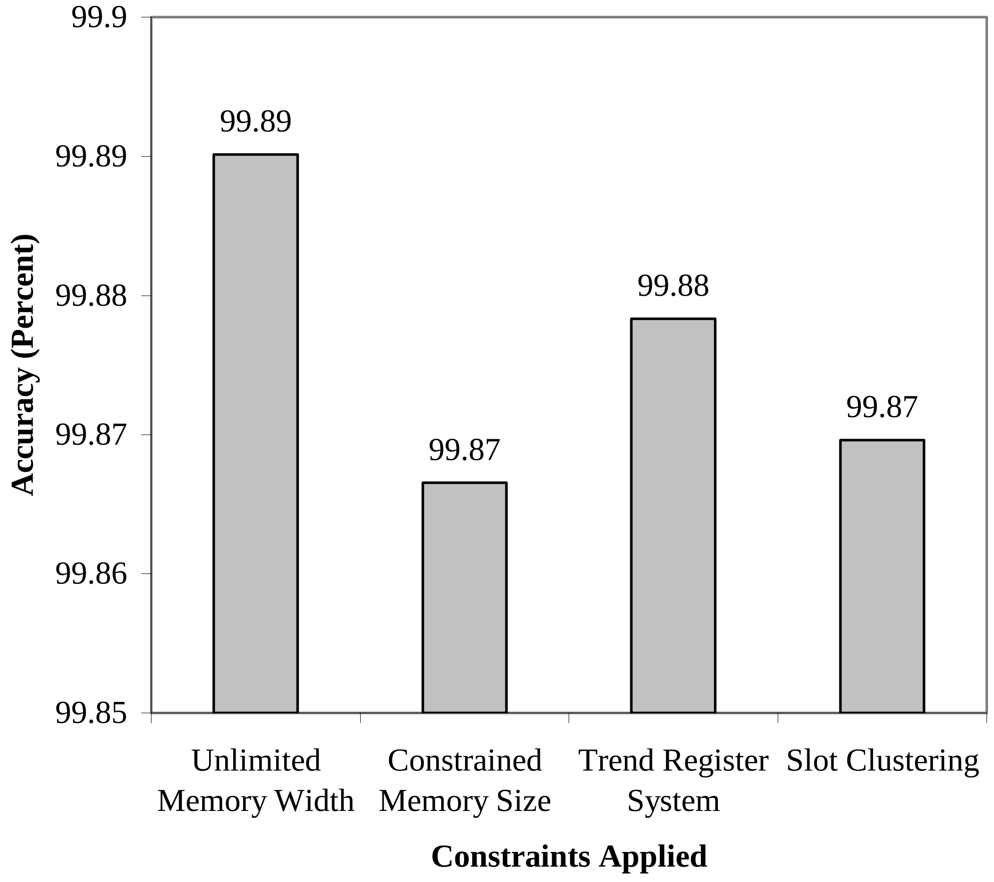


Figure 3.19: Accuracy of HAIL as additional constraints are applied

origin. A trend register helps to minimize the effect of words and phrases such as this, provided that they are used sparingly.

Adding clustering to the trend register system decreases accuracy by a small amount. This is to be expected, as once clustering is introduced, language identifiers at each memory location are only indicative of the best language identifier for each *slot*. However, it should be noted that using slot clustering remains more accurate than only storing one language identifier at each memory location (refer to Figure 3.3 in Section 3.2.2). This is significant because storing only one language identifier at each memory location is the only way to avoid the use of slot clustering in a trend register system. Therefore, the use of slot clustering is more accurate than the alternative.

Figure 3.20 compares the latency of the same configurations displayed in Figure 3.19. The latency is seen increasing steadily as more constraints are added. This is understandable; in figures provided throughout this chapter, a decrease in accuracy is

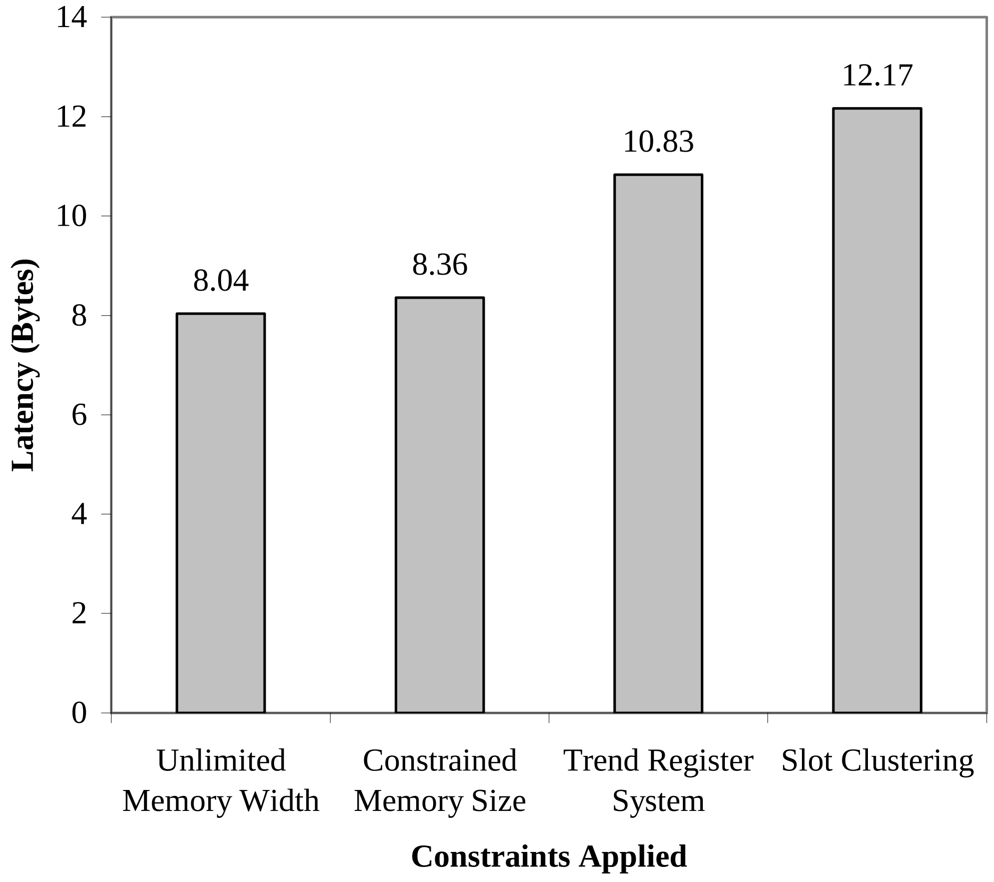


Figure 3.20: Latency of HAIL as additional constraints are applied

often (but not always) accompanied by an increase in latency. This is because a less accurate configuration is likely to experience more "confusion" and thus take longer to arrive at a conclusion when compared to a more accurate configuration.

The increase in latency between the second and third configurations (despite corresponding to an increase in accuracy in Figure 3.20) can be explained by the nature of the trend register system. This system makes it more difficult for a language to be granted a counter. A threshold of several consecutive n-grams representing the same language must be reached. Without a trend register, the correct language can be counted as soon as it is encountered.

Despite the various constraints applied, however, the latency increased by only a little over four bytes from the first configuration that was evaluated to the last.

Chapter 4

HAIL Implementation

An implementation of HAIL has been created for the Field-programmable Port eXtender (FPX) [48, 49], an open hardware platform developed at Washington University in St. Louis. The FPX is used extensively to process streaming network data. Several applications have been developed for use on the FPX, including firewalls [50], Bloom filters [37] and regular-expression matching engines [54] for deep packet inspection, and routers for enforcing quality-of-service (QOS) on high-speed networks [84].

As the name implies, the FPX was originally created to "extend" ports within a network switch by being placed between the switch's line card interface and switch fabric. In this configuration, the FPX can expand the utility of the switch by providing various functions.

While the implementation of HAIL has been implemented on this particular platform, HAIL is certainly not limited to use only on the FPX. The design is suitable for implementation on a wide variety of platforms.

4.1 The FPX Platform

The FPX platform consists of several main devices: The Reprogrammable Application Device (RAD), the Network Interface Device (NID), three banks of Zero-Bus Turnaround (ZBT) SRAM, two banks of SDRAM, and two Universal Test and Operations Physical Interface for ATM (UTOPIA) [2] network interfaces.

The RAD is a Xilinx Virtex XCV2000E-8 FPGA that serves as the primary component of the FPX. Application modules such as HAIL are loaded into the RAD in order to perform data processing. The RAD is connected to two 2.25 MB banks of

ZBT SRAM devices that provide low-latency memory access and two 64 MB banks of SDRAM that provide high-volume data storage.

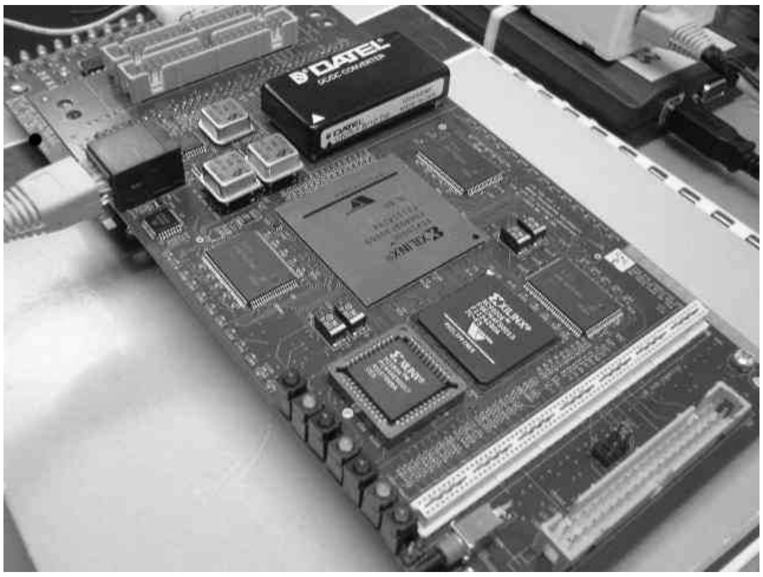


Figure 4.1: The Field-programmable Port eXtender (FPX)

The NID is a second FPGA that routes data between the RAD and the FPX's two external interfaces. It also programs the RAD's FPGA bitfile. The NID is a Xilinx Virtex XCV600E FPGA. The NID is connected to a bank of ZBT SRAM for buffering the bitfile that programs the RAD.

Data travels between the RAD, NID, and UTOPIA interfaces in Asynchronous Transfer Mode (ATM) cells [56]. ATM defines 53-byte *cells* as a basic data encapsulation unit, and can be used to transmit IP packets [47, 41]. An ATM cell contains a 5-byte header and 48 bytes of payload data. The header includes a Virtual Path Interface (VPI) and Virtual Circuit Interface (VCI) to route data between destinations. These two pieces of header information are used by the NID to route data.

The function of the NID hardware is remotely controlled over a network with a program called the Networked Configurable Hardware Administrator for Reconfiguration and Governing via End-systems (NCHARGE) [71]. NCHARGE sends commands that configure VPI and VCI routing within the NID, program the RAD, and check the status of the system.

A photograph of an FPX is shown in Figure 4.1. The RAD is the large device in the middle of the board, while the NID is directly below and to the right of it. Two of the SRAMs are on either side of the RAD, while the third is adjacent to the NID. The two SDRAM banks are attached to the bottom of the card.

Multiple FPX cards can be stacked, as shown in Figure 4.2, and configured so that data is processed by each FPX platform and transmitted to the next card in the stack. A stacked configuration of FPX platforms can be used to implement systems containing multiple modules. Details of the components within this stacked configuration are discussed in Section 4.2.

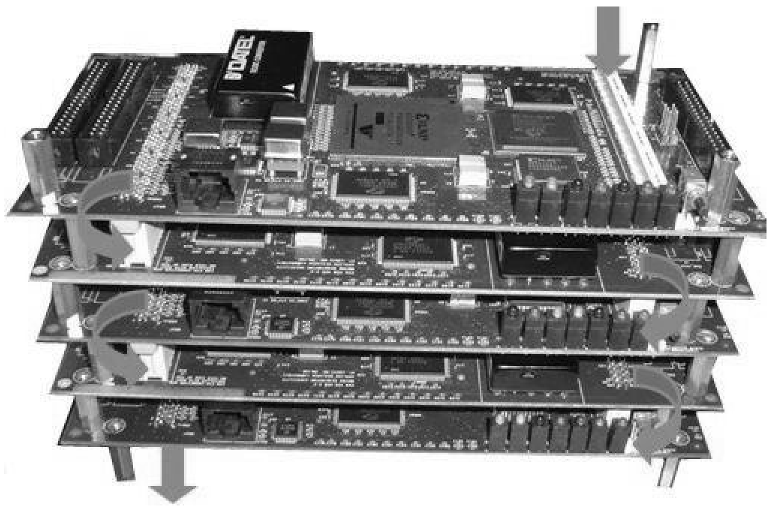


Figure 4.2: Stacked FPX cards performing multi-card processing

4.2 VHDL Infrastructure

Several infrastructure components have been created for use within the RAD. Each component has been created using the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). The relevant components include a wrapper for TCP flow processing and a communication wrapper.

4.2.1 TCPLite Wrapper

Studies show that 85% of all packets on the Internet [68] make use of the Transmission Control Protocol (TCP) [57]. The protocol is used for the reliable transmission of data across a network link. It contains mechanisms for reliable re-transmission of lost packets and for reordering out-of-order packets. An ordered stream of packets transferred via TCP is referred to as a *flow*. A flow can be uniquely identified by a combination of the packets' source IP address, destination IP address, source port, and destination port.

A TCP processor was developed for use in the FPX platform [67]. This circuit is implemented on a dedicated FPX card and provides ordered delivery of TCP packets within a flow. It bundles TCP packet data together with flow state information and transmits the package to a second FPX card, which implements an application such as HAIL. This application card uses a module called the TCPLite wrapper to separate flow state information from packet data and provide both pieces of data to the application circuit.

The flow state information provided to the application circuit includes a flow identifier. The flow identifier is created from a hash of a TCP packet's source and destination addresses and ports, and is used to uniquely identify a flow. Application circuits can use this flow identifier as an index to memory that stores flow-specific state information.

4.2.2 Communication Wrapper

A module called the *communication wrapper* was developed to simplify communication between stacked FPX modules, as well as between FPX modules and remote software hosts [5]. This component groups related ATM cells together and provides a stream of data and control signals to the application circuit that is free of ATM cell headers.

For communication with external software hosts, the communication wrapper works in conjunction with a separate FPX card which implements a circuit called the reporting module. When packets are received from hosts, the reporting module uses the packet's destination port to determine which FPX card should receive the packet data. It then strips off the packet headers and sends only the payload portion to the recipient card using one or more ATM cells.

Likewise, an FPX card can send data to a host via the reporting module. It transmits the data payload through the communication wrapper to the reporting module, which uses the ATM cell headers to determine which FPX card sent the data. It then prepends a packet header to the payload and sends it to the appropriate remote host.

4.3 Architecture

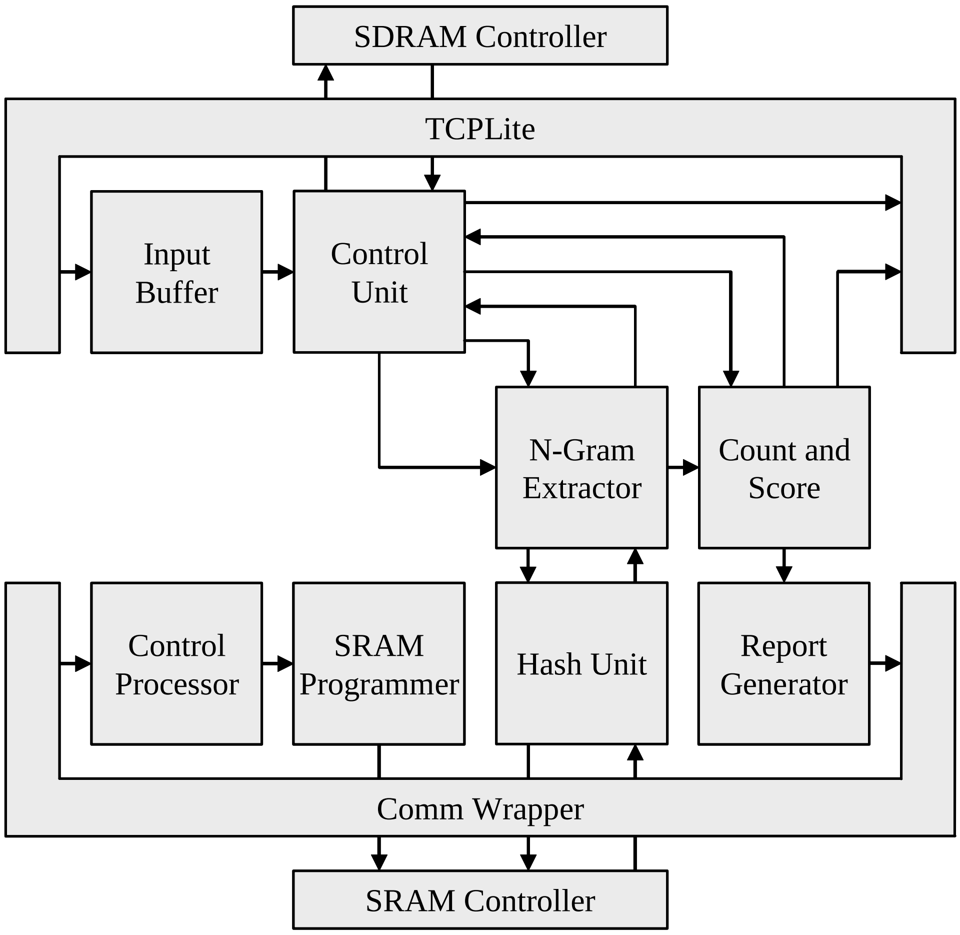


Figure 4.3: Architecture of the HAIL implementation for the FPX platform

The implementation of HAIL on the FPX platform is based upon the algorithm outlined and evaluated in Chapter 3. The circuit makes use of the TCPLite and communication wrappers outlined in Section 4.2. The use of these wrappers is important for abstracting the underlying protocols used to transport data on a network and between FPX modules. The use of these two components requires a second FPX to perform TCP processing and a third FPX to communicate with control software.

The two SRAM banks present on the FPX are used to store hash tables containing the languages best associated with each hashed n-gram. Each SRAM bank is programmed with an identical copy of the hash table.

The circuit also makes use of off-chip SDRAM to store intermediate flow state information. Since the FPX is designed to process streaming network traffic, it can be expected that packets from numerous TCP flows will be interleaved. The state of components that process each flow, as well as the intermediate counts of n-grams in different languages, are stored in SDRAM after processing each packet that is not the final packet within a TCP flow. The process of storing data from one TCP flow and retrieving that of another is often referred to as a *context switch*.

