High-Performance Regular Expression Matching
with Parabix and LLVM

by

Dale L. Denis
B.Tech. (Hons.), British Columbia Institute of Technology, 2005
B.A. Business Administration, Trinity Western University, 1988

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Dale L. Denis 2014
SIMON FRASER UNIVERSITY
Fall 2014

All rights reserved.
However, in accordance with the Copyright Act of Canada, this work may be reproduced without authorization under the conditions for “Fair Dealing.” Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
APPROVAL

Name: Dale L. Denis

Degree: Master of Science

Title of Thesis: High-Performance Regular Expression Matching with Parabix and LLVM

Examinining Committee:
Dr. Ted Kirkpatrick,
Professor,
Chair

Dr. Robert D Cameron,
Professor, Senior Supervisor

Dr. Nick Sumner,
Assistant Professor, Supervisor

Dr. Thomas C. Shermer,
Professor, Supervisor

Dr. Anoop Sarkar,
Associate Professor, SFU Examiner

Date Approved: October 6th, 2014
Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the non-exclusive, royalty-free right to include a digital copy of this thesis, project or extended essay[s] and associated supplemental files (“Work”) (title[s] below) in Summit, the Institutional Research Repository at SFU. SFU may also make copies of the Work for purposes of a scholarly or research nature; for users of the SFU Library; or in response to a request from another library, or educational institution, on SFU’s own behalf or for one of its users. Distribution may be in any form.

The author has further agreed that SFU may keep more than one copy of the Work for purposes of back-up and security; and that SFU may, without changing the content, translate, if technically possible, the Work to any medium or format for the purpose of preserving the Work and facilitating the exercise of SFU’s rights under this licence.

It is understood that copying, publication, or public performance of the Work for commercial purposes shall not be allowed without the author’s written permission.

While granting the above uses to SFU, the author retains copyright ownership and moral rights in the Work, and may deal with the copyright in the Work in any way consistent with the terms of this licence, including the right to change the Work for subsequent purposes, including editing and publishing the Work in whole or in part, and licensing the content to other parties as the author may desire.

The author represents and warrants that he/she has the right to grant the rights contained in this licence and that the Work does not, to the best of the author’s knowledge, infringe upon anyone’s copyright. The author has obtained written copyright permission, where required, for the use of any third-party copyrighted material contained in the Work. The author represents and warrants that the Work is his/her own original work and that he/she has not previously assigned or relinquished the rights conferred in this licence.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2013
Abstract

This thesis investigates the feasibility of constructing a high-performance, Unicode-capable, regular expression search tool by combining parallel bit stream technologies and algorithms together with the dynamic compilation capabilities of the LLVM compiler infrastructure. A prototype implementation of icGREP successfully demonstrates the feasibility of this undertaking, with asymptotic performance fully in line with that predicted by earlier prototyping work. The icGREP implementation extends the Parabix regular expression algorithms to include new techniques for efficient Unicode character matching. Performance evaluations in comparison with other Unicode-capable regular expression search tools show asymptotic performance advantages that are often over 10X, although the overhead of dynamic compilation techniques confines the benefits to relatively large input files.
Acknowledgments

First, I would like to express my gratitude to my senior supervisor Dr. Rob Cameron for providing me with the opportunity to complete this thesis. I have considered it an honor.

I would also like to thank Dr. Nick Sumner for his assistance and for the hours that he has spent proof reading my thesis. His guidance, and suggestions have been invaluable in the completion of this work.

I would like to thank Dr. Thomas Shermer for reviewing my work and for being on my supervisory committee.

Last, I would like to thank my parents Frank and Julia Denis for their encouragement and support. This thesis is dedicated to them.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Partial Copyright License</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vi</td>
</tr>
</tbody>
</table>

1. Introduction 1

2. Background 4
   - 2.1 Regular Expressions 4
   - 2.2 Unicode 6
   - 2.3 Parabix 8
     - 2.3.1 Parallel Bit Streams 8
     - 2.3.2 Basis Bit Streams 8
     - 2.3.3 Character Class Bit Streams 9
     - 2.3.4 Marker Bit Streams 10
     - 2.3.5 Advance 11
     - 2.3.6 ScanThru 11
     - 2.3.7 MatchStar 12
   - 2.4 LLVM 13
   - 2.5 Regular Expression Matching 15
     - 2.5.1 Algorithms 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.2</td>
<td>Parallelization</td>
<td>17</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Bitwise Data Parallel Regular Expression Matching</td>
<td>19</td>
</tr>
</tbody>
</table>

3 Implementation 20

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>System Architecture</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Parser</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>UTF-8 Encoding</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Nullable Optimizations</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Simplification Optimizations</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Reducer Optimization</td>
<td>29</td>
</tr>
<tr>
<td>3.7</td>
<td>Character Class Equations</td>
<td>30</td>
</tr>
<tr>
<td>3.8</td>
<td>Regular Expression Matching Equations</td>
<td>32</td>
</tr>
<tr>
<td>3.9</td>
<td>Unicode</td>
<td>32</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Processing of Multibyte Sequences</td>
<td>32</td>
</tr>
<tr>
<td>3.9.2</td>
<td>If Hierarchy Optimizations</td>
<td>34</td>
</tr>
<tr>
<td>3.10</td>
<td>LLVM</td>
<td>34</td>
</tr>
</tbody>
</table>