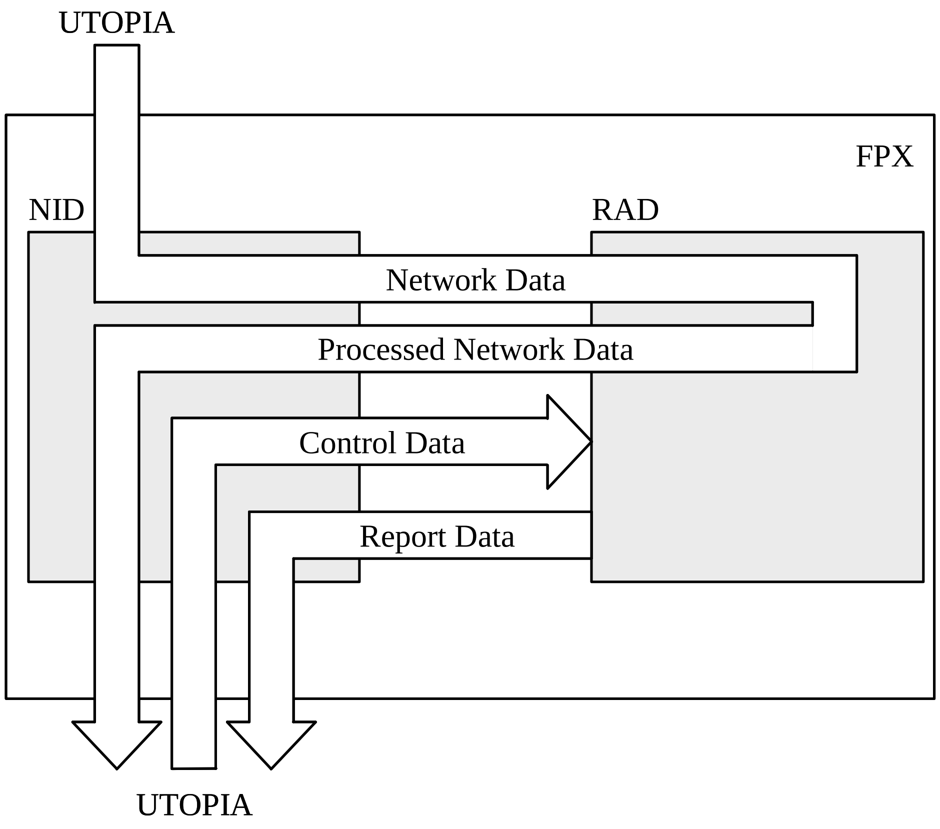


Figure 4.4: Routes for data and control propagation in HAIL

The overall architecture of the HAIL implementation can be seen in Figure 4.3. Network data passes through the incoming TCPLite wrapper and is first buffered in a FIFO queue. It is then streamed through the n-gram extractor, hash unit, and count and score unit for processing. The result of processing is sent to the outbound TCPLite wrapper and/or the report generator.

The HAIL circuit contains a separate data path for the processing of control data. This data passes through the incoming communication wrapper and is buffered into a queue contained within the control processor. If necessary, the control processor requests the use of SRAM. Once granted access, the control processor writes data to off-chip SRAM.

The flow of data in and out of the FPX card implementing HAIL is shown in Figure 4.4. Network data enters the NID via one of the two UTOPIA interfaces on the FPX card. The NID then transmits the data to the RAD. HAIL, which is loaded into the RAD, processes the data and sends it back to the NID, which then forwards the data to a UTOPIA interface. Control data also enters the system through a UTOPIA interface, and is sent to HAIL by the NID. Per-flow reports generated by HAIL are sent to the NID and the propagated to a UTOPIA interface. A detailed description of the different architectural components is presented below.

4.3.1 Input Buffer and Control Unit

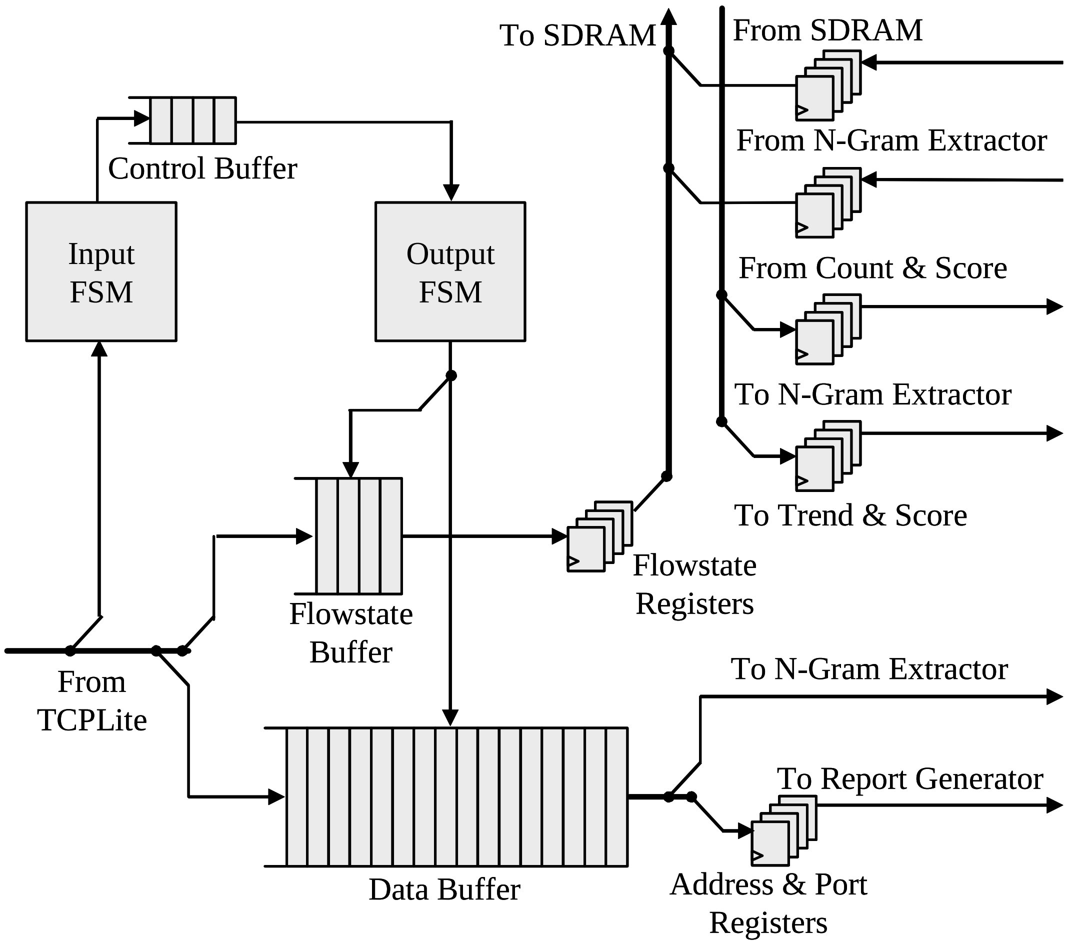


Figure 4.5: The input buffer and control unit

The input buffer and control unit (Figure 4.5) are the first two components in HAIL's data path. They are highly intertwined, as the control unit's finite state machines (FSMs) govern the reading and writing of the FIFO queues in the input buffer unit.

The input buffer unit contains three different queues: The data buffer, the flowstate buffer, and the control buffer. HAIL's data buffer in this implementation is forty bits wide and stores several signals arriving from the front end of the TCPLite wrapper. These signals include the 32-bit packet data field, a 4-bit mask indicating which bytes within the packet data should be processed, a signal indicating when valid TCP payload data is present in the data field, and bits marking the start of the IP packet, the start of the IP packet's payload, and the end of the packet.

The flowstate buffer is thirty-five bits wide and stores TCP state information that the TCPLite wrapper transmits during the first several clock cycles of packet data. The signals stored inside this buffer include a 32-bit flow state signal which contains the TCP flow ID, a bit indicating whether the packet uses the TCP protocol, and bits indicating whether or not the packet is the first or last within the corresponding TCP flow.

An input state machine uses a ten-bit counter to track the amount of packet data written into the data buffer. When the end-of-packet bit arrives, the state machine writes this counter into the control buffer. This serves two purposes. First, if the control FIFO contains any data, there is at least one full packet stored inside the data buffer for processing. Second, the control unit uses the counter to determine how many times it must read from the data FIFO in order to read out all of the packet data.

The control unit also maintains an output state machine that manages aspects of reading from the buffers. Once it detects that an entire packet is buffered inside the data buffer, the state machine reads the information stored within the flowstate buffer. Once this data is read, the state machine examines the "is TCP" bit to determine whether or not the packet is a TCP packet. If so, the state machine also examines the new flow bit to determine whether the packet is the first in the corresponding TCP flow.

If the packet does use the TCP protocol but is not the first packet in the corresponding flow, the output state machine uses the packet's Flow ID as an SDRAM address. Data at the corresponding SDRAM location is read into a set of state registers and then passed to the appropriate processing units. If the packet does not use the TCP protocol or if it is the first packet in the corresponding flow, the aforementioned state registers are cleared and then propagated to the processing units.

Once the flowstate information is sent to the processing units, the output state machine begins reading packet data from the data buffer. The state machine first reads the stored counter from the control buffer. Once read, the state machine reads the entire packet out of the data buffer. As data exits the buffer, the state machine performs two functions: It sends the entire packet to the outbound TCPLite wrapper so that it can be propagated to other processing modules or to an outbound network link, and it sends the payload of the packet to be processed for language identification. The state machine can detect and parse the payload portion of packets that use the TCP or UDP protocols; if a packet does not use either of those protocols, the packet is simply sent to the outbound TCPLite wrapper and not processed for language identification.

Once the packet has been processed, one of two actions can take place. If the packet uses the TCP protocol, and it is not the final packet in the corresponding flow, intermediate state information is written to SDRAM for later retrieval. If the packet uses the UDP protocol or is the last packet of a TCP flow, state information is not written to SDRAM. Instead, the final results of language identification for the TCP flow or UDP packet are sent to the report generator module (Section 4.3.6). The report generator packages the results and sends them to a software program running on a remote host.

4.3.2 N-Gram Extractor

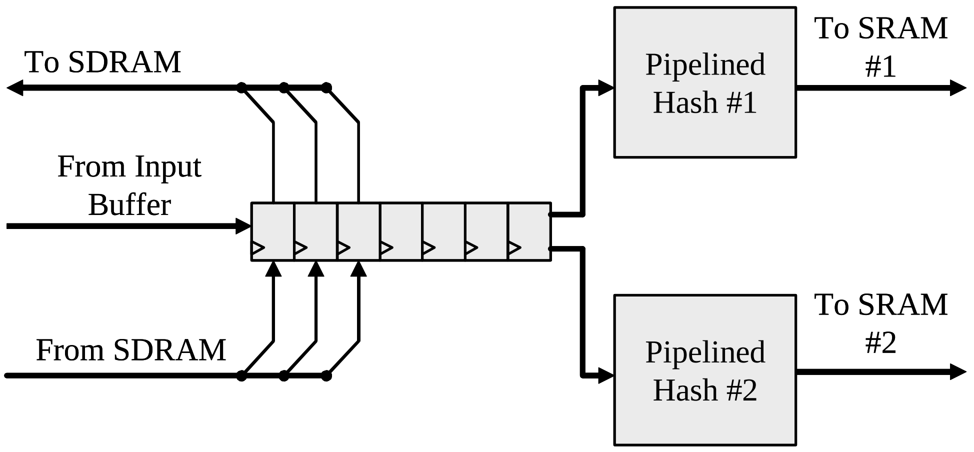


Figure 4.6: The n-gram extractor and hash unit

The first unit in the language identification operation is the n-gram extractor, shown in Figure 4.6. This unit scans the incoming data stream and extracts n-grams within it.

The FPX can send and receive up to four bytes of data per clock cycle. It also contains two banks of off-chip SRAM, which are used to store the languages associated with n-grams. These banks can each be accessed once per clock cycle. When data streams through the FPX at the maximum rate, it is not possible to extract n-grams at every byte offset. Rather, if data enters the system every clock cycle then only half of the n-grams can be looked up in SRAM for language identification. The effects of sampling only one half of the n-grams in a document are shown at the end of this section.

The n-gram extractor is implemented as a seven-entry shift register. As data enters the system, it is shifted into this register. As this occurs, an equal amount of data is shifted out. Bytes entering the shift register are marked as valid for inclusion as part of an n-gram. Once the register contains enough bytes to form one to two n-grams, they are extracted from the register for further processing. As n-grams are extracted from the register, their first and second bytes are marked as invalid for inclusion in other n-grams. This enforces consistency in extracting only every second n-gram.

An example of the shift register's operation is shown in Table 4.1. In this example, the phrase **Hello World** enters the register, up to four bytes at a time, over the course of several clock cycles. A total of four n-grams are extracted as the phrase passes through the register. The *Valid Bytes* column indicates valid bytes for inclusion in an n-gram with a "1", and invalid bytes with a "0". As n-grams are extracted, their first and second bytes are marked as invalid.

Table 4.1: Operation of n-gram extractor on sample payload **Hello World**

Clock Cycle	**Incoming Data**	**Register Contents**	**Valid Bytes**	**Extracted N-Grams**
1	`Hell`		`0000000`	
2	`o Wo`	`Hell`	`0001111`	`Hell`
3	`rld`	`ello Wo`	`0111111`	`llo , o_Wo`
4		`o World`	`0011111`	`Worl`
5		`o World`	`0000111`	

As illustrated in Figure 4.6, the shift register interfaces with off-chip SDRAM. At the end of a TCP packet that is not the final packet in its flow, valid bytes may still be present in the register. This is illustrated in the final clock cycle shown in Table 4.1. The data stream has passed through the shift register, but the letters "rld" are still valid for inclusion in future n-grams. If the phrase **Hello World** was part of a larger data stream, and the remainder of the stream was contained in future TCP packets, the "rld" would be stored in SDRAM. When more TCP packets from the flow arrived, the contents of the register could be loaded from SDRAM for processing of n-grams beginning with these letters.

Data Subsampling

As mentioned above, platform constraints require the FPX implementation of HAIL to sample only half of the n-grams in a document. This has a small but non-negligible effect on the performance of HAIL.

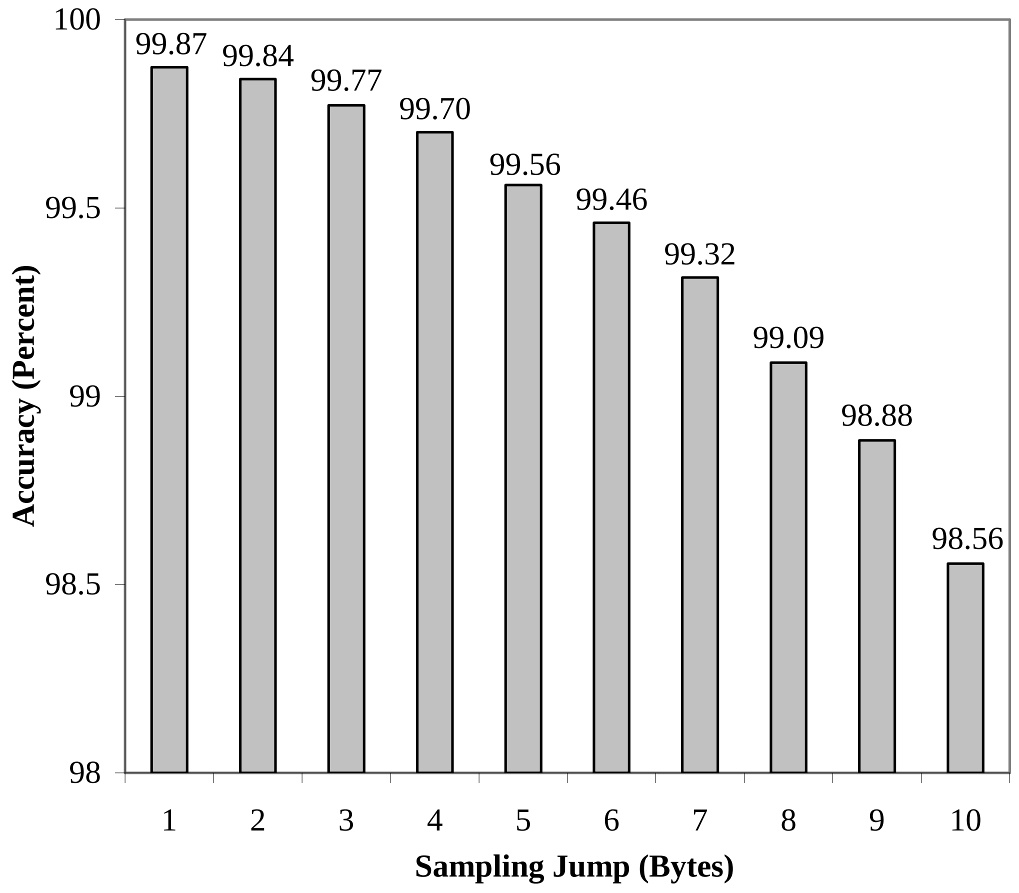


Figure 4.7: The effect of subsampling on accuracy of language identification

Figure 4.7 shows that sampling half of the n-grams has a small effect on accuracy. The "Sampling Jump" on the x-axis indicates the ratio of n-grams sampled in a document; $1/j$ (where j represents the sampling jump) is the ratio of n-grams sampled.

In systems that contain fewer and/or slower memory banks than the FPX platform, HAIL remains fairly viable. In the experiments performed on 34 languages, a 99%

rate of accuracy is achievable even when only 1/8 of the n-grams can be sampled. However, reducing the sampling ratio beyond this point causes a rapid drop-off in accuracy.

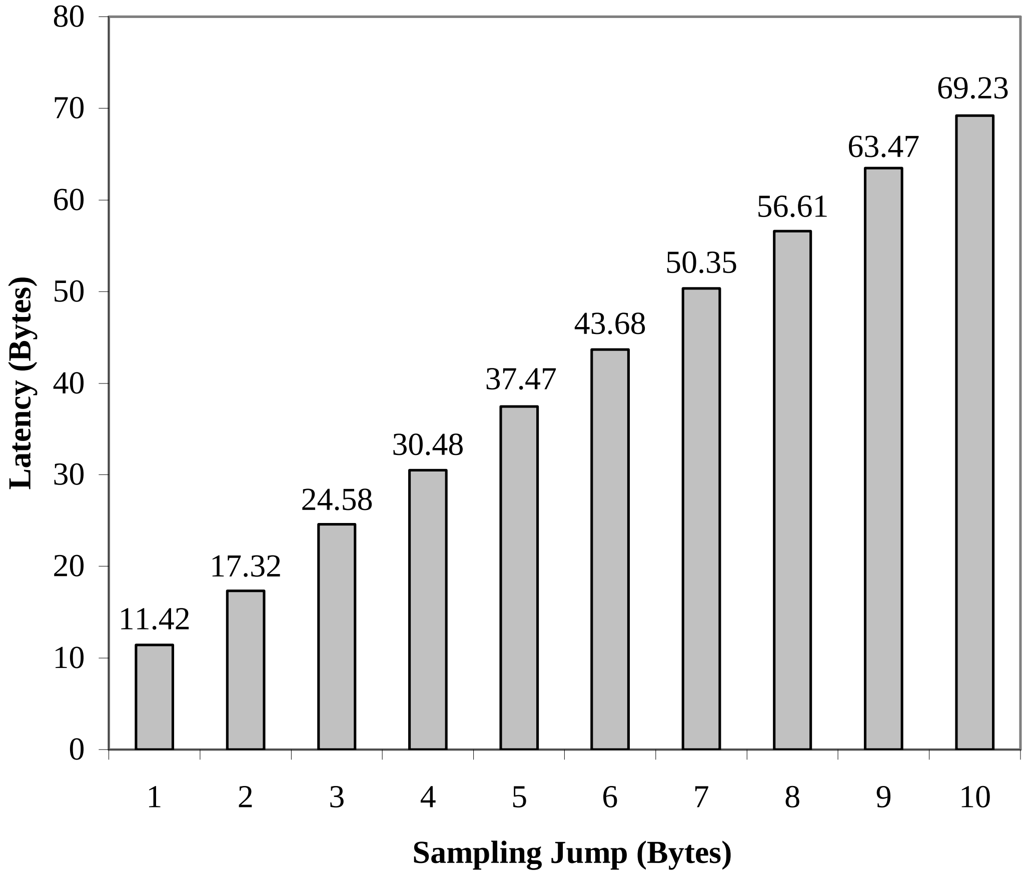


Figure 4.8: The effect of subsampling on latency of language identification

Figure 4.8 shows the effect of the sampling ratio on latency. Unfortunately, decreasing the sampling ratio from 1 to 1/2 causes an approximately 6-byte increase in latency. The increase in latency as the sampling jump is raised further is roughly linear.