4 Unicode Feature Evaluation 38

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>icGREP</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Alternate grep Implementations</td>
<td>43</td>
</tr>
</tbody>
</table>

5 Performance Evaluations 44

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Regular Expression Optimization Transformations</td>
<td>46</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Nullable</td>
<td>47</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Simplification</td>
<td>48</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Reducer</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>LLVM Optimizations</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>icGREP vs. the Prototype</td>
<td>51</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Asymptotic Performance and Dynamic Compilation</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>Unicode Performance</td>
<td>55</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Comparisons with other Unicode Capable GREP Implementations</td>
<td>55</td>
</tr>
<tr>
<td>5.4.2</td>
<td>The If Hierarchy Optimization of Named Classes</td>
<td>56</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Searching for string patterns in text is an important task in the field of computing, and with the ever increasing amount of data that is being generated, transferred, and stored the need for applications that are able to search for and identify patterns in text in a timely manner is now more important than ever. Examples of applications that search for patterns in text include parsers, web search engines, anti-virus scanners, natural language processors, and applications in the field of bioinformatics, such as those that identify DNA markers in the human genome [27] [21].

A significant amount of research has been put into finding ways to search for patterns in text. In the 1950s the mathematician Stephen Kleene developed theories on automata, and in the 1960s Ken Thompson converted regular expressions into nondeterministic finite automata (NFA). Regular expressions are special sequences of characters that are used to define search patterns. Many of the tools in use today that use regular expressions are still based upon the work of Kleene and Thompson.

With the arrival of multi-core processors (CPUs), graphical processing units (GPUs), and field-programmable gate arrays (FPGAs) researchers have tried improving the performance of regular expression matching through parallelization. Many of the efforts in this area have realized performance gains simply by adapting the traditional algorithms of Kleene and Thompson to the new parallel architectures. One completely new approach to parallelization, and one that has shown considerable promise, has been to use the data-level parallelism of wide SIMD units to implement parallel bit streams through the use of bitwise data parallelization. The algorithms for using bitwise data parallelism for regular expression matching were originally developed for Unicode transcoding by Robert D. Cameron [19].
The algorithms were also applied to XML parsing by Cameron, Herdy, and Lin [20] and have since been built into a framework called Parabix. When compared to conventional XML parsers that do not use parallelism, the XML parser using bitwise data parallelism demonstrated a 2x-7x speedup and a 4x improvement in energy efficiency [11].

Recently, a prototype compiler for regular expressions was constructed by Robert D. Cameron, et al. that used components from the new XML parser to apply bitwise data parallelization to the search for string patterns in text [21]. The compiler generated an implementation of grep. Grep stands for “Global Regular Expression Print”. A grep program searches a file for lines with matches of the regular expression and either prints the lines or prints a count of the number of lines that were found. The first implementation of grep was built by Ken Thompson and became available in 1973 as a Unix utility for finding content [15, 24]. In evaluations, the prototype compiler demonstrated a 5x or better performance advantage over its closest competitors that used finite automata [21]. These results provided a strong indication of the potential that existed for a new high-performance regular expression engine constructed using the bitwise data parallelization technology of parallel bit streams.

The prototype compiler consisted of a tool chain that included a Java application, three scripts, and three other compilers. Compiling a regular expression required the three scripts to be run in a specific order resulting in a statically compiled executable. The executable could then be used to process an input file with the regular expression. It remained to be seen, however, if the performance advantage of the prototype could be maintained after unifying the tool chain into an application that could both compile a regular expression and then process an input file in a single high-level step, similar to other grep implementations.

The LLVM compiler infrastructure is a relatively new compiler infrastructure that has a reputation for being easy to learn and to modify and that also supports just-in-time (JIT) optimization and compilation. A compiler with JIT capabilities or an interpreter is what is required in order to achieve the goal of building such a unified solution.

The prototype did not support Unicode, it only supported ASCII. ASCII includes only the letters, digits, and punctuations that are a part of the English language, while Unicode is a standard that was designed expressly for the purpose of encoding all of the characters in all of the world’s writing systems. The Unicode standard is the standard for character encoding that has been adopted by many of the industry leaders such as Apple, IBM, HP, Oracle, and Microsoft.
CHAPTER 1. INTRODUCTION

By building a unified, Unicode capable application for grep, this thesis will show that a high-performance, Unicode capable regular expression matching application built with LLVM and Parabix technologies can be performant.
Chapter 2

Background

2.1 Regular Expressions

Regular expressions are a domain specific programming language that has a general pattern notation that allows a user to search texts for occurrences of string patterns. Regular expressions can be effectively applied to a large number of use cases. Regular expressions can be used for input validation, such as to ensure that phone numbers and postal codes have been entered in the correct format, and they can be used to search for and extract information such as for retrieving email addresses from large documents. Regular expressions have also been used for network intrusion detection, tokenization, virus scanning, in spam filters, and to search for patterns in the field of bioinformatics.