An increase in latency is an inevitable side effect of reducing the number of n-grams sampled. As shown in the figure, the average latency for sampling every n-gram is 11.42 bytes. Thus, an average of 11.42 different n-grams can be sampled if the n-grams that begin at each of these bytes are extracted. If every other n-gram is sampled, an average of only 5.72 n-grams can be extracted in the same amount of data.

The average latency for sampling every other n-gram is 17.32 bytes. Sampling half of the n-grams beginning at each of these bytes allows an average of 8.66 n-grams to be extracted. Therefore, when sampling every other n-gram, *fewer* n-grams are required. Increasing the size of the jump further decreases the average number of n-grams required to arrive at a result; an average of 8.19 n-grams must be sampled when 1/3 of the n-grams are extracted, and an average of only 6.92 n-grams need to be sampled when 1/10 of the n-grams are sampled.

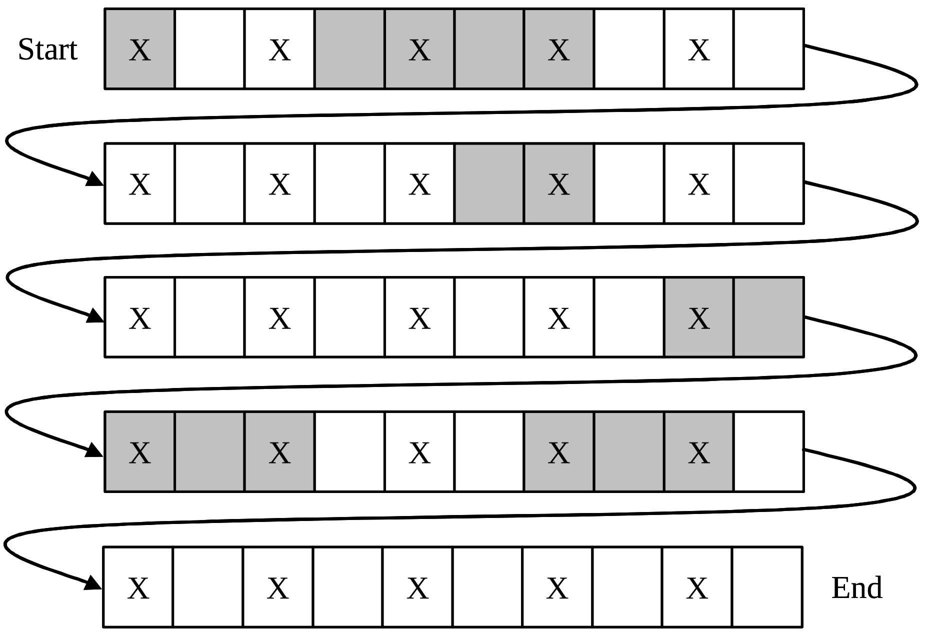


Figure 4.9: An approximation of n-grams in a document with 50 n-grams

This result is certainly counter-intuitive. One might assume that the number of n-grams required to identify a document's language would remain constant or even grow as the sampling jump was increased. To help explain this, an experiment was run to evaluate patterns of “good” n-grams (those that hash to a value containing the correct language identifier for a document) and “bad” n-grams (those that hash to a value that does not contain the correct language identifier for a document).

After creating a training set and then analyzing approximately 2,000 documents, three pieces of information were found. First, when good n-grams appear, they appear, on average, in a series of 6.77 sequential good n-grams. Second, when bad n-grams appear, they appear, on average, in a series of 3.36 bad n-grams. Finally, the first good n-gram appears, on average, as the 2.24nd n-gram in a document.

For simplicity of demonstration, these three figures were rounded from 6.77 to seven, 3.36 to three, and 2.24 to two. Figure 4.9 shows the breakdown of a hypothetical document, which follows these figures, into a series of n-grams. White boxes represent good n-grams, while gray boxes represent bad n-grams. Boxes containing an **X** represent n-grams that would be analyzed if only every other n-gram was sampled. The figure is meant to be read left-to-right, top-to-bottom.

Examining the boxes in the figure will show that the first good n-gram is the second n-gram in the example document, that there are five streaks of bad n-grams with an average length of three, and there are five streaks of good n-grams with an average length of seven.

Processing the document if every n-gram is sampled reveals that, on the 10th byte of the document, the correct language will be granted a permanent counter (assuming a trend depth of three is used, as in Section 3.2.3). This corresponds to the 10th n-gram that was sampled.

On the other hand, processing the document by sampling every other n-gram reveals that the correct language will be granted a permanent counter on the 13th byte of the document. This corresponds to only the seventh n-gram that was sampled.

Obviously, this is not always the case. If the second group of bad n-grams (corresponding to the fourth through seventh n-grams in the document) were pushed back by one n-gram, then sampling every n-gram would yield a permanent counter for the correct language after only four n-grams. In the average case, however, this example appears to be accurate.

There is another factor related to this observation, as well. Since bad n-grams appear in relatively short streaks, sampling every n-gram is likely to pick up on trends of incorrect language identifiers more frequently than sampling every other n-gram.

Figure 4.10 validates this claim. As the figure shows, a sampling jump of one n-gram begins to require (albeit slightly) more counters than a sampling jump of two n-grams. This means it is more common for an incorrect language to be granted a permanent counter first when sampling every n-gram compared to sampling every other n-gram.

4.3.3 Hash Unit

The hash units (shown previously in Figure 4.6) are used to reduce n-grams to 19-bit SRAM addresses. The hash units each perform a cyclic redundancy check (CRC),

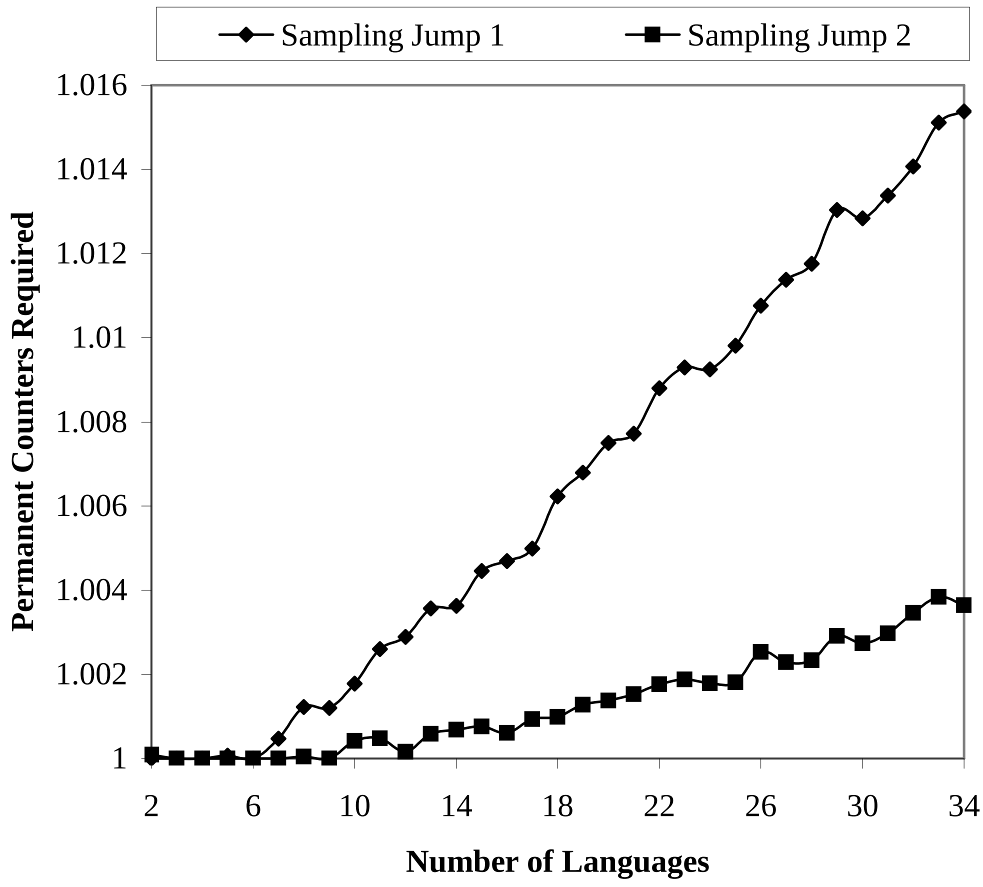


Figure 4.10: The number of permanent counters required for different sampling jumps as the number of languages is adjusted from two to 34

which is a hash function typically used by computer systems in error detection. Fundamentally, a CRC is calculated by treating the input data as an integer and dividing it by another binary value, referred to as a *generator polynomial*. The generator polynomial must be one bit longer than the CRC that will be created, and its most significant and least significant bits must be equal to 1. The remaining bits may be any combination of 0s and 1s. The remainder of this division operation becomes the CRC. Details on the calculation of CRCs are provided in [61].

A commonly-used CRC generator polynomial is defined in the standard for detecting errors in data transmitted over Ethernet links [69]. This polynomial is shown below in three common forms: binary (Equation 4.1); hexadecimal (Equation 4.2), in which the most significant bit is omitted because a generator polynomial's most significant bit is always 1; and in polynomial form (Equation 4.3).

$$G(x) = 1\ 0000\ 0100\ 1100\ 0001\ 0001\ 1101\ 1011\ 0111 \tag{4.1}$$

$$G(x) = \text{0x04C11DB7} \tag{4.2}$$

$$G(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1 \tag{4.3}$$

CRCs are particularly well-suited for implementation in hardware because they are calculated using modulo 2 binary arithmetic. This is a binary arithmetic system in which addition, subtraction, and multiplication are performed without carrying. Thus, addition and subtraction are performed by the exclusive-OR (XOR) of individual bits. Modulo 2 binary division is performed as a series of subtractions; therefore, a CRC can be performed as a series of simple XOR operations.

HAIL's hash units use the generator polynomial defined by the Ethernet standard. When n-grams are extracted from the shift register, they are sent to the hash units; each hash unit performs a pipelined CRC operation, operating on one byte of an n-gram each clock cycle[1]. The CRC hashes the n-grams into a 32-bit output, which is then truncated to 19 bits and used as an SRAM address.

The addresses generated by the hash units are used to read SRAM. Each SRAM location is 36 bits in width and contains four nine-bit language identifiers. Each of these identifiers represents the language from the corresponding slot (refer to Section 3.3 for the definition and assignment of slots) which contains the highest frequency of n-grams that map to the SRAM address. Once read from SRAM, the individual language identifiers are separated and sent to the count and score unit.

4.3.4 Count and Score Unit

Each clock cycle, up to two 36-bit values are passed to the count and score unit from SRAM. The count and score unit separates the four 9-bit language identifiers from these values, passes them to the individual trend units, performs the trend and counting operations described in Section 3.2.3, and determines the language that best matches the document.

Figure 4.11 illustrates the general architecture of the count and score unit. Incoming 36-bit values are split into their four 9-bit language identifiers and dispatched to the

[1]The CRC function used in HAIL was created with the Easics CRC tool [38], which automatically generates a synthesizable CRC implementation in VHDL based on a user-specified input width and generator polynomial.

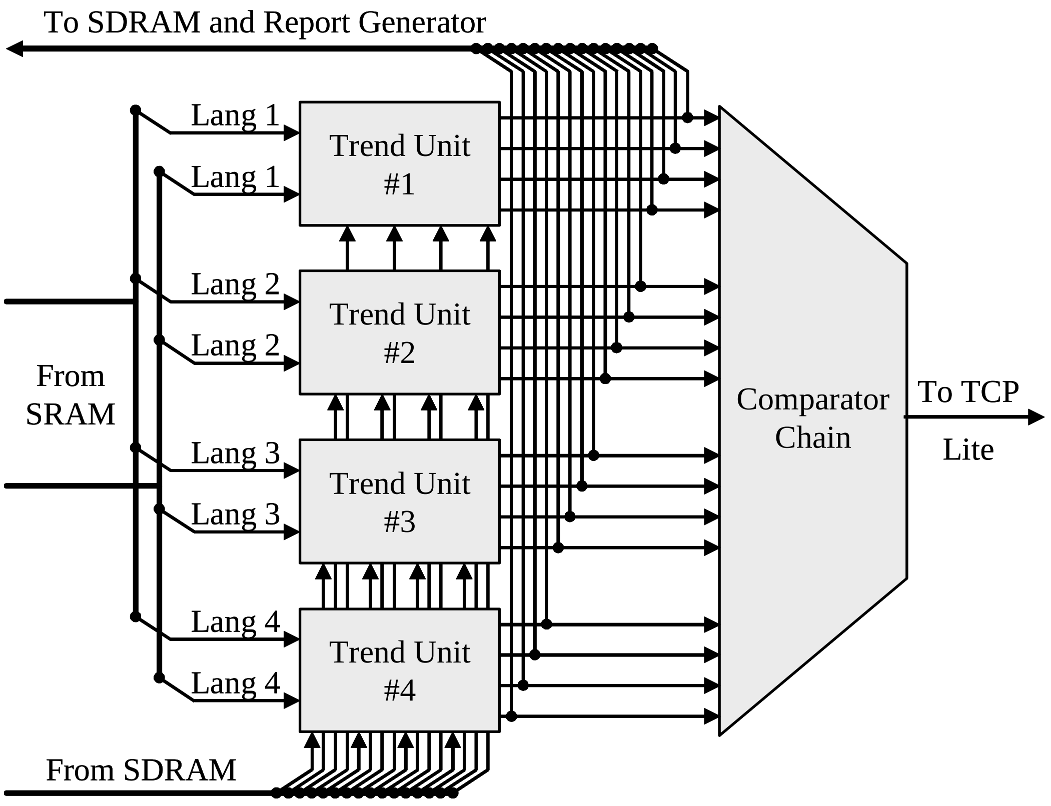


Figure 4.11: Architecture of the count and score unit

trend units. Languages appearing in a particular SRAM position are always passed to the same trend unit, as explained in Section 3.3.

Each individual trend unit (shown in Figure 4.12) maintains its own trend register and set of permanent counters. When language identifiers enter a trend unit, they are first compared to the permanent counters. If one or both of the language identifiers match a language stored in a permanent counter, the corresponding counter or counters are incremented appropriately.

If there is no match, the language identifiers are sent to the trend register and a variety of actions can occur. A distinction is made between the first language identifier and the second language identifier; the first language identifier corresponds to the n-gram that was extracted earlier in the data stream than the second. There are several actions that can be taken based on the values of the incoming language identifiers, the value of the current trend, and the value inside the trend register itself.

1. If both language identifiers are set to 0 (indicating no language), nothing occurs.

2. If one of the two language identifiers is valid (greater than 0) and the language matches the current trend, increment the value inside the trend register. If this causes the trend register to reach its threshold, find a permanent counter for the language.

3. If one of the two language identifiers is valid and the language does not match the current trend, reset the value inside the trend register to 1 and set the current trend to the incoming language.

4. If both language identifiers are valid, equal to each other and match the current trend, add 2 to the value inside the trend register. If this causes the trend register to reach its threshold, find a permanent counter for the language.

5. If both language identifiers are valid, equal to each other and do not match the current trend, reset the value inside the trend register to 2 and set the current trend to the incoming languages.

6. If both language identifiers are valid and **not** equal to each other, the first incoming language is equal to the current trend and would cause the trend register to reach its threshold, find a permanent counter for the first language. Reset the value inside the trend register to 1 and set the current trend to the second incoming language.

7. If both language identifiers are valid and **not** equal to each other, and either the first language is **not** equal to the current trend or its presence would **not** cause the trend register to reach its threshold, reset the value inside the trend register to 1 and set the current trend to the second incoming language.

In steps 2, 4 and 6 the process of finding a permanent counter for a language is mentioned. This process is straightforward. The set of permanent counters inside the trend unit is probed. The first unused permanent counter is assigned to the language that has caused the trend register to reach its threshold, and the value inside the trend register is moved inside the permanent counter.

Each permanent counter is ten bits in width and can count to a maximum value of 1,023. When a counter's maximum value is reached, the counter *saturates*, meaning that it remains fixed at the maximum value rather than rolling back to zero.

If all of the permanent counters are already used, then the language is simply discarded. However, this situation should rarely affect the circuit's identification of the best language associated with the document; given the figures presented earlier for

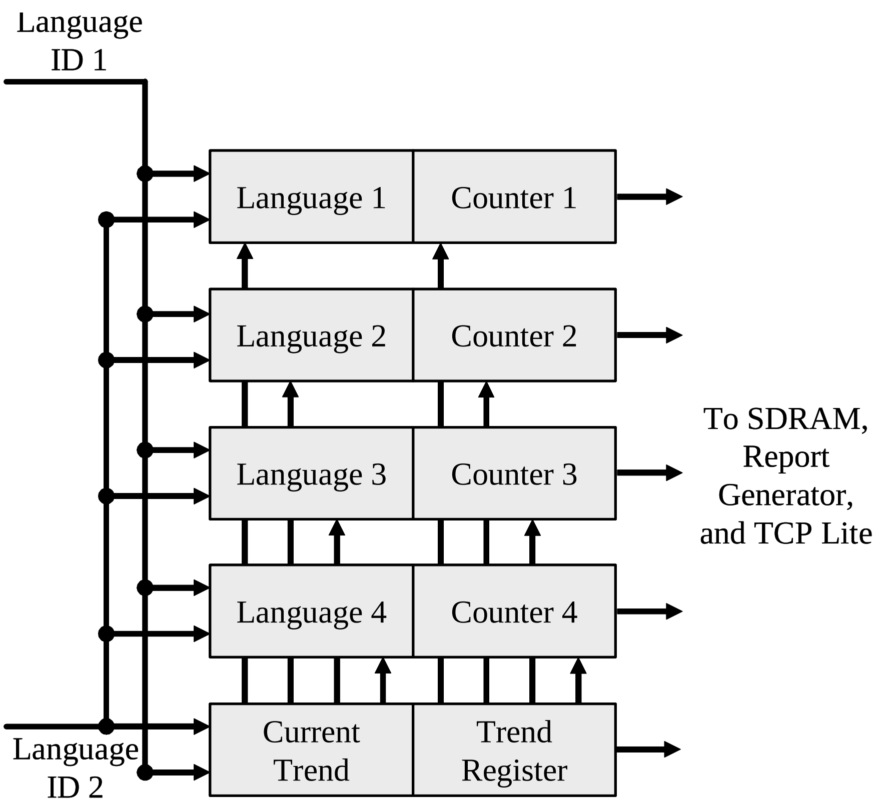


Figure 4.12: Architecture of a single trend unit

the number of permanent counters required, the document's language is typically found very early in the document.