In regular expressions, there are two types of characters: literals, and metacharacters. A literal is a character that stands for itself, for example the character A is a literal that will match any character that is a capital “A”. The regular expression dog is a concatenation of three literals that will match the three character string “dog”. Metacharacters are characters that have a special meaning; they tell the regular expression how to match the other characters in the regular expression. ^$[]{}\.?*+ are all common metacharacters. The metacharacter ^ matches the beginning of a line while the metacharacter $ matches the end of a line. For a metacharacter to match the literal value of itself, the character has to be preceded or escaped with a back slash. For a dollar sign in a regular expression to match a dollar sign in a text, the dollar sign needs to be escaped and would appear in the regular expression as \$.

There are a number of difference standards for regular expressions. In the mid 1980’s the
POSIX standard was introduced. POSIX stands for “Portable Operating System Interface” and it was designed to help ensure portability across operating systems. A specification for regular expressions was included as a part of the POSIX standard. Also in the mid 1980’s, the programming language Perl was introduced along with its own unique implementation of regular expressions. The Perl implementation has become popular with programming languages such as Python, Ruby, and the Microsoft .Net family of languages. All of these have regular expression implementations that are considered to be “Perl compatible” [25].

icGREP follows the POSIX notation for regular expressions.

According to the POSIX standard, square brackets in a regular expression indicate POSIX bracketed expressions. A POSIX bracketed expression is used to indicate a character class, which is a set of characters from which a single character can be matched. [0#%] is an expression that specifies a character class where a character may match any one of the three bracketed symbols. A contiguous range of characters specified with a hyphen indicates a character class that consists of a range of characters that can be matched. The regular expression [a-z] expresses a character class that will match a single lowercase character from a to z, and the regular expression [0-9] will match a single number from 0 to 9. If the metacharacter ^ is used within a bracketed expression, such as in [^abc] it means that the character class is negated. A negated character class matches any character except for a character that is within the bracketed expression. Round brackets are used to specify precedence. There are also repetition characters such as the metacharacter * which specifies a match of zero or more, and + which specifies a match of one or more. [a-z]* specifies a match of zero or more characters that are in the lowercase range of a to z. Optional characters can be specified with a question mark, a? specifies that an “a” may or may not exist in the text. Braces can be used to specify boundaries. [a]{3} means that a string of three lower case a’s must be matched. [a]{3,5} specifies that there must be at least three consecutive a’s but that there can be no more than five. [a]{3,} means that there must be at least three but beyond that there can be an unlimited number of consecutive a’s. The alternation operator | allows for options, such as (am) | (pm) that could be used when matching the time of the day.
2.2 Unicode

Unicode was invented in order to assign a unique number to each character of every writing system in the world. Before Unicode, hundreds of encodings existed and the encodings often overlapped. The same numbers represented different characters in different encodings, and at times the same characters would be represented in different encodings by different numbers. A code point is unique address or number within the Unicode code space. There are well over a million addresses within the Unicode code space, many of these addresses are still unassigned.

Three different encoding formats are specified by the Unicode standard: the 32-bit UTF-32 encoding, the 16-bit UTF-16 encoding, and the 8-bit UTF-8 encoding.

UTF-32 is the least complex of the three encoding formats to use. A code unit is defined as the minimal bit combination that can be used to encode a single unit of text. With UTF-32 each code point is encoded as a 32-bit code unit. The problem with UTF-32 is that it is very inefficient. Twenty one bits are the maximum number of bits that can be used for a Unicode code point, so with UTF-32 eleven bits will always go unused. For text that is just in English the problem is even worse. Every character in the English language can be encoded with a maximum of seven bits, so for the English language twenty five bits will always be unused. Text encoded with UTF-32 is easy to use but it is also very inefficient to store and transmit.

UTF-16 is a variable width encoding format that uses up to two 16-bit code units. A single two byte UTF-16 code unit is sufficient for the majority of the characters that are normally used. Characters that can be represented within this range are said to reside within the Basic Multi-lingual Plane (BMP). For characters that are in the supplemental plane outside of the BMP, two 16-bit, two byte code units are used that together are referred to as a surrogate pair.

The most widely used encoding format is UTF-8 [30]. UTF-8 is the byte oriented format of Unicode that was specifically designed to be backwards compatible with ASCII [26]. UTF-8 is a variable width encoding that uses from one to four 8-bit code units per code point. The high bits of the first byte of a sequence of UTF-8 code units are used to indicate the total number of bytes that are required. The encoding format of UTF-8 is shown in figure Figure 2.1. The “x”s that are shown in the columns for bytes 1 through 4 would be replaced by the bits from the code point being encoded. For example, the code point for the lower
case letter “a” is the hexadecimal value 0x61, which is less than the maximum value of 0x7F for a UTF-8 encoded single byte. Therefore, the code point would have the UTF-8 encoding of a single byte with the value 0x61. The code point for the euro symbol € on the other hand is the hexadecimal value 0x20AC which is greater than the minimum value, and less than the maximum value for the three byte UTF-8 encoding. With the correct prefixes in place, the 0x20AC code point would be encoded as the three byte sequence of 0xE2, 0x82, and 0xAC.

<table>
<thead>
<tr>
<th>Bits of code point</th>
<th>First code point</th>
<th>Last code point</th>
<th>Bytes in sequence</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0x0000</td>
<td>0x007F</td>
<td>1</td>
<td>10xxxxxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0x0080</td>
<td>0x07FF</td>
<td>2</td>
<td>110xxxxx</td>
<td>10xxxxxx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0x0800</td>
<td>0xFFFF</td>
<td>3</td>
<td>1110xxxxx</td>
<td>10xxxxxx</td>
<td>10xxxxxx</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0x10000</td>
<td>0x1FFFFF</td>
<td>4</td>
<td>11110xxx</td>
<td>10xxxxxx</td>
<td>10xxxxxx</td>
<td>10xxxxxx</td>
</tr>
</tbody>
</table>

Figure 2.1: UTF-8 Character Encoding [10]

UTF-8 is computationally expensive, due to the validation and the classification that is required for each byte, and for the algorithmic mapping that is required between each code point and the unique bytes sequences. The benefits of UTF-8 though, are that it is backwards compatible with ASCII, it is compatible with widely deployed 8-bit ASCII based networking software, and in general it requires less room to store, and less bandwidth to transmit.

The Unicode standard also contains guidelines for the support of Unicode in regular expressions. The guidelines specify three different levels of Unicode support that can be offered by a regular expression engine. The first level is the basic level of support, and it specifies the minimal set of features that are required to support Unicode. This level is targeted at the programmer, or the technical user, and it provides all of the basic tools that a user needs to work with Unicode. The second level of support is considered as extended Unicode support. The second level of Unicode support specifies features that are intended to allow a user to use the regular expression engine without having detailed knowledge of the underlying Unicode structure and encoding. The third level of support is tailored support, which specifies requirements for when the regular expression engine has been tailored to support a specific user’s locale, so that it meets a the user’s expectations of what constitutes a character in the user’s language [9].
2.3 Parabix

Modern multicore processors provide applications with the opportunity for improved performance through parallelization, but unfortunately applications that process text have a difficult time taking advantage of this opportunity as text processing applications are notoriously difficult to parallelize [14]. Traditional text processing techniques use finite automata, which use the byte-at-a-time processing model; a model that as well as being difficult to parallelize, is also prone to significant inefficiencies due to branch and cache miss penalties [11]. Parabix provides an entirely new approach to text processing. Through the use of data parallelization, Parabix has been shown to provide a significant performance advantage over other traditional text processing methods. Parabix achieves its data parallelization by leveraging features on the new SIMD (single-instruction multiple-data) registers that are available on modern processors.

Parabix is a programming framework that is made up of three parts: an architectural specification for text parsing through the use of parallel bit streams, a tool chain of components to perform the transformations that are required to support the architecture, and a runtime environment that provides an abstraction layer for SIMD programming that provides portability over multiple target processor architectures.

2.3.1 Parallel Bit Streams

Parabix stands for “Parallel Bit Streams”, and it provides a new transform representation of text. Parabix is built upon the novel concept of bitwise data parallelism. The underlying abstraction model for Parabix is the use of unbounded, very large integers. The fundamental operations that are used by Parabix are bitwise logic, stream shifting, and long-stream addition.

2.3.2 Basis Bit Streams

Instead of processing text sequentially a byte at a time, Parabix first transposes the input byte stream into eight parallel bit streams that are known as basis bit streams. Each basis bit stream $b_k$ represents the k-th bit of each byte. This provides a one-to-one correspondence between the i-th bit of each of the basis bit streams and the i-th character of the input text. With the use of 128-bit SIMD registers this allows a block of 128 bytes to be processed in parallel, one block-at-a-time. The amortized cost of the transposition process on 64-bit
commodity hardware using 128-bit SSE2 SIMD registers is approximately 1 CPU cycle per byte [11]. The concept of the transposition from a byte stream to parallel bit streams is analogous to forward and inverse Fourier transforms to transform between the time and frequency domains in signal processing. With Parabix, the transformation is between byte-space and bit-space. Figure 2.2 shows how the input string of “.com” would be represented after being transposed into 8 basis bit streams:

<table>
<thead>
<tr>
<th>STRING</th>
<th>.</th>
<th>c</th>
<th>o</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>00101110 01100011 01101111 01101101</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b₀</th>
<th>b₁</th>
<th>b₂</th>
<th>b₃</th>
<th>b₄</th>
<th>b₅</th>
<th>b₆</th>
<th>b₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ASCII</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ASCII</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ASCII</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.2: Basis Bit Streams for the String “.com”

### 2.3.3 Character Class Bit Streams

With traditional text parsers, the value of each byte of the search text is compared sequentially with a set of search characters. With Parabix up to 128 bytes of input text are compared with the set of search characters in parallel. To perform this comparison Parabix first performs in parallel a set of boolean-logic operations that merge the eight basis bit streams into character class bit streams that mark the locations of each search character. For example, the character “m” is the decimal value 109 which is the binary value 01101101. To identify the character m in a stream of ASCII characters the following boolean-logic operations would have to be applied: \((b₁ \land \neg b₀) \land (b₂ \land \neg b₃) \land (b₄ \land b₅) \land (b₇ \land \neg b₆) = 1\). Ranges of characters such as [0–9] can also be identified by applying boolean-logic operations in a similar manner. Figure 2.3 shows an example of the character class bit streams that would be generated for the characters a and z and for the range [0–9] from the eight basis bit streams transposed from a stream of input text.
2.3.4 Marker Bit Streams

To match a pattern of characters in a stream of input text, such as with regular expression matching, marker bit streams are used to mark the current position of the match during the matching process. Figure 2.4 illustrates how the marker bit streams are used to match the regular expression \(a[0-9]*z\).