At the end of each packet, counters and their associated languages are sent to the pipelined comparator chain shown in Figure 4.11. The comparator chain performs $c - 1$ comparisons in $\log_2 c$ clock cycles, where c is the total number of permanent counters in the trend units. This is performed simply by comparing pairs of counters in each clock cycle and propagating the larger of the two counters to the next stage of the pipeline. The language with the largest counter is sent to the TCPLite wrapper for use by other modules, if appropriate.

Other data is also sent from the count and score unit at the end of each packet. If the packet was the last in its corresponding TCP flow, the counters and their corresponding languages are sent to the report generator (see Section 4.3.6), which packages the data into a packet and sends it to a remote host. If the packet was not the last in its corresponding TCP flow, the counters, their corresponding languages, the trend register and the current trend are sent to SDRAM. This data, along with the shift register from Section 4.3.2, is stored and later retrieved when the next packet from the corresponding TCP flow arrives.

4.3.5 Control Processor and SRAM Programmer

The SRAM lookup tables and various flags within the HAIL circuit can be programmed remotely over a network by sending control packets into the system. These control packets are processed by the reporting module discussed in Section 4.2.2 and the payloads are forwarded to the HAIL module. The communication wrappers parse these payloads and provide the data, as well as various control signals, to the control processor shown in Figure 4.13.

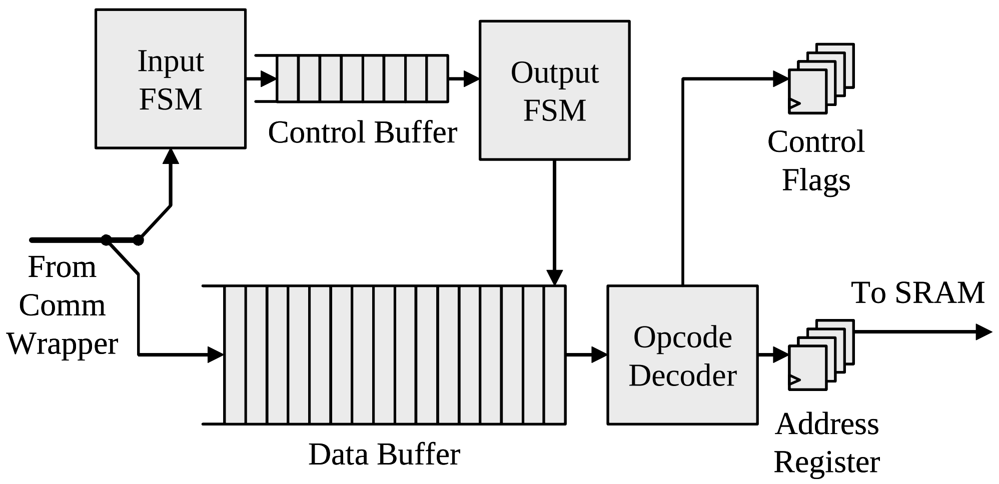


Figure 4.13: Architecture of the control processor and SRAM programmer

The control processor consists of two buffers, two finite state machines, and supporting logic. As data enters the module from the communication wrapper, it is buffered inside the module's data buffer. Once an entire control packet's payload is stored inside the data buffer, the input state machine writes a counter into the control buffer. This counter indicates the size of the control packet payload written into the data buffer.

The output state machine monitors the status of the control buffer. If the control buffer is not empty, an entire packet has been stored inside the data buffer. At this point, the output state machine will read the control buffer to determine the size of the packet stored inside the data buffer. It then reads the specified amount of information from the data buffer and performs the appropriate action.

The action taken by the output state machine is determined by the opcode, which is contained within the first 32-bit word of the control packet payload. Currently, only three opcodes are used: `0x1`, which indicates that the packet contains SRAM locations to be programmed; `0x2`, which enables the report generator (see Section 4.3.6); and `0x3`, which disables the report generator. The report generator is enabled by default. The two relevant packet payload formats are shown in Figures 4.14 and 4.15, respectively.

The packet to program SRAM is variable length. After the opcode field, it contains between one and 180 (the maximum number that will fit within a packet sent to the reporting module) address/data pairings. As shown in Figure 4.14, the packet consists of one 32-bit data word containing a 19-bit address and four data bits, followed by a second 32-bit word containing thirty-two data bits. The data being written to SRAM must be split across two 32-bit words because the data written to each SRAM location is 36 bits wide. The most significant bit of each 9-bit language identifier is stored within the field labeled `Hi Bits`, while the least significant bit of each 9-bit language identifier is stored within the fields labeled `Slot [0-3] lo bits`. When the first 32-bit word is read from the data buffer, the address and the four most-significant bits are stored in the address register. When the second 32-bit word is read, the four language identifiers are assembled and sent to SRAM alongside the address.

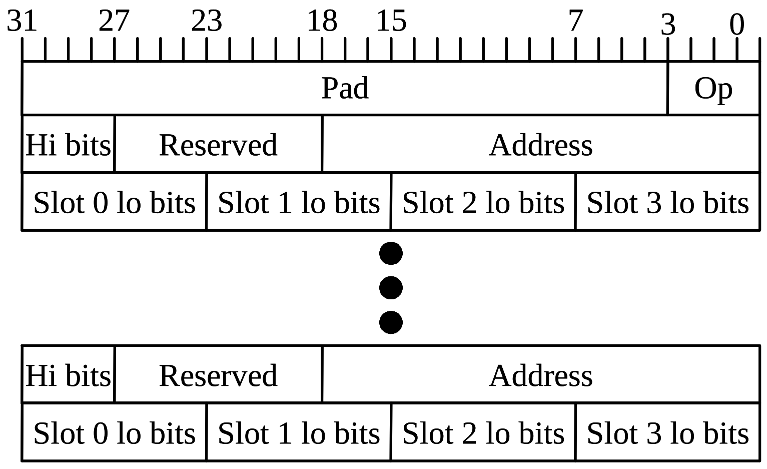


Figure 4.14: Packet format to program SRAM

The packet to toggle report generation is fixed at one 32-bit word, which contains the opcode. Four bits are currently allocated for the opcode, but this size can be easily increased if needed to expand the number of options.

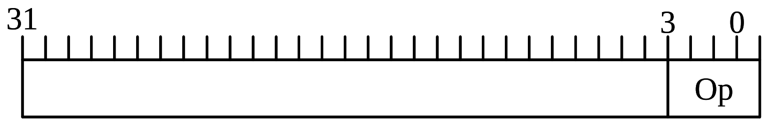


Figure 4.15: Control packet to set and disable hardware control bits

4.3.6 Report Generator

In addition to providing the highest-scored language identifier to the outbound TCPLite wrapper, HAIL can also provide a detailed summary to a remote software host. At the conclusion of a TCP flow or UDP packet (as UDP does not define flows in its protocol specification), the circuit can output language identification and statistical data to the report generator (Figure 4.16), which is not to be confused with the reporting module described in Section 4.2.2.

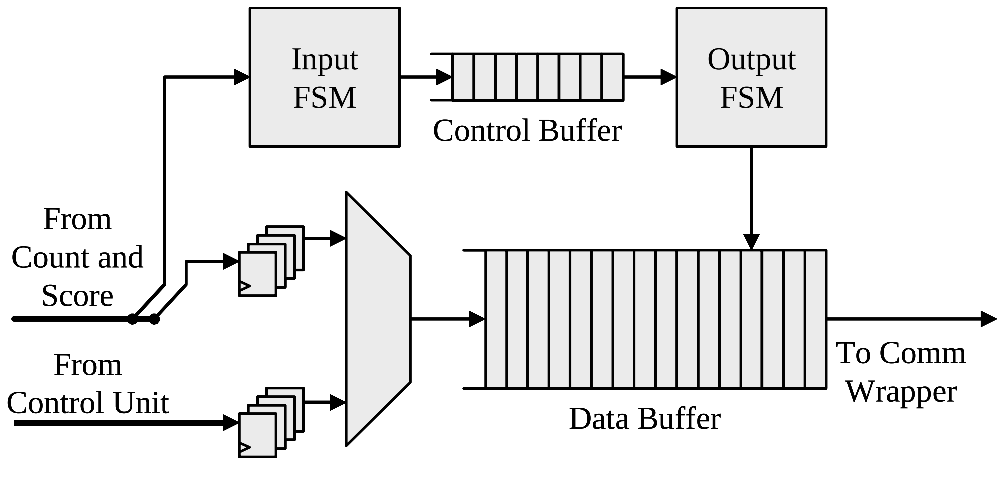


Figure 4.16: Architecture of the report generator

When the TCP flow or UDP packet terminates, several pieces of data are sent to the report generator. The source and destination IP addresses and ports of the flow or packet are sent from the control unit, and the sixteen language identifiers and counters are sent from the count and score unit. This data is sent in parallel, and is initially placed in registers for storage. Once the data is received, it is streamed into a data buffer, 32 bits at a time. The data is packed in the order and format used to transmit it in packet form; the exact packet format is shown in Figure 4.17.

Once all of the information is stored in the data buffer, a single bit is written to the control buffer for control purposes. Unlike with the input buffer and control cell

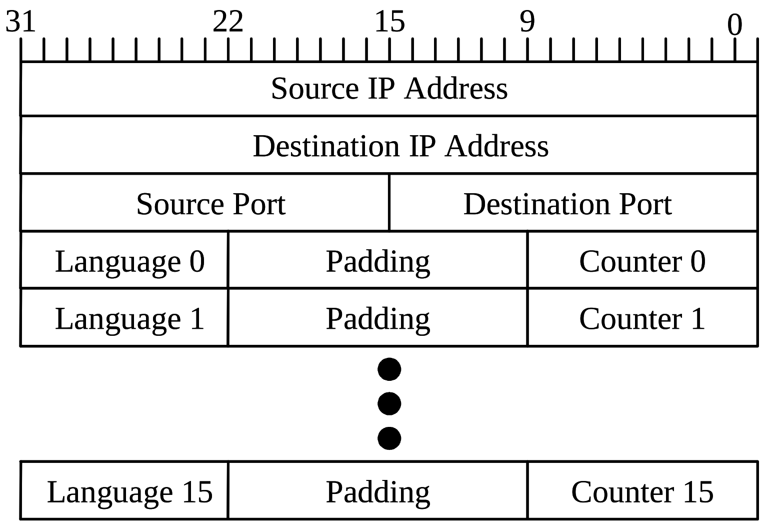


Figure 4.17: Packet format sent by report generator

buffer, a counter does not need to be written to the control buffer; the length of the packet payload is fixed at nineteen 32-bit words. The presence of a bit in this buffer triggers the output state machine to begin reading from the data buffer and into the outbound communication wrapper. The communication wrapper splits this data into ATM cells and sends it to the reporting module, which attaches an IP, UDP, and reporting header before sending the packet to a remote software host.

4.4 System Implementation

The architecture described in this chapter was implemented in VHDL, synthesized, placed, and routed for the Xilinx XCV2000E-8 FPGA found on the FPX platform. This section describes the results and other relevant details.

4.4.1 Implementation Results

The FPX implementation of HAIL, along with infrastructure components used for communication and memory access, consumed 29.5% of the four-input lookup tables, 24.5% of the flip-flops, and 53.1% of the Block RAMs available within the FPGA. Details of the device utilization are shown in Table 4.2. The circuit achieves a maximum clock frequency of 87.53 MHz.

Table 4.2: HAIL device utilization on Xilinx XCV2000E-8

Component Name	**Look-up Tables**	**Flip Flops**	**Block RAMs**
Input Buffer and Control Unit	1,696	2,217	24
N-Gram Extractor	503	294	0
Hash Units	505	348	0
Count and Score Unit	2,182	1,244	0
Control Processor	274	257	10
Report Generator	661	633	3
Top Level Logic	1,072	310	0
HAIL Subtotal	6,893	5,303	37
SRAM Controllers	199	271	0
SDRAM Controllers	1,362	1,520	8
TCPLite Wrapper	1,277	912	21
Communication Wrapper	1,014	898	19
Infrastructure Subtotal	3,852	3,601	48
Grand Total	10,745	8,904	85
Total Available	36,400	36,400	160
Percent	29.5%	24.5%	53.1%

This implementation consumes less than one third of the logic elements and slightly over half of the memory available on the Virtex XCV2000E-8. The HAIL circuit itself, not counting infrastructure components, uses under one fifth of the logic elements and slightly over one fifth of the memory elements available.

The low resource utilization has two implications. First, the implementation of HAIL is small enough so that other processing functions can be carried out within the same FPGA. Appendix C shows a configuration that contains optional processing modules on other FPX cards; provided that the optional processing modules use few enough resources, they could in fact be implemented on the same FPGA.

A second implication of the resource utilization is that HAIL can be implemented in virtually any modern FPGA. At the time of writing, the Virtex XCV2000E-8 is several years old and contains fewer resources than many current FPGAs. Thus, most modern FPGAs should be capable of utilizing the HAIL circuit.

4.4.2 Throughput

The throughput of the system is a function of average packet size, clock rate, packet processing overhead and bytes processed per clock cycle. To calculate throughput, the number of packets per second must first be calculated as in Equation 4.4. In the HAIL implementation for the FPX, the clock rate is $8.0 * 10^7$, four bytes are processed per clock cycle (as stated in Section 4.3.2), and the packet processing overhead is 31 clock cycles. The overhead includes time required to read state information and determine appropriate actions for each packet, as well as time required to store TCP context information in and retrieve TCP context information from SDRAM. The overhead cannot be overlapped with packet data processing.

$$\frac{Packets}{Second} = \frac{Clock\ Rate}{\frac{Packet\ Size}{Bytes\ per\ Clock} + Overhead} \tag{4.4}$$

The throughput in Gigabits per second can be calculated from the number of packets per second that can be processed for the given average packet size. The figure is multiplied by eight to convert bytes to bits and divided by $1.0 * 10^9$ to convert bits to Gigabits.

$$\frac{Gigabits}{Second} = \frac{\frac{Packets}{Second} * Packet\ Size * 8}{1.0 * 10^9} \tag{4.5}$$

Because the number of packets processed per second changes with the average packet size, throughput also changes with average packet size. Figure 4.18 shows the change in throughput as the average packet length is altered from 0 to 1500 bytes. The horizontal line between 2 and 3 Gigabits per second represents the bandwidth of a fully-utilized OC-48 fiber optic network link. The Virtex XCV2000E-8 implementation is capable of processing at this capacity when the average packet length is greater than approximately 985 bytes.

In addition to providing throughput for the Virtex XCV2000E-8 (part of the Xilinx's Virtex-E FPGA series), Figure 4.18 also shows throughput figures for the same HAIL architecture if it were implemented on a Virtex-II and Virtex-4 FPGA. Based on the approximate maximum clock frequencies of the circuit if implemented on these FPGAs, the systems could process at OC-48 capacity when the average packet length is greater than 192 and 86 bytes, respectively. More detail on these FPGA devices is provided in Section 4.4.3.

For comparison, the HAIL software used throughout much of this thesis was configured in the same way as hardware and timed for 500-byte documents. On average, the software processed 500-byte documents at a rate of 0.0128 Gigabits per second (approximately 1.60 Megabytes per second) when documents are pre-loaded into memory.

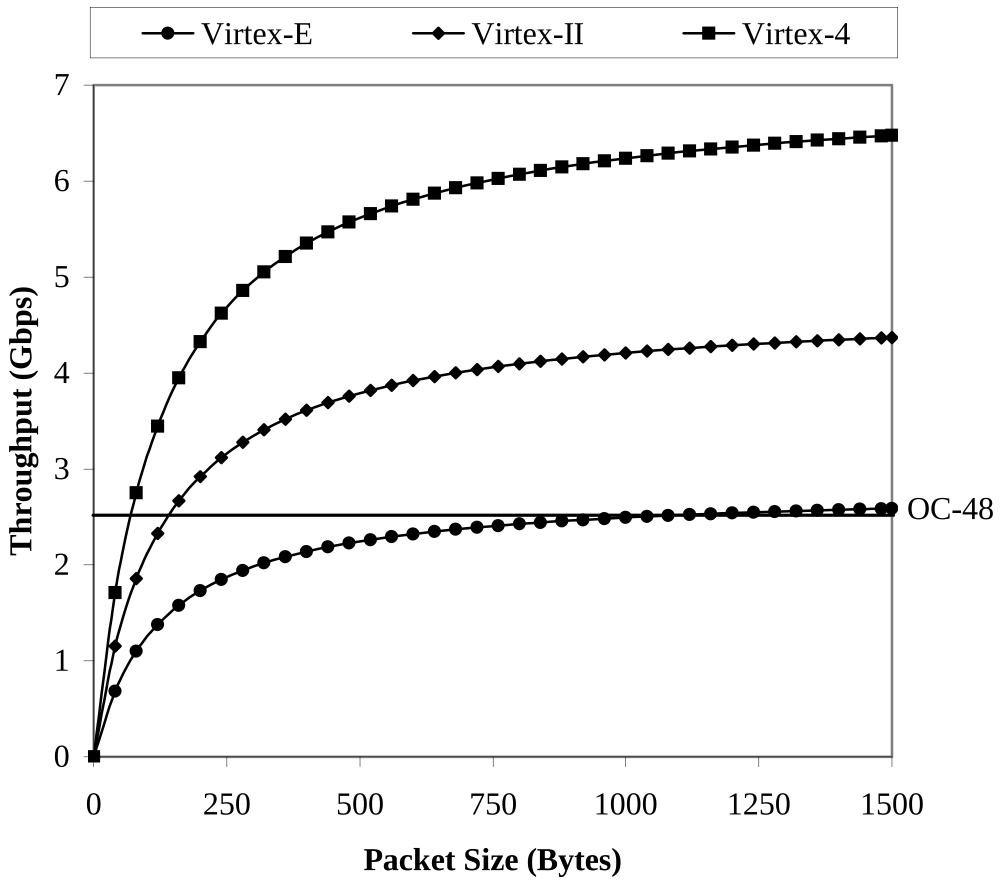


Figure 4.18: HAIL circuit throughput as a function of average packet size

It processed these documents at an average rate of 0.00892 Gigabits per second (approximately 1.12 Megabytes per second) when documents were loaded from a hard disk. These results were obtained on a personal computer with an AMD Athlon 64 X2 Dual Core Processor 4200+ clocked at 2.21 GHz and one Gigabyte of DDR SDRAM.

4.4.3 Next-Generation FPGAs

The Xilinx Virtex XCV2000E-8 FPGA used on the FPX platform belongs to Xilinx's Virtex-E series of FPGAs, which were first introduced in 1998. Since that time, Virtex has released new FPGAs in the series, including the Virtex-II in 2001 and the Virtex-4 in 2004 [79].