The marker bit stream \(M_1\) indicates all of the locations in the stream of input text just after a match of the first character in the regular expression. In this case the first character is the character “a”. The marker bit stream \(M_2\) indicates the locations in the input text of the second character in the regular expression that occur just after the matches of the first character in the regular expression. For this regular expression, the match is of zero or more of the decimal values from 0 to 9. The marker bit stream \(M_3\) is used to mark the...
locations just after the third character in the regular expression that occurred after matches of the second character in the regular expression. As $M_3$ contains the matches for the last character in the regular expression, the bit stream $M_3$ is the bit stream that is returned to indicate the locations in the input text where the string pattern specified by the regular expression has been matched.

### 2.3.5 Advance

The Advance operation is the operation in Parabix that sets the marker bit that is one position past the position of the latest match. This bit is called the cursor bit. To perform the advance, the advance operation accepts as input a marker bit stream, and it advances each cursor forward one position. If a cursor that is to be advanced is the last bit of a block of input text, the bit is placed into a carry queue, and the bit is then carried into the next block of input text.

### 2.3.6 ScanThru

ScanThru is an operation in Parabix that sets the cursor positions immediately after a run of marker positions in the input bit stream [21]. This allows a run of marker positions to be matched all at once, instead of having to advance the cursors through the marker positions one bit position at a time. ScanThru accepts two input parameters, an input bit stream of marked cursor positions and a bit stream that contains the positions of the markers that are to be matched. For the input cursor positions in $M_1$, Figure 2.5 shows the steps that are used by ScanThru to determine the cursor positions immediately following the consecutive occurrences of characters in the character class marker bit stream $CC$.

```
<table>
<thead>
<tr>
<th></th>
<th>input data</th>
<th>12abcd--efghij----345xyz---</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>.1........1.</td>
<td></td>
</tr>
<tr>
<td>$CC$ = [a-zA-Z]</td>
<td>.1111....111111.111...</td>
<td></td>
</tr>
<tr>
<td>$M_2$ = $M_1 + CC$</td>
<td>....1........1.111...</td>
<td></td>
</tr>
<tr>
<td>$M_3$ = $M_2 \land \lnot CC$</td>
<td>....1........1.</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 2.5: $M_3 = \text{ScanThru}(M_1, CC)$
2.3.7 MatchStar

MatchStar is the parallel bit stream implementation of a character class followed by a Kleene star. A Kleene star in a regular expression, indicates a concatenation of zero or more of the strings represented by the expression immediately preceding the star [37]. Matchstar takes a character class bit stream and a marker bit stream as input, and it returns a bit stream that indicates all of the positions in the input text where zero or more consecutive occurrences of the characters of the character class can be found.

\[
\begin{align*}
\text{input data} & \quad \text{a453z--b3z--az--a12949z--ca22z7--} \\
M_1 & \quad .1\.\.\.\.1\.1\.\.\.\.1\.\ldots \\
D & = [0-9] \quad .111\.\.\.\.11111\.\.\.\.111\.\ldots \\
T_0 & = M_1 \land D \quad .1\.\.\.\.1\.\.\.\.1\.\ldots \\
T_1 & = T_0 + D \quad \ldots1\.\.\.\.1\.\.\.\.1\.\ldots\ldots11\ldots \\
T_2 & = T_1 \oplus D \quad .1111\.\.\.\.111111\ldots111\ldots \\
M_2 & = T_2 | M_1 \quad .1111\.\.\.\.1\.1111\ldots111\ldots \\
\end{align*}
\]

Figure 2.6: \( M_2 = \text{MatchStar}(M_1, D) \) [21]

Figure 2.6 illustrates the steps that are required to implement MatchStar. The input marker bit stream is \( M_1 \), and the input character class bit stream is represented as \( D \). The input marker bit stream contains markers that indicate the locations in the input text where the MatchStar process is to begin. The first step in the MatchStar process is to perform a logical \( \land \) of \( M_1 \) and \( D \). This results in the bit stream \( T_0 \), which marks either the locations of the individual characters, or the beginning of the sequences of the characters in the bit stream \( D \) that match \( M_1 \). The second operation, which results in \( T_1 \), is an \( \lor \) of \( T_0 \) and \( D \). Adding \( T_0 \) and \( D \) marks the locations immediately following one or more consecutive occurrences of the characters contained in \( D \). \( T_1 \) also contains markers indicating the locations of characters in \( D \) that are in the input text but that are not matches for the MatchStar operation. These matches are eliminated by the following two operations. The third operation is an \( \oplus \) of \( T_1 \) and \( D \) that results in the bit stream \( T_2 \). \( T_2 \) contains markers for all of the characters contained in \( D \) plus the character immediately following each match. As a Kleene star indicates zero or more of the characters contained in \( D \) the only markers missing from \( T_2 \) after this operation are the markers from \( M_1 \) that did not
have any matches in $D$. Therefore, the final operation of the MatchStar process is an \textit{OR} of $T_2$ and $M_1$ that results in the bit stream $M_2$. $M_2$ is the bit stream that is returned as the output from MatchStar [21].

2.4 LLVM

The LLVM compiler infrastructure is a set of modular compiler components and tools that were developed as a research project at the University of Illinois in 2000. The motivation for the project was the recognition of the fact that while the compilers that were available at that time were able to provide the application performance that was needed, the time that it took to compile applications was becoming far too long. The primary goal for the project was to reduce application compile time by introducing a new compiler that would contain a unique new multi-stage optimization system. The key to the system was the LLVM virtual instruction set that could be used to support lifelong program analysis and transformation; it supported aggressive optimizations at both link and post-link time, and it also supported optimizations at run-time and at idle-time both on the target machine.