While the Virtex XCV2000E-8 is among the largest and most powerful FPGAs in its family [83], it is a relatively old device that has been surpassed by newer technologies. While lower-end versions of newer devices contain fewer logic elements and less memory than the FPGA used on the FPX, the Virtex XCV2000E-8 is inferior in both logic and memory to no fewer than three varieties of the Virtex-II and nine varieties of the Virtex-4 [80].

The most logic-dense variety of Virtex-4, the XC4VLX200, contains 4.64 times as many logic elements and 9.45 times as much on-chip memory as the Virtex XCV2000E-8 [83, 80]. Other varieties of the Virtex-4 sacrifice logic and memory for embedded digital signal processors and PowerPC microprocessors that provide added functionality [80].

Table 4.3: Estimated HAIL clock frequencies on Virtex series FPGAs

Device Name	**FPGA Series**	**Frequency**	**Maximum Throughput**
XCV2000E-8	Virtex-E	87.53 MHz	2.59 Gbps
XC2V8000-6	Virtex-II	147.9 MHz	4.37 Gbps
XC4VLX200-12	Virtex-4 LX	219.1 MHz	6.47 Gbps

Table 4.3 shows a comparison in clock rate between the version of HAIL currently implemented on the Virtex-E and the approximate maximum clock rate if the same architecture were implemented in Virtex-II and Virtex-4 LX series FPGAs. The clock rates provided for the latter two FPGAs were approximated by Synplify Pro, an FPGA synthesis tool created by Synplicity [72]. Implemented in the Virtex-4 LX, the clock rate and maximum throughput are nearly tripled.

In addition to next-generation FPGAs, improvements to HAIL can also be achieved through newer memory technology [3]. The size of available SDRAM devices has increased, allowing for support of a larger number of TCP flows. Newer SDRAM technology is also available, including Dual Data Rate (DDR) SDRAM which allows a burst of two contiguous memory locations to be read or written in the same clock cycle. This would eliminate time from the context switch overhead since memory reads and writes could be performed in half as many clock cycles.

Newer SRAM technology is available as well. First, larger banks of SRAM than what is found on the FPX are commercially available. A larger address space would cause fewer hash collisions and therefore improvements in both accuracy and latency. Furthermore, newer SRAM technology is available, such as Quad Data Rate (QDR) SRAM which allows bursts of two contiguous memory locations to be read and written in the same clock cycle. This would increase the number of language identifiers that could be associated with each hashed n-gram. By storing up to eight language identifiers in consecutive SRAM addresses, all eight could be read in a single clock cycle. This would provide a modest improvement in accuracy and latency. The drawback of this would be that each hashed n-gram would require data to be stored two contiguous memory locations, halving the effective address space.

The combination of newer FPGAs and newer memory technology could also have a particularly profound effect on HAIL: The elimination of the trend register and memory slot clustering schemes. Recall that the trend register was created to reduce hardware resources, to reduce the amount of time required to perform context switches and to reduce the amount of SDRAM that must be reserved for each TCP flow. The memory slot clustering was an effort to ease the amount of logic required by a trend register system that spanned multiple SRAM positions.

With significantly larger and faster FPGAs, maintaining large numbers of on-chip counters and comparators becomes a much more trivial task. Furthermore, larger SDRAM devices can cause an increase the amount of data allotted to each TCP flow while supporting the same overall number of flows. Finally, faster SDRAM devices can reduce the time required to perform context switches of large amounts of data. With these three challenges overcome, the elimination of the trend register system is possible.

Chapter 5

Conclusion

In 2004, those who spoke English as their first language constituted only one-third of Internet users, and approximately one-third of all web pages were written in a language other than English. As the number of non-English speakers and web pages increases, several problems arise. A system capable of rapidly and accurately identifying the primary language used in documents traveling over network links can prove useful in a variety of tasks, including language-based forwarding and routing, as a preprocessor for document classification, as a system for labeling records, and as a means of removing unwanted documents from a data stream.

The problem of language identification is not trivial. Most languages can each be represented with numerous character encodings. While there is a migration towards the Unicode standard, a great deal of applications still use older single- and double-byte encodings. To complicate the situation, documents are not always labeled with their respective character encoding. When documents are labeled, the encoding can be represented in numerous ways and at different locations throughout the document depending on the application that created the document. Therefore, a useful method for language identification requires knowledge of both character encodings and the languages themselves.

5.1 Summary of Approach

HAIL, the algorithm developed in this thesis for the hardware-accelerated identification of languages and character encodings, is based on existing n-gram based methods for language identification. The algorithm must be trained on sets of data in each language and character encoding pair of interest. During the training process, n-grams of one length are extracted from each document and hashed.

Once training is complete, the algorithm cycles through the hashed n-grams and determines the frequency at which each hashed n-gram appears in each language. However, constraints of memory devices mandate that only four language identifiers can appear in each memory location. Although storing the best four languages associated with each hashed n-gram would be the ideal configuration of memory, it causes problems.

The problem is caused primarily due to a system based on a "trend register". This system requires several consecutive n-grams representing the same language to appear before the language is counted. While the system confers benefits to the hardware implementation, attempting to search for trends across four positions within memory creates implementation problems of its own.

This problem was solved by using data clustering to ensure that language identifiers for a particular language always appear in the same position within SRAM. Consequently, language identifiers stored at each memory location do not necessarily represent the languages best associated with each n-gram. Rather, they represent the best languages *from their position within memory* associated with each n-gram.

Despite the tradeoffs required for hardware implementation including using only one length of n-gram, scoring documents by a simple counting of language identifiers, using a simple hash table without collision resolution, not maintaining counters for every language and character encoding pair, using clustering to force language identifiers into less-than-optimal arrangements within memory and not sampling every n-gram, the implementation of HAIL achieved accuracy in excess of 99.8% during experiments. Furthermore, language identification typically occurred after sampling less than 18 bytes of a document's text.

5.2 Contributions

Several contributions were made through the work culminating in this thesis. First and foremost is HAIL itself, which is a novel algorithm designed for identifying languages and character encodings in reconfigurable hardware. The second was the creation of a functional circuit in FPGA hardware that carries out the HAIL algorithm exactly as described in this document. As demonstrated in this thesis, the algorithm is highly accurate, capable of identifying a document's language and character encoding from up to 511 language and encoding pairs, and doing so after sampling only a small amount of data. Furthermore, the current hardware implementation achieves a maximum throughput above that of a fully utilized OC-48 network link.

Other contributions can be found in the nuances of the HAIL algorithm. The first is the trend register system used primarily to reduce the amount of hardware resources required in the algorithm's implementation. This technique could be extended to other classification tasks, such as identifying a document's author or meaning, or distinguishing between different file types.

The second is the use of an expectation maximization algorithm to assign language identifiers to particular positions within memory in order to minimize the number of comparisons. While this specific use is not particularly useful outside of a trend register-based system, this unconventional use of expectation maximization indicates that the method may prove useful in other problems regarding memory configuration.

5.3 Future Work

There are areas in which future work on HAIL can be performed. First, the circuit developed for the Virtex XCV2000E-8 FPGA could be ported to more modern FPGAs. As shown previously, newer FPGAs yield drastically higher clock rates and can increase throughput well beyond the maximum achieved on the FPGA used currently.

On a related note, newer memory technologies can also provide advantages. Higher-throughput SDRAM can decrease the time required to perform context switches. Larger SDRAM devices can increase the number of TCP flows supported by the system. Larger SRAM devices can reduce the number of hash collisions, improving latency and accuracy. Furthermore, higher-throughput SRAM can allow more languages to be associated with each n-gram to further improve accuracy and latency. The combination of larger FPGAs and larger, faster SDRAM can also eliminate the need for the trend register system described in this thesis.

HAIL would also benefit from a larger corpus of multilingual documents. In the experiments performed throughout this thesis, data in only 34 languages and six different character encodings was available. Testing HAIL with more languages would be highly desirable, as in its current form it can support up to 511 languages and character encoding pairs. While some projections were made in this thesis as to how additional languages would affect HAIL, experimental evidence would be preferred. This does not, however, hinder an FPGA implementation. For instance, if further experimentation revealed that a different length of n-gram was more appropriate for a larger (or different) set of languages, changing the n-gram length is trivial in difficulty and simple to re-implement for an FPGA.

Appendix A

HAIL Files

Source files for the implementation of HAIL can be found in the `HAIL` project within the Reconfigurable Network Group's Concurrent Versions System (CVS) repository or within the Reconfigurable Network Group's internal web page at the following URL:

`http://www.arl.wustl.edu/projects/fpx/fpx_internal/HAIL/hail.tar.gz`

Upon downloading this file, decompress and extract it using the `tar` archiving utility [60] . A folder called `HAIL` will be extracted, which contains files needed to run the HAIL software and create a HAIL implementation for the FPX platform.

A.1 Directory Structure

The structure of the HAIL project is shown in Figure A.1. It contains five directories and multiple subdirectories. The `vhdl` directory contains all of the VHDL source files for the project. Subdirectories within `vhdl` contain files that implement the TCPLite wrapper, communication wrapper, SRAM controller, and SDRAM controller. The `sim` directory contains files needed for simulating the design in digital verification software such as ModelSim [51]. It contains two subdirectories: The `Testbench` directory contains files required for simulation, while the `work` directory is used to store compiled VHDL source files. The `syn` directory contains a project file for synthesizing the VHDL design with Synplicity tools [72]. It contains a subdirectory called `language_app`, which holds files needed to build the FPGA with Xilinx ISE tools [82]. The `software` directory contains source files for the HAIL simulation software. It contains a subdirectory called `control` which contains source files for software that communicates with the FPX circuit implementing HAIL. Both sets of software tools are described in detail in Appendix B. Finally, the `corpus` directory contains the data files used for experiments throughout this thesis.

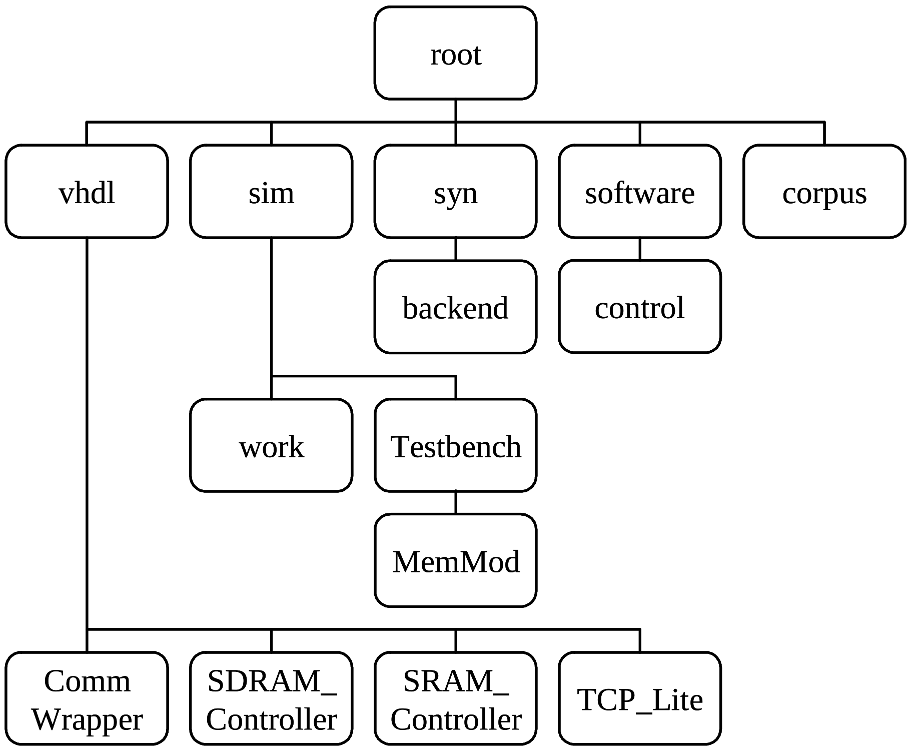


Figure A.1: The structure of the HAIL project directory

A detailed list of the all important data and source files within the directories are displayed in a structured list below.

- `vhdl`: This directory contains VHDL files for the implementation of HAIL described in Section 4.3. Relevant contents are listed below.
  - `input_buffer.vhd`: This file contains the VHDL implementation of the input buffer and control unit described in Section 4.3.1.
  - `ngram_crc.vhd`: This file contains an implementation of the Ethernet CRC, which is shown in Equations 4.1, 4.2, and 4.3. It takes as input an extracted n-gram, performs the CRC, and outputs a 19-bit SRAM address.
  - `ngram_hasher.vhd`: This file contains the VHDL model of the shift register discussed in Section 4.3.2 and instantiates the two n-gram hash units.

  - `sram_reader.vhd`: This file contains a simple pipeline to propagate control signals while SRAM is being read. From a functional standpoint, it is effectively part of the hash unit described in Section 4.3.3.
  - `trend_register.vhd`: This file contains the VHDL implementation of a single trend register, which is described in Section 4.3.4.
  - `count_score_unit.vhd`: This file instantiates four trend registers and contains the VHDL implementation of the comparator chain detailed in Section 4.3.4.
  - `language_module.vhd`: This is a file that instantiates the n-gram extractor, SRAM reader, and trend/score units and ties their signals together.
  - `control_processor.vhd`: This file contains the VHDL model of the control processor and SRAM programmer described in Section 4.3.5.
  - `report_generator.vhd`: This file contains the VHDL implementation of the report generator detailed in Section 4.3.6.
  - `fifo_AxB.vhd`: There are several different VHDL files implementing FIFO queues used in the design as buffers. In the actual file names, `A` and `B` are numbers; `A` represents the depth of the FIFO, and `B` represents the width of the FIFO, in bits. FIFOs were created using the Xilinx CORE Generator, part of the Xilinx ISE package.
  - `rad.vhd`: This is the top-level VHDL file for the HAIL implementation. It connects the input and output pins of the FPGA to signals within the circuit.
  - `TCP_Lite`: This subdirectory contains VHDL files needed to implement the TCPLite wrapper, which was described in Section 4.2.1.
  - `CommWraper`: This subdirectory contains VHDL files needed to implement the communication wrapper, which was described in Section 4.2.2.
  - `SDRAM_Controller`: This subdirectory contains VHDL files needed to implement the controller for SDRAM, which holds flow-specific state information.
  - `SRAM_Controller`: This subdirectory contains the VHDL file needed to implement the controller for SRAM, which contains n-gram tables.

- `sim`: This directory contains files that can be used to simulate the implementation of HAIL using digital verification software such as Modelsim [51]. Relevant contents are listed below.

  - INPUT_CELLS.DAT: This file contains control packet data used to program SRAM during simulation. The data was created from the set of documents in the Corpus path.
  - INPUT_CELLS_LC.DAT: This file contains network traffic to run through the simulation.
  - Makefile: This is a script used to compile the VHDL files and simulate the HAIL design. It is described in detail in Section B.5.
  - work: This directory is used to store the results of compiling the VHDL source files.
  - Testbench: This path contains VHDL files needed to read INPUT_CELLS.DAT and INPUT_CELLS_LC.DAT, and to create output files during simulation.
    - MemModel: This path contains files needed to simulate the SRAM and SDRAM modules connected to the FPX.

- syn: This directory contains files needed to implement HAIL on an FPGA.
  - language_app.prj: This is a project file for Synplicity, required to synthesize the HAIL design.
  - Makefile: This is a script used to synthesize and build the HAIL design. It is described in detail in Section B.5.
  - language_app: This directory contains files needed for building the HAIL FPGA implementation using Xilinx ISE tools.
    - *.edn: All .edn files contain synthesized descriptions of FIFOs created using the Xilinx CORE Generator. These are used during the building of the FPGA implementation.
    - bitgen.ut: This is a constraint file used during the building of the FPGA implementation.
    - fpx.ucf: This file contains a mapping of pins on the FPGA to top-level signals inside the rad.vhd file contained in the vhdl path. It is used during the building of the FPGA implementation.

- software: This directory contains source files for the HAIL software implementation, which is described in detail in Appendix B.
  - hail.cpp: This file contains directory parsing and hash functions shared by several different HAIL software components.
  - hail_config.cpp: This file contains functions for the HAIL software configuration tool, detailed in Section B.1.

- `hail_main.cpp`: This is the top-level file for the HAIL simulation program, detailed in Section B.3. It contains functions to read command-line options and create output.
- `hail_xml.cpp`: This file reads data from a file containing the output of the HAIL configuration program.
- `hail_train.cpp`: This file contains functions needed for training the HAIL system.
- `hail_ngram.cpp`: This file reads data from a file containing the results of training to use during experiments.
- `hail_test.cpp`: This file contains functions needed for simulating the behavior of HAIL in software.
- `hail_modelsim.cpp`: This file outputs the results of training to `INPUT_CELLS.DAT`, one of the two input files used by the aforementioned ModelSim simulation.
- `hail_parser.cpp`: This file contains functions for the document parsing tool, detailed in Section B.2.
- `Makefile`: This is a script used to compile the source files in this directory. It is described in detail in Section B.5.
- `control`: This directory contains source files for software that communicates with the FPX implementation of HAIL.
  * `catcher.c`: This file contains the main program that spawns threads for the control program and calls functions to receive packets from hardware.
  * `convertXML.c`: This file calls functions to convert binary data received from the hardware into XML files.
  * `hail.c`: This file receives data sent to the control software from the FPX implementation of HAIL and dispatches it to other functions.
  * `hail_config_rd.c`: This is an XML parser that is used to load SRAM on the FPX implementation of HAIL. It reads data from the file described in Section B.6.2.
  * `hail_output_rd.c`: This is an XML parser that converts the control software's XML output files (see Section B.6.4) into delimited files for easy importation into a spreadsheet.
  * `hailXML.c`: This file contains functions called by `convertXML.c`.
  * `hardware.c`: This file contains functions that check for the presence of files required to load SRAM on the FPX.

        * `loadtable.c`: This file calls functions to send data from the appropriate configuration files to hardware.
        * `monitor.c`: This file contains functions to keep track of metrics such as the number of packets received.
        * `utils.c`: This file contains miscellaneous functions to perform tasks such as computing flow identifiers and converting a string of hexadecimal characters to a decimal integer.
        * `queue.c`: This file performs functions related to a queuing mechanism that distributes data to the HAIL output processing functions. It can also distribute data to other processing functions if necessary.
        * `sendHail.c`: Functions in this file are called from `loadtable.c` and send data to load SRAM on the FPX implementation of HAIL.
        * `sendRM.c`: This file contains functions to check the status of the reporting module.
        * `server.c`: This file contains functions to receive UDP packets from the reporting module and place the message into the appropriate queue.
        * `signal.c`: This file contains functions to handle a Ctrl-C (kill) signal by closing all sockets and queues and querying the reporting module for a final set of metrics.
        * `Makefile`: This is a script used to compile the source files in this directory. It is described in detail in Section B.5.

- `corpus`: This directory contains the text documents used in the experiments described in this thesis.
    - `corpus.tar.gz`: This is a compressed archive file containing the documents used for experiments. The files within the archive are organized by language into subdirectories.

A.2 Corpus

The 1.7 MB of files in the `corpus` directory are divided into 34 different subdirectories. Each subdirectory is named with the language of the documents contained within it.

Table A.1: Languages and character encodings used in data set

Language	**Character Encoding**		**Language**	**Character Encoding**
Albanian	ISO 8859-1		Malaysian	ISO 8859-1
Arabic	ISO 8859-1		Nepali	UTF-8
Bulgarian	Windows 1251		Norwegian	ISO 8859-1
Czech	ISO 8859-1		Pashto	UTF-8
English	ISO 8859-1		Persian	UTF-8
Estonian	ISO 8859-1		Polish	Windows 1250
French	ISO 8859-1		Portuguese	ISO 8859-1
German	ISO 8859-1		Romanian	Windows 1250
Greek	ISO 8859-7		Russian	ISO 8859-5
Hausa	ISO 8859-1		Serbian	ISO 8859-1
Hindi	UTF-8		Spanish	ISO 8859-1
Indonesian	ISO 8859-1		Swedish	ISO 8859-1
Italian	ISO 8859-1		Tamil	UTF-8
Kazakh	UTF-8		Thai	UTF-8
Kirundi	ISO 8859-1		Turkish	ISO 8859-1
Kyrgyz	UTF-8		Urdu	UTF-8
Lithuanian	ISO 8859-1		Uzbek	UTF-8

Table A.2: Source of languages used in data set

Language	**Source**		**Language**	**Source**
Albanian	LDC ECI/MCI [25]		Malaysian	LDC ECI/MCI
Arabic	LDC English/Arabic [26]		Nepali	BBC Nepali [14]
Bulgarian	BBC Bulgarian [7]		Norwegian	LDC ECI/MCI
Czech	LDC ECI/MCI		Pashto	BBC Pashto [15]
English	LDC ECI/MCI		Persian	BBC Persian [16]
Estonian	LDC ECI/MCI		Polish	BBC Polska [17]
French	LDC ECI/MCI		Portuguese	LDC ECI/MCI
German	LDC ECI/MCI		Romanian	BBC Romanian [18]
Greek	LDC ECI/MCI		Russian	LDC ECI/MCI
Hausa	BBC Hausa [9]		Serbian	LDC ECI/MCI
Hindi	BBC Hindi [10]		Spanish	LDC ECI/MCI
Indonesian	BBC Indonesia [11]		Swedish	LDC ECI/MCI
Italian	LDC ECI/MCI		Tamil	BBC Tamil [19]
Kazakh	BBC Kazakh [12]		Thai	BBC Thai [20]
Kirundi	BBC Great Lakes [8]		Turkish	LDC ECI/MCI
Kyrgyz	BBC Kyrgyz [13]		Urdu	BBC Urdu [21]
Lithuanian	LDC ECI/MCI		Uzbek	BBC Uzbek [22]

Appendix B

HAIL Software

Before the VHDL implementation of HAIL was written, a simulation of the hardware's functionality, along with several other supporting programs, were created in in the C++ programming language.

Rather than performing analysis on live network data, the HAIL software operates on files stored on a hard disk or other storage media. This decision was made to allow a degree of control over the data to be analyzed. If experiments are performed on data in which the true language is known, then the true language can be compared to the output of HAIL to determine the accuracy of the algorithm. By measuring the accuracy as it responds to various parameter modifications, an optimal configuration can be determined. Indeed, this is the process used during the configuration of parameters detailed in Chapter 3.

The following sections detail the operation of the HAIL simulation software, as well as the programs designed to support it. Before executing any of these programs, they must be downloaded and compiled as previously described in Appendix A, and the directory must be switched to `software`.

B.1 HAIL Software Configuration

The HAIL simulation software must read the names and languages of training and testing files from an XML configuration file. This file can be quite large and must follow a particular format. A HAIL configuration program has been created to automatically generate this XML file from a simple input file. This program can be executed with the command `./hail_config`. There are several options that can be specified at the command line as flags. All of these are optional, and detailed below:

- `-r textfile`: This flag is used to specify the text file used as input by the configuration program. The `textfile` parameter must be the path of an input file formatted as described below. If not specified, this parameter will be set to `hailtrain.txt`.
- `-d directory`: This flag is used to specify the directory in which to place files to be used for the experiment. The `directory` parameter specifies this path; if it does not exist, it will be created. Any files existing in this path before program execution will be deleted. If not specified, this parameter will be set to `./Documents`.
- `-o output`: This flag is used to specify the file output by the configuration program. The `output` parameter specifies this file, which will be an XML file used as input by the language identification program. If not specified, this parameter will be set to `hailconfig.xml`.

The file accepted as input by this program must consist of a series of lines, which are each formatted as follows:

```
[Path] [Language] [Encoding] [Kilobytes] [Identifier]
```

The `Path` parameter must be a directory that contains files in a particular language/character encoding pair. The `Language` and `Encoding` parameters specify the respective language and character encoding of the files in this directory. These parameters can be any combination of letters, numbers, and symbols except for spaces. The `Kilobytes` parameter indicates the number of kilobytes of data from this path that will be used as training. All data in this path that is not used for training will be used for testing. Finally, the `Identifier` parameter must be a positive integer that acts as a language ID for the aforementioned language and character encoding pair. Currently, the identifier must be a positive integer between 1 and 511; any modification to the range of language identifiers will require a change in the code.

B.2 Text Partitioning Tool

For purposes of evaluating optimal parameters for the language identification, certain types of documents are inappropriate. Extremely long documents provide a very large number n-gram samples, so that the language and character encoding are almost certain to be correctly identified. Conversely, extremely short documents provide only a few n-gram samples, and are quite difficult to identify correctly. When attempting to gauge optimal parameters, medium-length documents are more useful.

A text partitioning tool was created to break a set of documents into documents of roughly equal length. The program works by combining all documents within a directory into one large string, then evenly splitting the string to create documents of a user-specified size. The text partitioning tool can be executed with the command `./hail_parse`. Two command-line options must be specified at runtime, and are detailed below:

- `-p path`: This option specifies the path containing files to be parsed. The `path` parameter must be a valid path containing documents to be parsed.
- `-s size`: This option specifies the size of the partitions to create. The `size` parameter must be a positive integer, representing the desired size of the partitioned documents, in bytes.

All files in the directory specified by `path` will be moved into a directory with the same name as `path`, except the suffix `_old` will be appended. The files output by the partitioning tool are guaranteed to be at least `size` bytes in length. If the total size of the input files is not evenly divisible by `size`, the size of the files will be increased slightly to accommodate the remaining data.

Note that the text partitioning tool does not split files across sentence or word boundaries; thus, many of the files created by the text partitioning tool will contain fragments of words and/or incomplete sentences at the beginning and end. This has no effect on language identification, as HAIL simply examines fixed-length portions of data; it does not rely on the structure of sentences or boundaries of words.

B.3 Language Identification Software

The HAIL simulation software executes the same algorithm performed by the HAIL hardware described in Chapter 4. However, many parameters of the software can be changed via command line options. This degree of flexibility is not possible using the HAIL hardware.

The software version of HAIL can be executed with the command `./hail`. There are several options that can be specified at the command line as flags. Most of these flags are optional, although some are required. The options are detailed below:

- `-r`: This flag causes the program to train HAIL before performing any tests. The files used for training are obtained from the file specified with the `-i` option, which is outlined below. The option to train HAIL is disabled by default.
- `-s`: This flag causes the program to perform an experiment on the testing set listed in the file specified with the `-i` option. The option to perform a test is disabled by default. Either `-r`, `-s`, or both must be specified in order for the software to run.
- `-m`: This flag is used to output training results in a form usable by the VHDL testbench for ModelSim simulations of HAIL.
- `-n size`: This flag is used to set the length of n-grams used in training and/or testing. The parameter `size` must be a positive integer, and specifies the length of n-grams in bytes. This parameter will affect both training and testing. If not specified, `size` will be set to 5.
- `-w width`: This flag is used to specify the width of the n-gram lookup table. The parameter `width` must be a positive integer, and specifies the width of the table measured in the number of n-grams stored at each memory location. This parameter will affect both training and testing. If not specified, `width` will be set to 4.
- `-j jump`: This flag is used to specify the number of n-grams sampled. The parameter `jump` must be a positive integer, and specifies the size of the "jump" taken by the parser after sampling an n-gram. Consequently, 1/`jump` is the ratio of n-grams that will be sampled. This parameter will only affect testing. If not specified, `jump` will be set to 2.
- `-t trend`: This flag is used to specify the depth of the trend register. The parameter `trend` must be a positive integer, and specifies the number of sequential n-grams from one language that must appear before a language is granted a permanent counter. This parameter will only affect testing. If not specified, `trend` will be set to 3.
- `-c counter`: This flag is used to specify the number of permanent counters in the system. The parameter `counter` must be a positive integer, and specifies

the number of counters used by each SRAM slot. Consequently, `width*jump` is the total number of counters used by the system. This parameter will only affect testing. If not specified, `counter` will be set to 4.

- `-i inputfile`: This flag is used to specify the HAIL configuration file. The parameter `inputfile` must be the path of an XML file created by the `hail_config` program described in Section B.1. This parameter will affect both training and testing. If not specified, `inputfile` will be set to `hailconfig.xml`.
- `-f ngramfile`: This flag is used to specify the n-gram file. The parameter `ngramfile` must be a file name. During training, the n-gram lookup table is output to this file (which is created if it does not exist); during testing, the n-gram lookup table is read from it. If not specified, `ngramfile` will be set to `ngram.xml`.
- `-o outputfile`: This flag is used to specify the output file. The parameter `outputfile` must be a file name. During testing, the results of file processing is stored in this file, which is created if it does not exist. If not specified, `outputfile` will be set to `output.xml`.

B.4 Control Software

The control software is, in fact, three distinct programs: The `pingRM` program, which configures the reporting module; the `catcher` program, which sends data to configure the FPX applications and receives their output; and the `convertXML` director, which converts binary output from the applications into XML files.

These programs were created to accommodate any FPX application with any output format. This section will only describe how to configure HAIL and supporting infrastructure circuits (the TCP processor and reporting module mentioned in Section 4.2). To execute these programs, one must first switch to the `control` subdirectory within the `software` path.

The `pingRM` program is used to ensure that the reporting module is active and informs the module of what ports to use. It is executed with the following command:

```
./pingRM [applications]
```

The `applications` parameter is the list of applications that are being used. The ports associated with each application must be hard-coded into the `pingRM` source files. In order to configure the reporting module for the TCP processor and the HAIL circuit, the following command must be issued:

```
./pingRM HT
```

In this command, `H` represents HAIL while `T` represents the TCP processor.

The `catcher` program is used to send data, such as HAIL's hash table, to the individual FPX applications. It is executed with the following format:

```
./catcher -c configDir -o outputDir
```

In this command, `configDir` is the directory containing data files that are to be parsed and sent to the FPX applications. This directory must exist and contain all necessary files. `outputDir` is the directory into which binary output from the catcher will be placed. This directory must also exist.

The final program, `convertXML`, is used to convert the catcher's binary output into XML files which can be read easily by programs designed to parse text data. It is executed with the following command:

```
./convertXML -h directory
```

The `directory` parameter is the directory from which binary data is read (this will typically be the `outputDir` of the `catcher`) and to which XML data is written. The format of this XML data is shown in Section B.6.4.

B.5 Makefiles

The HAIL project contains numerous scripts for the compiling of software, creation of hardware simulations and building of hardware. These scripts are in the form of makefiles [59]. A total of four makefiles are located inside the HAIL project. The location of each makefile and the commands performed by each are listed below.

In order for the scripts to execute properly, various tools must be installed on the computer system. These tools include the aforementioned make utility, a C++ compiler such as G++ [74], ModelSim [51], Synplify Pro [73] and Xilinx ISE [81]. For functions in the `control` directory, additional software is also required:

To program bitfiles onto the FPX cards in this stacked configuration, the Networked Configurable Hardware Administrator for Reconfiguration and Governing via End-systems (NCHARGE) must be properly installed on the system [71]. NCHARGE is a software program designed to communicate with the NID device on the FPX platform and can perform a variety of tasks such as configuring routing between interfaces on the card, checking the status of the FPX and programming bitfiles onto the RAD. NCHARGE can be compiled and run on Unix and Linux platforms, as well as on Windows platforms through Cygwin, "a Linux-like environment for Windows" [63].

In order to perform experiments with data stored on a disk rather than life network traffic, tcpreplay [70] must be installed. This is an open-source tool that allows captured network traffic to be re-sent on a network link.

- `/software`: This makefile is used to compile the various programs described in Sections B.1 through B.3.
  - `make config`: This command will compile the HAIL configuration program described in Section B.1.
  - `make parse`: This command will compile the text partitioning tool described in Section B.2.
  - `make hail`: This command will compile the language identification software described in Section B.3.
  - `make all`: This command will compile the HAIL configuration program, text partitioning tool and language identification software.
  - `make clean`: This command will clean the `software` directory of compiled programs and various temporary files.
  - `/software/control`: This makefile is used to compile the various programs described in Section B.4.
    - `make catcher`: This command will compile the catcher, the part of the control software that sends data to program SRAM tables on HAIL and receives information from the system.
    - `make convertXML`: This command compiles a program to convert binary data received from the hardware system into XML files.
    - `make pingRM`: This command compiles a program to configure the reporting module.
    - `make clean`: This command will clean the `catcher` directory of compiled programs and various temporary files.

- `/sim`: This makefile is used to compile the VHDL source files described in Section A.1 and to perform simulations of hardware.
  - `make compile`: This command will compile VHDL source files in the `vhdl` directory so that simulations of hardware can be performed.
  - `make sim`: This command will simulate the compiled VHDL design with ModelSim. Input for programming the SRAM tables will be read from `INPUT_CELLS.DAT` while input for language identification will be read from `INPUT_CELLS_LC.DAT`.
  - `make clean`: This command will clean the `vhdl` and `sim` directories of compiled VHDL and various temporary files.
- `/syn`: This makefile is used to synthesize, place and route the VHDL source files in the `vhdl` directory to create a bitfile for loading into the FPGA.
  - `make build`: This command will use Synplicity to synthesize the VHDL source files, then place and route the synthesis output using Xilinx software.

B.6 XML file formats

XML files are used to store data used by different software components. XML was used as a storage format because it is a well-defined standard that provides an easy-to-read structure and because tools exist to parse XML data [27].

Four XML files are used by the HAIL software tools. The first, `hailconfig.xml`, is output by the HAIL software configuration program (Section B.1). This file contains a list of training and testing files to be used by the language identification software described in Section B.3. It also contains a list that maps numerical language identifers to specific language and character encoding pairs.

The second XML file, called `ngram.xml`, is output after training is performed by the language identification software. It is used by both the testing portion of the language identification software as well as the control software. It contains a list of memory locations and the language identifiers stored at each one. Omitted memory addresses are implied to contain no language identifiers.

The third XML file, `output.xml`, is output after testing is performed by the language identification software. It contains details of the testing run, a mapping of language identifiers to specific language and character encoding pairs, and lists indicating which files were classified correctly and incorrectly during testing.

The fourth XML file format is output by the control software described in Section B.4. As the FPX implementation of HAIL outputs data on a flow-by-flow basis, it is received by the control software and output into an XML file. One file is created per TCP flow (or UDP packet) and contains information about the flow or packet, as well as the languages identified during the processing of the document.

The structure of each XML file is detailed below.

B.6.1 hailconfig.xml

The `hailconfig.xml` file is the output of the HAIL configuration software detailed in Section B.1. The first line is an XML declaration which identifies the version of XML used in the file, the character encoding, and notes that the file contains no external dependencies. The second line contains data indicating that the file is a HAIL configuration file and was generated by version 1.5 of HAIL (the version described in this thesis).

Subsequent sections identify the mapping of numerical language identifiers to language and character encoding pairs, a list of files to be used for training and a list of files to be used for testing in software. File paths shown in the example are relative to the path in which the software was executed.

An example of the file format for the `hailconfig.xml` file follows.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<HAILFile>
  <MetaData File_Type="HAIL" version="1.5"/>

  <Mapping num = "2">
    <identifier language="Indonesian" encoding="ISO8859-1">12</identifier>
    <identifier language="Thai" encoding="UTF-8">31</identifier>
  </Mapping>

  <TrainFiles num = "2">
    <file identifier="12">../corpus/indonesian_iso8859/file_1.fil</file>
    <file identifier="31">../corpus/thai_utf8/file_3.fil</file>
  </TrainFiles>

  <TestFiles num = "3">
    <file identifier="12">../corpus/indonesian_iso8859/file_2.fil</file>
    <file identifier="31">../corpus/thai_utf8/file_1.fil</file>
    <file identifier="31">../corpus/thai_utf8/file_2.fil</file>
  </TestFiles>

</HAILFile>
```

B.6.2 ngram.xml

The `ngram.xml` file is the output of training performed by the software detailed in Section B.3. It contains the same XML declaration and meta data tag as the other XML files presented in this section.

The remainder of the file contains a list of data to be placed in the hash table used to store the languages associated with n-grams. This section is prefaced with a tag indicating the number of address bits in the hash table and the number of language identifiers stored at each memory location.

The file then indicates the language identifiers stored at each position in each hash table location. The language identifier 0 is reserved to indicate that no language was suitable to be stored at that memory location. Memory locations not listed in this section of the file can be assumed to contain 0 at every location.

An example of the file format for the `ngram.xml` file follows.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<NGramFile>
  <MetaData File_Type="HAIL" version="1.5"/>

  <Memory addressbits="19" width="4">
    <location address="1">
      <slot0>0</slot0>
      <slot1>12</slot1>
      <slot2>0</slot2>
      <slot3>0</slot3>
    </location>
    <location address="9">
      <slot0>0</slot0>
      <slot1>12</slot1>
      <slot2>0</slot2>
      <slot3>31</slot3>
    </location>
  </Memory>

</NGramFile>
```

B.6.3 output.xml

The `output.xml` file is the output of testing performed by the software detailed in Section B.3. It contains the same XML declaration and meta data tag as the other XML files presented in this section.

The file also contains information about the software test, including the n-gram size used, the ratio of n-grams sampled, the number of slots per memory address, the trend depth and the number of permanent counters per memory slot. It also contains information about the files used during training and testing, including the number of languages used in the test, the number of training files and the number of testing files. The file also contains a summary of the results from the testing, including the latency, number of files correctly classified, number of files incorrectly classified and overall accuracy. The mapping of language and character encoding pairs to numeric language identifiers is also provided, so that the file can be read without the supporting `hailconfig.xml` file.

The remainder of the file includes a list of the files that were incorrectly classified along with their actual language identifiers and the language they were identified as. Following this is a list of correctly-classified files along with their language identifiers.