With traditional compilers an application is first compiled into an object file containing machine code and then the linker links the object files that the application needs in order to create an executable. With LLVM, instead of compiling directly to machine code the compiler front-end first emits a file containing a representation of the LLVM instruction set. The optimizing linker then links this file with the other files that are needed and optimizes them together as a unit. This process allows for aggressive inter-procedural optimizations to be applied at the point in the compilation process where the optimizations are able to be the most effective. The output from the optimizing linker is the final executable that is written to disk.

LLVM provides both static compilation and dynamic compilation, using the just-in-time (JIT) compiler or with the new machine-code-just-in-time (MCJIT) compiler. MCJIT is LLVM’s most recent version of their compiler for dynamic compilation [28].

The LLVM instruction set is designed to be represented in three different forms: as an in memory Intermediate Representation (IR), as an on-disk bitcode representation, or in a human readable assembly type form. Tools that are available with the LLVM framework are able to transform the IR from one form of representation into another. The instruction set is a simple static single assignment (SSA) form based three address code representation that
is similar in nature to machine code. The instructions are polymorphic, and the instruction set contains only a small number of simple operations which helps to reduce the number of distinct opcodes. As a virtual instruction set, the instruction set does not define any specific runtime or operating system functions [22].

An additional goal for the LLVM project was that it was to serve as a host for leading research and development. In order to achieve this goal the compiler was designed to be as language and target-independent as possible; the compiler had to be source language agnostic so that it could support arbitrary languages. While the IR is strongly typed, the type system only includes source-language-independent low level primitive types, and it only includes four derived types. LLVM does not represent any high-level language features directly, and it does not capture any of the machine dependencies used by the back-end code generators. For a front end, LLVM appears as a strict RISC architecture, meaning that any high-level features of the language must be mapped down to the low level primitive form. While this may have the appearance of being somewhat restrictive, this is a feature that both adds to the portability of the generated code, and to the over-all versatility of the compiler.

To make it easier to write new compiler front-ends, and to support arbitrary languages the LLVM infrastructure includes a number of tools and an API. There are tools that will compile C or C++ code into bitcode, and there are tools that will turn this bitcode into human readable form. The LLVM IR Builder API is helpful in that it supplies a large number of methods that can be called in order to generate the function calls that a front-end needs in order to emit the IR.

Optimization passes are also included with the LLVM infrastructure, that can be used when generating IR. For example, even though the LLVM back-end will only consume IR that is in SSA form front-end implementations do not need to generate IR in SSA form directly. The front-ends simply need to use the alloca instruction to allocate the memory for variables on the stack and then the mem2reg optimization pass will handle the promotion of the memory allocations into SSA registers.
CHAPTER 2. BACKGROUND

2.5 Regular Expression Matching

2.5.1 Algorithms

The first version of grep that was introduced by Ken Thompson was based upon NFA. With NFA a regular expression of the length \( m \) must first be converted into \( m \) NFA nodes in \( O(m) \) time. When searching text of the length \( n \) it takes the maximum time of \( O(mn) \) with \( m \) NFA nodes [21].

In 1976, Alfred Aho introduced a new version of grep called egrep that was based upon deterministic finite automata (DFA) [12]. With DFA, text of the length \( n \) can be searched in the maximum time \( O(n) \) [21]. With DFA there is never more than one possible transition from each state of the automaton to the next, however the setup time needed for DFA is much longer than it is with NFA. Regular expressions are normally first converted into an NFA and then the DFA are generated from the NFA [37]. To help reduce the impact of the additional setup time that is required by DFA, egrep was later enhanced with lazy evaluation in order to defer the setup of the DFA until just before it is needed.

There is also a risk that is known to be inherent in the process of converting NFA to DFA, and that is the risk of state explosion. State explosion is when the time and the space that is required for the conversion grows exponentially with the length of the regular expression.

Much of research that has gone into regular expression engines since the original work of Thompson and Aho has continued to use NFA and DFA as the basis of their designs. Perl, PCRE, Ruby, Python, and the .Net languages all have regular expression engines that have designs that are based upon NFA, while RE2 and most versions of awk and egrep are all based upon DFA. A number of the regular expression engines such as GNU awk and GNU grep/egrep are even based upon hybrid NFA/DFA engines. Hybrid NFA/DFA engines provide a way of dealing with state explosion. If the engine detects an impending DFA blowup the NFA-to-DFA conversion is interrupted and the engine uses a hybrid automata [16].

RE2 is one of the regular expressions engines for grep that provides at least partial Unicode support. RE2 supports UTF-8 by compiling UTF-8 character classes down into DFA that read input one byte at a time. The decoding of the UTF-8 byte sequences is built right into the automaton [7].

In the early 1990's Baeza-Yate and Gonnet introduced the Shift-Or algorithm that uses bit-parallelism to simulate an NFA. Bit-parallelism uses bitwise operations to take advantage