An example of the file format for the `output.xml` file follows.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

```
<HAILResultFile>
  <MetaData File_Type="HAIL" version="1.5"/>

  <Run>
    <NGramSize>4</NGramSize>
    <SamplingRatio>1.000000</SamplingRatio>
    <MemoryWidth>4</MemoryWidth>
    <TrendDepth>2</TrendDepth>
    <CountersPerSlot>4</CountersPerSlot>
  </Run>

  <FileData>
    <NumLanguages>2</NumLanguages>
    <NumTrainFiles>2</NumTrainFiles>
    <NumTestFiles>3</NumTestFiles>
  </FileData>

  <ResultSummary>
    <AverageTime>9.709959</AverageTime>
    <NumberCorrect>2</NumberCorrect>
    <NumberIncorrect>1</NumberIncorrect>
    <Accuracy>66.666667</Accuracy>
  </ResultSummary>

  <Mapping num = "2">
    <identifier language="Indonesian" encoding="ISO8859-1">12</identifier>
    <identifier language="Thai" encoding="UTF-8">31</identifier>
  </Mapping>

  <Incorrect num = "1">
    <file trueid="12" hailid="0">../corpus/indonesian_iso8859/file_2.fil</file>
  </Incorrect>

  <Correct num = "2">
    <file trueid="31" hailid="31">../corpus/thai_utf8/file_1.fil</file>
    <file trueid="31" hailid="31">../corpus/thai_utf8/file_2.fil</file>
  </Correct>
```

```
</HAILResultFile>
```

B.6.4 Hardware output format

The control software detailed in Section B.4 receives data from the report generator found in the FPGA implementation of HAIL. The software can output XML files for each TCP flow (or UDP packet) received from the system. These files contain the same XML declaration and meta data tag as the other XML files presented in this section.

The files also contain flow statistics that can be used to uniquely identify the TCP flow, including the source and destination IP addresses and the source and destination ports.

The files also contain the contents of the permanent counters at the time that the flow terminated. The counters are simply output in the order that they are accessed by the FPGA circuit and are therefore not sorted into any order. The information provided includes both the language identifier associated with the counter and the final value of the counter.

An example of the file format for these files follows. While the hardware implementation presented in this thesis uses more than two permanent counters, the example shows only two permanent counters for simplicity.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<HAILOutputFile>
  <MetaData File_Type="HAIL" version="1.5"/>

  <FlowStatistics>
    <SourceIP>128.252.153.79</SourceIP>
    <DestIP>72.14.203.104</DestIP>
    <SourcePort>4815</SourcePort>
    <DestPort>80</DestPort>
  </FlowStatistics>

  <FlowLanguages>
    <language id="0">
```

```
      <identifier>0</identifier>
      <counter>0</counter>
    </language>
    <language id="1">
      <identifier>12</identifier>
      <counter>234</counter>
    </language>
  </FlowLanguages>

</HAILOutputFile>
```

Appendix C

Laboratory Configuration

An FPX card implementing HAIL can be placed within a GVS-1000, a chassis created by Global Velocity, Inc. [1] to store two stacks of FPX cards. The bottom cards in each stack are connected by a backplane that sits at the bottom of the GVS-1000 unit. A photograph of the GVS-1000, with several stacked FPX cards, is shown in Figure C.1.

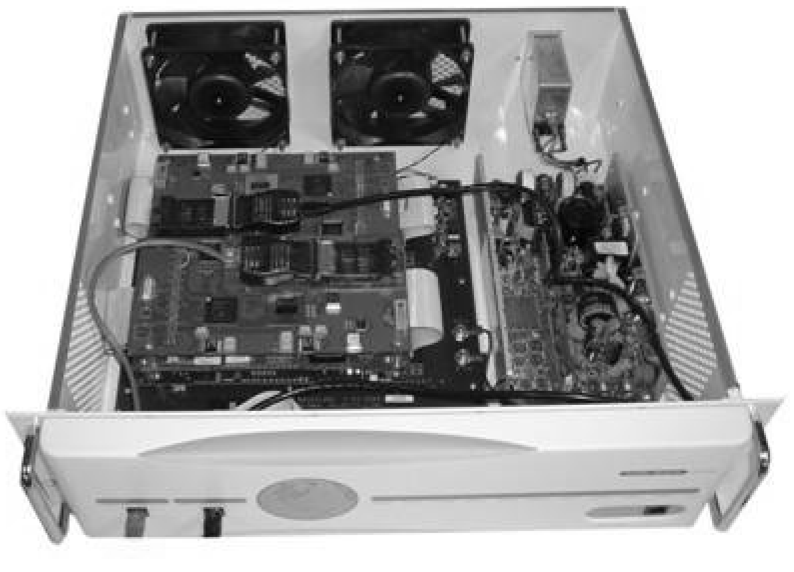


Figure C.1: The GVS-1000 containing stacked FPX cards

Using the GVS-1000, HAIL can be combined with the TCP processor and reporting module described in Sections 4.2.1 and 4.2.2, respectively. In the left-hand stack, the TCP processor receives data from a line card connected to a network link; it performs operations to deliver in-order TCP packets to HAIL, which receives the data and processes it for language identification.

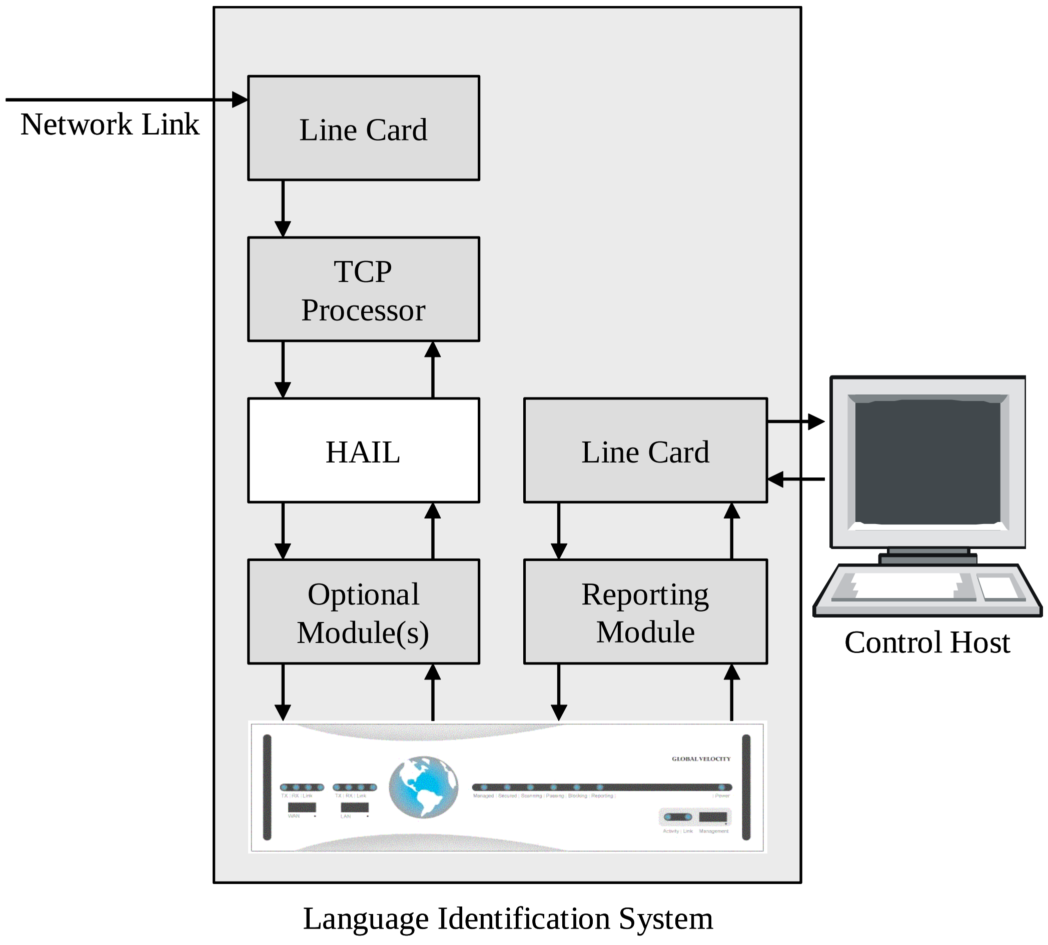


Figure C.2: Configuration of HAIL inside the GVS-1000

In the right-hand stack, the reporting module receives control data from a software host and sends the data to HAIL for programming SRAM and control bits. The reporting module also forwards the outputs of HAIL's report generator (see Section 4.3.6) through this line card interface. Optional FPX modules for further processing can be inserted between HAIL and the reporting module. In this configuration, HAIL inserts the best language identifier for a TCP flow into each packet from the flow. The stacked configuration is illustrated in Figure C.2.

The control software described in Section B.4 must be used in order to configure the cards placed inside the GVS-1000. Rather than requiring a user to enter all commands manually, a script called `hailscript` is provided within the `control` directory inside the HAIL distribution. This script performs a series of NCHARGE commands to place HAIL and the TCP processor on the FPX cards inside the GVS-1000, configures the reporting module, loads the hash table on the HAIL circuit and starts the `catcher` program.

Before running `hailscript`, the appropriate hardware must be configured. First, a computer must be configured with the software tools listed in Section B.5. Second, circuits must be loaded into a GVS-1000 as shown in Figure C.2. (The current script assumes that no "optional processing" cards alluded to in the figure will be in place.) Third, a network interface on the computer must be connected to the appropriate line card as seen in Figure C.2 via an appropriate type of network link. Finally, the second line card in the GVS-1000 must either be connected to a network tap (to process live data) or to a second network interface on a computer (to process stored data sent via `tcpreplay`).

C.1 HAIL script contents

This section explains the contents of `hailscript` in detail. Comments are provided to explain the function of each line in the script.

Kill all NCHARGE processes currently running.

```
killall ncharge
```

Launch an NCHARGE process for communicating with reporting module. 192.168.50.2 is the default IP address of the line card in the GVS-1000.

```
ncharge -i 192.168.50.2 -p 0 -s 6 -r &
```

Launch a process for communicating with HAIL.

```
ncharge -i 192.168.50.2 -p 0 -s 1 -r &
```

Launch a process for communicating with TCP processor.

```
ncharge -i 192.168.50.2 -p 0 -s 0 -r &
```

Load the reporting module bitfile into the appropriate FPX. `basic_send` is a program that sends commands to FPX cards via NCHARGE processes running in the background.

```
basic_send 0.6 c rm.bit
```

Configure the FPX card that will be loaded with HAIL to receive commands on VCI 0x24.

```
basic_send 0.0 e 2a
```

Configure the FPX card that will be loaded with HAIL to propagate data on VCI 0x28 onward.

```
basic_send 0.1 t 0 28 1 0 0 0
```

Configure the FPX card that will be loaded with HAIL to propagate data on VCI 0x35, which is control data for programming SRAM and setting flags in hardware, to the appropriate RAD input. VCI 0x35 is also used for output from the report generator; this command sends that data downwards.

```
basic_send 0.1 t 0 35 2 0 0 0
```

Configure the FPX card that will be loaded with HAIL to propagate data on VCI 0x39, which is network data from the TCP processor, to the appropriate RAD input.

```
basic_send 0.1 t 0 39 0 3 0 0
```

Configure the FPX card that will be loaded with HAIL to propagate data on VCI 0x2B, which is output from the TCP reserializer, downwards.

```
basic_send 0.1 t 0 2B 0 0 0 0
```

Load the HAIL bitfile into the appropriate FPX.

```
basic_send 0.1 c language_app.bit
```

Configure the FPX card that will be loaded with the TCP processor to receive commands on VCI 0x22.

```
basic_send 0.0 e 22
```

Configure the FPX card that will be loaded with the TCP processor to propagate data on VCI 0x33, which is network data from the line card, to the appropriate RAD input.

```
basic_send 0.0 t 0 33 0 2 0 0
```

Configure the FPX card that will be loaded with the TCP processor to propagate data on VCI 0x39, which is output, downard.

```
basic_send 0.0 t 0 39 0 0 0 0
```

Load the TCP processor bitfile into the appropriate FPX.

```
basic_send 0.0 c streamextract_4v2.bit
```

Inform the reporting module that the TCP processor and HAIL will be loaded into the GVS-1000.

```
sendRM HT
```

Start the catcher program, which will send data for loading HAIL's SRAM tables and catch data being output.

```
catcher -c ./hailconfig -o ./hailoutput
```

Running the `convertXML` program and sending network or file data to be processed by HAIL is left to the user.

Appendix D

Additional Figures

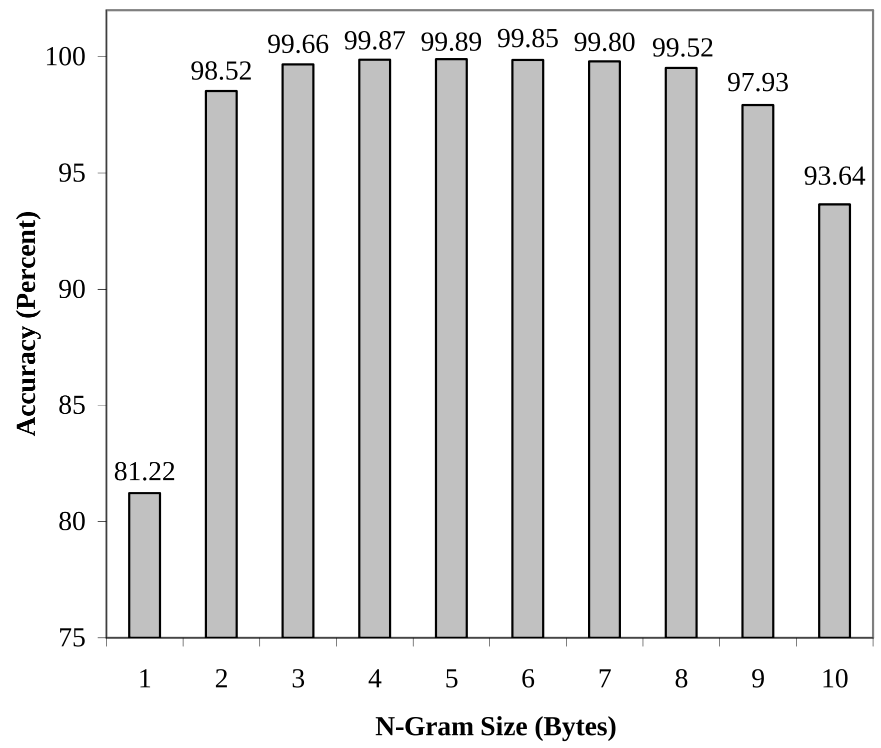


Figure D.1: The effect of n-gram size on accuracy as size is altered from one to ten bytes

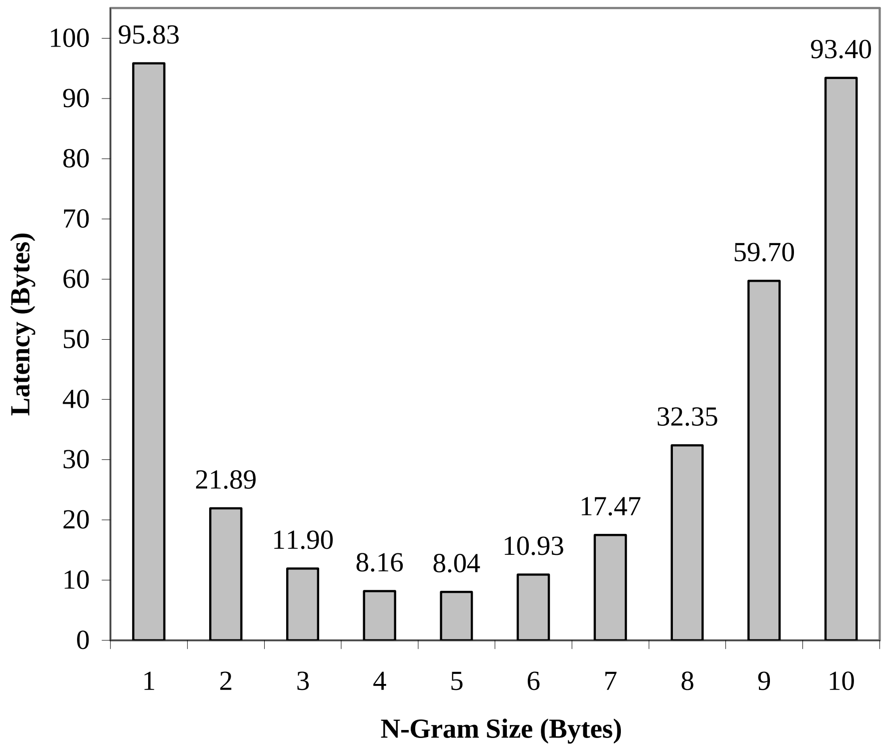


Figure D.2: The effect of n-gram size on latency as size is altered from one to ten bytes

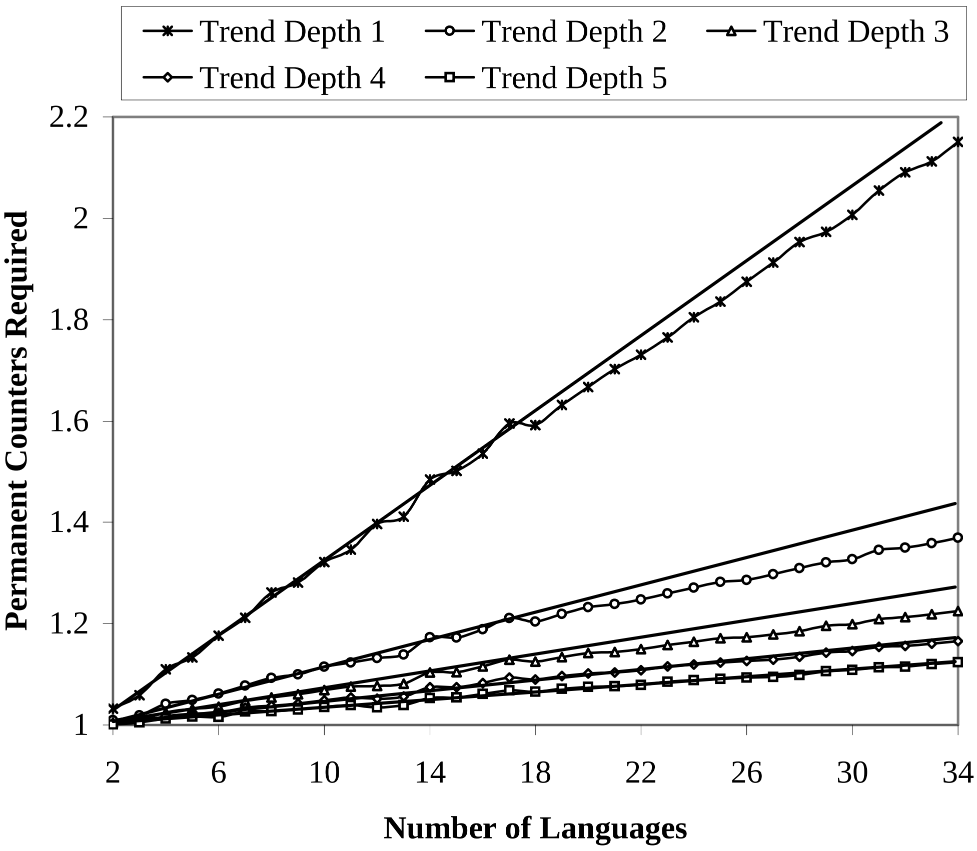


Figure D.3: The number of permanent counters required for varying trend depths as the number of languages is adjusted from two to 34

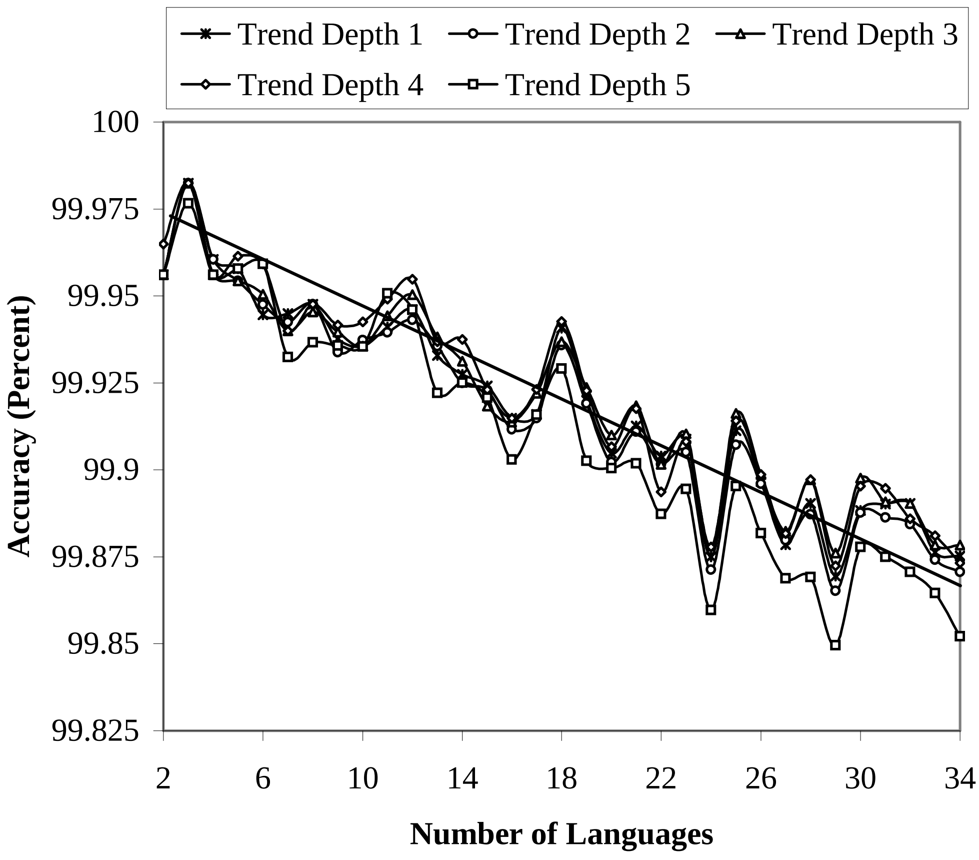


Figure D.4: The accuracy of varying trend depths as the number of languages is adjusted from two to 34

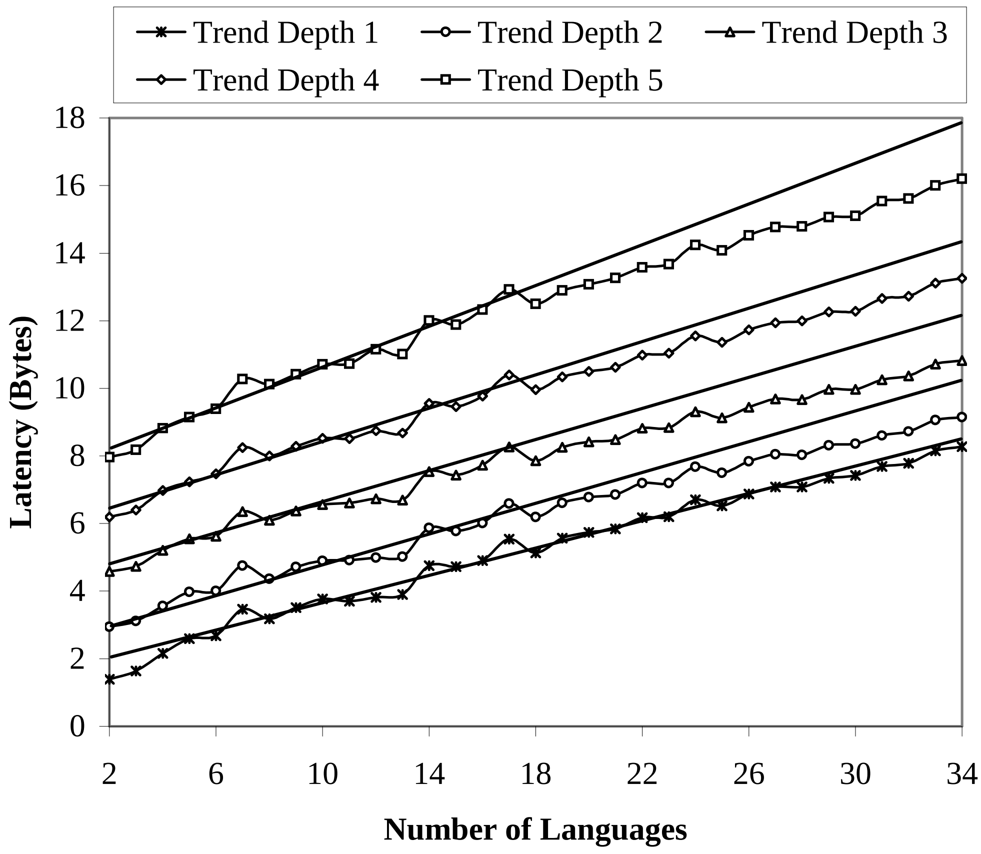


Figure D.5: The latency of varying trend depths as the number of languages is adjusted from two to 34

Appendix E

List of Acronyms

ASCII	American Standard Code for Information Interchange
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
CAM	Content Addressable Memory
CLB	Configurable Logic Block
CP	Code Page
CVS	Concurrent Versions System
DDR	Dual Data Rate
DRAM	Dynamic Random Access Memory
EPROM	Erasable Programmable Read-Only Memory
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPX	Field programmable Port eXtender
FSM	Finite State Machine
HAIL	Hardware-Accelerated Identification of Languages
HDL	Hardware Description Language
HEC	Header Error Correct
IANA	Internet Assigned Numbers Authority
IP	Internet Protocol
ISO	International Organization for Standardization
LUT	Look-Up Table
NID	Network Interface Device
OC	Optical Carrier
PC	Personal Computer
QDR	Quad Data Rate
RAD	Reprogrammable Application Device

RLDRAM	Reduced-Latency Dynamic Random Access Memory
RTL	Register-Transfer Level
SDRAM	Synchronous Dynamic Random Access Memory
SRAM	Synchronous Random Access Memory
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UTF	Unicode Transformation Format
UTOPIA	Universal Test and Operations Physical Interface for ATM
VCI	Virtual Circuit Identifier
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VISCII	Vietnamese Standard Code for Information Interchange
VPI	Virtual Path Identifier
ZBT	Zero Bus Turnaround

References

[1] Global Velocity web page. Online as: http://www.globalvelocity.com, 2005.

[2] UTOPIA bus description. Online as http://www.interfacebus.com/Design_-Connector_UTOPIA.html, March 2005.

[3] Altera, Inc. Selecting the right high-speed memory technology for your system. Online as: http://www.altera.com/literature/wp/wp_memoryselect.pdf.

[4] American National Standards Institute. eStandards Store. Online as: http://-webstore.ansi.org/ansidocstore/find.asp?find_spec~8859, 2005.

[5] Michael Attig. Architectures for rule processing intrusion detection and prevention systems. Masters Thesis, Washington University in St. Louis, April 2005.

[6] Thomas Bayes. Studies in the history of probability and statistics: Thomas bayes's essay towards solving a problem in the doctrine of chances. *Biometrika*, 45:296, 1958.

[7] British Broadcasting Corporation. BBC Bulgarian. Online as: http://www.-bbc.co.uk/bulgarian/, November 2005.

[8] British Broadcasting Corporation. BBC Great Lakes. Online as: http://www.-bbc.co.uk/greatlakes/, November 2005.

[9] British Broadcasting Corporation. BBC Hausa. Online as: http://www.bbc.-co.uk/hausa/, November 2005.

[10] British Broadcasting Corporation. BBC Hindi. Online as: http://www.bbc.-co.uk/hindi/, November 2005.

[11] British Broadcasting Corporation. BBC Indonesia. Online as: http://www.-bbc.co.uk/indonesian/, November 2005.

[12] British Broadcasting Corporation. BBC Kazakh. Online as: http://www.bbc.-co.uk/kazakh/, November 2005.

[13] British Broadcasting Corporation. BBC Kyrgyz. Online as: http://www.bbc.-co.uk/kyrgyz/, November 2005.

[14] British Broadcasting Corporation. BBC Nepali. Online as: `http://www.bbc.-co.uk/nepali/`, November 2005.

[15] British Broadcasting Corporation. BBC Pashto. Online as: `http://www.bbc.-co.uk/pashto/`, November 2005.

[16] British Broadcasting Corporation. BBC Persian. Online as: `http://www.bbc.-co.uk/persian/`, November 2005.

[17] British Broadcasting Corporation. BBC Polska. Online as: `http://www.bbc.-co.uk/polish/`, November 2005.

[18] British Broadcasting Corporation. BBC Romanian. Online as: `http://www.-bbc.co.uk/romanian/`, November 2005.

[19] British Broadcasting Corporation. BBC Tamil. Online as: `http://www.bbc.-co.uk/tamil/`, November 2005.

[20] British Broadcasting Corporation. BBC Thai. Online as: `http://www.bbc.-co.uk/thai/`, November 2005.

[21] British Broadcasting Corporation. BBC Urdu. Online as: `http://www.bbc.-co.uk/urdu/`, November 2005.

[22] British Broadcasting Corporation. BBC Uzbek. Online as: `http://www.bbc.-co.uk/uzbek/`, November 2005.

[23] William B. Cavnar and John M. Trenkle. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, page 161.

[24] Louis Comtet. *Advanced Combinatorics: The Art of Finite and Infinite Expansions*. Reidel Publishing Company, 1974.

[25] Linguistic Data Consortium. ECI multilingual text. Online as: `http://www.-ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC94T5`, 1994.

[26] Linguistic Data Consortium. Arabic English parallel news part 1. Online as: `http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=-LDC2004T18`, 2004.

[27] Clark Cooper. The Expat XML parser. Online as: `http://expat.-sourceforge.net/`.

[28] Thomas H. Cormen, Charles E. Leiseron, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[29] Roman Czyborra. Codepage & co. Online as: `http://aspell.net/charsets/-codepages.html`, June 1998.

[30] Roman Czyborra. The cyrillic charset soup. Online as: `http://aspell.net/-charsets/cyrillic.html`, November 1998.

[31] Roman Czyborra. Good ole' ASCII. Online as: `http://aspell.net/char-sets/iso646.html`, November 1998.

[32] Roman Czyborra. Traditional CJK charsets. Online as: `http://aspell.net/-charsets/cjk.html`, November 1998.

[33] Roman Czyborra. VISCII. Online as: `http://aspell.net/charsets/viet-namese.html`, June 1998.

[34] Marc Damashek. Method of retrieving documents that concern the same topic. U.S. Patent 5,418,951, September 1994.

[35] Marc Damashek. Gauging similarity with n-grams: Language independent categorization of text. *Science*, 267:843, February 1995.

[36] Arthur Dempster, Nan Laird, and Donald Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(1):1, 1977.

[37] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel Bloom filters. In *Hot Interconnects*, pages 44–51, Stanford, CA, August 2003.

[38] Easics, Inc. CRC tool. Online as: `http://www.easics.com/webtools/crctool`, November 2005.

[39] Global Reach. Global Internet statistics by language. Online: `http://www.-glreach.com/globstats/index.php3`, Dec. 2004.

[40] Global Reach. Global Internet statistics: Sources and references. Online: `http://www.glreach.com/globstats/refs.php3`, Dec. 2004.

[41] Juha Heinanen. Multiprotocol encapsulation over ATM adaptation layer 5. Internet Engineering Task Force: RFC 1483, July 1993.

[42] Steve Huffman. The genetic classification of languages by n-gram analysis: A computational technique. Doctoral Dissertation, Georgetown University, 1998.

[43] E. J. Johnson and A. Kunze. *IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Intel Press, 2003.

[44] Robert C. Paulsen Jr. and Michael J. Martino. Word counting natural language determination. U.S. Patent 6,704,698, August 1996.

[45] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671, 1983.

[46] Gokul Krishnan. Low-cost easypath FPGAs offer promise to ASSP companies. *Xcell Journal*, page 42.

[47] M. Laubach. Classical IP and ARP over ATM. Internet Engineering Task Force: RFC 1577, January 1994.

[48] John W Lockwood. An open platform for development of network processing modules in reprogrammable hardware. In *IEC DesignCon'01*, pages WB–19, Santa Clara, CA, January 2001.

[49] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor. Reprogrammable network packet processing on the field programmable port extender (FPX). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, February 2001.

[50] John W. Lockwood, Christopher Neely, Christopher Zuver, James Moscola, Sarang Dharmapurikar, and David Lim. An extensible, system-on-programmable-chip, content-aware Internet firewall. In *Field Programmable Logic and Applications (FPL)*, page 14B, Lisbon, Portugal, September 2003.

[51] Mentor Graphics. ModelSim Products. Online as: `http://www.model.com/-products/60/default.asp`, 2005.

[52] Microsoft Corporation. Code pages supported by Windows. Online as: `http://-www.microsoft.com/globaldev/reference/WinCP.mspx`, January 2006.

[53] J. Susan Milton and Jesse C. Arnold. *Introduction to Probability and Statistics*. McGraw-Hill Higher Education, 2003.

[54] James Moscola. FPgrep and FPsed: Packet payload processors for managing the flow of digital content on local area networks and the Internet. Masters Thesis, Washington University in St. Louis, July 2003.

[55] J. B. Paterson. DIS 8859-1, 8-bit single-byte coded graphic character sets part 1: Latin alphabet no. 1. Online as: `http://anubis.dkuug.dk/JTC1/SC2/WG3/-docs/n411.pdf`, February 1998.

[56] Larry Peterson and Bruce Davie. *Computer Networks: A Systems Approach.* Morgan Kaufmann Publishers Inc., 2002.

[57] J. Postel. DoD transmission control protocol. Internet Engineering Task Force: RFC 761, January 1980.

[58] Fletcher Pratt. *Secret and Urgent: The Story of Codes and Ciphers.* Aegean Park, 1996.

[59] GNU Project. GNU make. Online as: `http://www.gnu.org/software/make/`, 2004.

[60] GNU Project. Tar. Online as: `http://www.gnu.org/software/tar/`, 2005.

[61] Tenkasi Ramabadran and Sunil Gaitonde. A tutorial on CRC computations. *IEEE Micro*, 8(4):62, 1988.

[62] Jr. Raymond G. Gordon, editor. *Ethnologue: Languages of the World.* SIL International, 2005.

[63] Red Hat Cygwin. Cygwin information and installation. Online as: `http://-www.cygwin.com/`.

[64] H. Charles Romesburg. *Cluster Analysis for Researchers.* Lifetime Learning Publications, 1984.

[65] M.D. Salman, J.C. New, J.M. Scarlett, P.H. Kass, S. Hetts, and R. Ruch-Gallie. Human and animal factors related to the relinquishment of dogs and cats in 12 selected animal shelters in the usa. *Journal of Applied Animal Welfare Science*, 1(2):207–226, 1998.

[66] John C. Schmitt. Trigram-based method of language identification. U.S. Patent 5,062,143, February 1990.

[67] David Schuehler and John Lockwood. A modular system for FPGA-based TCP flow processing in high-speed networks. In *14th International Conference on Field Programmable Logic and Applications (FPL)*, pages 301–310, Antwerp, Belgium, August 2004.

[68] Stanislav Shalunov and Benjamin Teitelbaum. Bulk TCP use and performance on Internet2. In *ACM SIGCOMM Measurement Workshop*, August 2001.

[69] IEEE Computer Society. Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. Institute of Electrical and Electronics Engineers: Standard 802.3, March 2002.

[70] SourceForge.net. tcpreplay. Online as: `http://sourceforge.net/projects/-tcpreplay/`.

[71] Todd Sproull, John W. Lockwood, and David E. Taylor. Control and configuration software for a reconfigurable networking hardware platform. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, April 2002.

[72] Synplicity, Inc. Synplicity. Online as: `http://www.synplicity.com/`, 2005.

[73] Synplicity, Inc. Synplicity: Products: Synplify Pro. Online as: `http://www.-synplicity.com/products/synplifypro/index.html`, 2005.

[74] The GCC Team. GCC home page. Online as: `http://gcc.gnu.org/`, 2005.

[75] The Internet Assigned Numbers Authority. Character sets. Online as: `http://-www.iana.org/assignments/character-sets`, January 2005.

[76] The Unicode Consortium. Technical introduction. Online as: `http://www.-unicode.org/standard/principles.html`, September 2004.

[77] The Unicode Consortium. FAQ - basic questions. Online as: `http://www.-unicode.org/faq/basic_q.html`, February 2005.

[78] The Unicode Consortium. What is Unicode? Online as: `http://www.unicode.-org/standard/WhatIsUnicode.html`, May 2005.

[79] Xilinx, Inc. Virtex / E / EM. Online as: `http://www.xilinx.com/products/-silicon_solutions/fpgas/virtex/index.htm`, 2005.

[80] Xilinx, Inc. Xilinx: FPGA Product Tables. Online as: `http://www.xilinx.-com/products/silicon_solutions/fpgas/product_tables.htm`, 2005.

[81] Xilinx, Inc. Xilinx: ISE foundation. Online as: `http://www.xilinx.com/ise/-logic_design_prod/foundation.htm`, 2005.

[82] Xilinx, Inc. Xilinx: Logic Design. Online as: `http://www.xilinx.com/ise/-logic_design_prod/index.htm`, 2005.

[83] Xilinx, Inc. Xilinx: Virtex E/EM Product Tables. Online as: `http://www.-xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_e_em/-resources/virtex_e_em_table.htm`, 2005.

[84] Christopher Zuver. Architecture in reconfigurable hardware for quality of service in gigabit networks. Masters Project, Washington University in St. Louis, August 2004.

Vita

Charles M. Kastner

Date of Birth	July 21, 1981
Place of Birth	St. Louis, Missouri
Degrees	B.S. Summa Cum Laude, University of Missouri-Rolla, Computer Engineering, May 2003 M.S., Washington University in St. Louis, Computer Engineering, Expected May 2006
Honor Societies	Tau Beta Pi (Engineering Honor Society)
Publications	*HAIL: A Hardware-Accelerated Algorithm for Language Identification*, by Chip Kastner, Adam Covington, Andrew Levine, and John Lockwood; *In Proceedings of: The International Conference on Field Programmable Logic, Reconfigurable Computing, and Applications (FPL)*, Tampere, Finland, August 24-26, 2005. *Transformation Algorithms for Data Streams*, by John W. Lockwood, Stephen G. Eick, Doyle J. Weishar, Ron Loui, James Moscola, Chip Kastner, Andrew Levine, and Michael Attig; *In Proceedings of: IEEE Aerospace Conference*, Big Sky, Montana, March 5-12, 2005.

May 2006

Short Title: HAIL Kastner, M.S. 2006