of the parallelism of the bit manipulations within a computer’s word in order to perform multiple operations at the same time. Given the width of a computer’s word as \( w \) the Shift-Or algorithm has a worst-case runtime of \( O\left(\frac{mn}{w}\right) \).

With the Boyer-Moore algorithm, a pattern is matched in reverse starting from the back of the pattern instead of the front. The algorithm allows for characters of the input text to be skipped. For a pattern that is 5 characters long, on average \( \frac{i}{4} \) characters of the input text will have to be inspected before finding a match at character \( i \) [17].

Like the Boyer-Moore algorithm, the Backward DAWG Matching (BDM) algorithm matches in reverse order. The difference with the BDM algorithm though is that it matches the input text against suffix automata. The Backward Nondeterministic Dawg Matching (BNDM) pattern matching algorithm takes this one step further by adding the bitwise parallelism of the Shift-Or algorithm to the BDM algorithm. The pattern matching tool NR-grep is based upon the BNDM algorithm [21].

The Knuth-Morris-Pratt (KMP) algorithm is a linear time string-matching algorithm that was designed to find simple string patterns in text [29]. The motivation for the KMP algorithm was to design an algorithm that avoided the need to back-up through the input text after a failure in the match. With a brute-force pattern matching algorithm, if a 5 character long pattern started matching on the character \( i \) of an input text and the match failed on the 3rd character of the pattern, the pattern would be tried again on character \( i + 1 \). The algorithm would have backed up from \( i + 2 \) where the match had failed. With KMP a setup function is used to build an array that encapsulates information about how the input pattern shifts against itself. For a pattern of the length \( m \), the setup function can be completed in \( \Theta(m) \). Together with a linear search time for a search text of the length \( m \), the run time of the KMP algorithm is \( \Theta(m + n) \) [23].

The Aho-Corasick (AC) algorithm is a linear time algorithm that was introduced by Alfred Aho and Margaret Corasick in 1975. The algorithm was built upon ideas from the KMP algorithm that were combined with finite state machines. The setup function, instead of building the array with information about the search pattern replaces the array with a finite state machine. The AC algorithm is able to process input text in a single pass significantly faster that it would be able to without the finite state machine [13].
CHAPTER 2. BACKGROUND

2.5.2 Parallelization

Up until approximately 2003 the trend had been for ever increasing clock speeds with each new generation of CPU. At around 2003 though, the CPU manufacturers hit a wall where increasing the speeds of the CPUs was no longer practical due to problems with heat dissipation, excess power consumption, and power leakage. This did not mean that Moore’s law of exponential growth had ended, instead for increased performance the CPU manufacturers turned en masse to multicore architectures [40]. What this meant for application designers was that applications now had to be developed that used parallelization in order take advantage of the opportunities for increased performance that were made available by the new architectures.

With the new processor architectures, and with regular expressions being the method of choice to search for patterns for such a large number of classes of applications, researchers have been trying to find ways to parallelize regular expressions. The challenge with regular expressions though, is that pattern matching in text is famous for being difficult to accelerate via parallelization. In 2006 a study was released by the University of California, Berkeley into the state of parallel computing research. The study identified finite state machines as being “embarrassingly” sequential, and most regular expression engines are based on finite state machines [14]. The study went on to suggest that perhaps the best solution would be to find a new algorithmic approach.

This does not mean, however, that researchers have not been able to reach a certain level of success in the parallelization of regular expressions using traditional methods. For example, Yu, et al. proposed a DFA based parallelized solution for Network Intrusion Detection Systems (NIDS) that could be implemented on multi-core CPUs or Field Programmable Gate Arrays (FPGA). With NIDS regular expressions have been replacing explicit string patterns for deep packet inspection to search for patterns that identify malware. Regular expressions are preferred due to their expressive power and their flexibility in describing patterns. Unfortunately, as both the size of the malware dictionaries increase, and as the network speeds increase regular expressions have a hard time keeping up. For NIDS regular expressions can actually cause a significant reduction in network throughput. To accelerate regular expression matching for NIDS, Yu. et al. proposed a parallelized solution that included a grouping algorithm that grouped the DFAs into sets, or groups based upon the
DFAs interactions with each other. The sets of DFAs were then assigned to individual processing units so that the DFAs could be run in parallel. This parallelized solution was able to achieve a factor of 12 to 42 performance improvement over another commonly used DFA based NIDS implementation [41].

Sidhu and Prasanna presented a parallelization method for finding matches in text with regular expressions that could be used for both FPGAs and Self-Reconfigurable Gate Arrays (SRGA). The algorithm that they introduced provided a fast NFA construction. When compared against GNU grep, their solution proved to be several orders of magnitude faster when implemented on a FPGA, and several order of magnitude faster still when implemented on an SRGA. With the SRGA the NFA was constructed on the SRGA itself using self-configuration [38].

For General-Purpose Graphics Processing Units (GPGPUs) Naghamouchi, et al. have proposed a small-ruleset DFA based solution for tokenization. Large-rulesets are for applications such as virus scanners where regular expressions are used to represent tens of thousands of virus signatures. A small-ruleset tokenizer is an optimized solution that would be used for business analytics suites, search engines, XML and natural language processing processors, or for compilers. With Naghamouchi’s solution each thread is assigned its own DFA along with a list of documents. Each thread then processes its documents sequentially, appending output to a shared output table. Performance evaluations of the solution indicated a 49x speedup over a single threaded instance run on a typical commodity processor [33].

Pasetto, Petrini, and Agarwal presented a solution that provided a speedup on multicores processors that was of one to two orders of magnitude over the conventional single threaded implementations of Boost and RegexLib. Their solution, called DotStar compiled large sets of regular expressions into a compact intermediate language, which in turn was compiled into AC DFAs [35]. To run in parallel the DFAs were distributed across the number of processing cores that were available on the target machine [36].

Mytkowicz, Musuvathi, and Schulte recently presented a parallel solution for DFAs that in evaluations was up to 3 times faster than sequential implementations on a single processor, and 21 times faster on eight cores. The solution splits the input data into chunks, and then performs the computations for each chunk in parallel on SIMD hardware. Every DFA state is processed for each chunk, and each state is enumerated. The computations that are chosen from the next chunk to continue are the computations that start with the correct final states from the previous chunk. The disadvantage of the technique is the overhead of
the additional computations, but the authors were able to implement optimizations that made the computations more efficient [32]. As the solution focused on DFAs, the solution still ran the risk of state-explosion during the NFA to DFA conversion.

All of the parallelization efforts above sought to adapt traditional byte-at-a-time string matching algorithms using finite state machines to the new parallel architectures. Parabix presents an algorithmic approach to parallelization that is entirely new.

### 2.5.3 Bitwise Data Parallel Regular Expression Matching

For a regular expression of the length $m$ it was shown that the algorithm for bitwise data parallel regular expression matching requires $O(m)$ time, and the other steps of the Parabix tool chain required a time of $O(m \log m)$ [21]. Let $\Sigma$ be our input alphabet and $\sigma = |\Sigma|$. We assume that $\Sigma$ is a standard alphabet such as ASCII($\sigma = 128$), UTF-8($\sigma = 256$), UTF-16($\sigma = 65,536$), and UTF-32($\sigma = 1114112$). This allows us to limit the number of bits in the character encoding with $\lceil \log \sigma \rceil$. Parabix compiles a regular expression of the length $m$ into bitstream code that is $O(m \log \sigma)$ statements long. The bitstream code is then put inside a loop that is executed once for every $w$ characters of input text, where $w$ is equal to the width of the processor’s SIMD registers. Also executed for these characters is the transposition of the input bytes into parallel bit streams which takes $O(\log \sigma \log \log \sigma)$. If $n$ is the total number of characters in the input document, then $O(n/w)$ iterations of the processing loop will be required to process the document. Multiplying the required number of iterations by the work that must be done within each iteration of the loop provides a total of $O\left(\frac{n(m+\log \log \sigma)}{w} \log \sigma\right)$ work [21].
Chapter 3

Implementation

The prototype regular expression compiler was a research compiler that was made up of a tool chain that required user interaction at each step of the compilation process. The first challenge of this thesis therefore, was to use LLVM to unify the tool chain of the prototype. The goal was to unify the tool chain into a single application in such a way that the compiler would transform a regular expression into machine code that would be executed immediately without first writing the code to disk. To process an input file, a regular expression would be compiled and the generated code would be executed in a single step, similar to other grep implementations. The second challenge was to add Unicode support. The prototype only supported ASCII, where only the characters of the English language are encoded, and where each character is encoded as a single byte. The goal for the new compiler was to support Unicode UTF-8 encoding, where all of the characters for all of the world’s writing systems can be encoded in either single byte or in multibyte sequences.

3.1 System Architecture

The tool chain of the prototype included a Java application, a number of scripts, and three other compilers. The architecture of the prototype is shown in figure 3.1. Two of the compilers, the character class compiler and the Pablo compiler were from the Parabix tool chain while the third was the GCC C++ compiler from the GNU compiler collection. The purpose of the Java application was to parse a regular expression, extract the character classes, and to generate the Pablo parallel bit stream equations that were required to support the various operators and constructs included in the regular expression. The character
class compiler was written in Python, and it was used to transform the extracted character classes into Pablo character class equations. Pablo is a language that was developed to allow parallel bit stream programs to be written using primitives that manipulate arbitrary-length bitstreams [3]. The character class equations and the parallel bit stream equations required for the regular expression matching were together fed as input into the Pablo compiler, which transformed the equations into block-at-a-time C++ code, adding carry queues as necessary for each of the additions and shifts that were included in the parallel bit stream equations. The Pablo compiler also added the necessary function calls to the Parabix IDISA run-time libraries. The IDISA run-time libraries provide an abstraction for portable SIMD programming [4]. After generating the C++ code, the Pablo compiler inserted the generated code into a C++ template file that contained the static code that was necessary for the block-at-a-time processing of the input file for grep. Finally, the GCC compiler was used to compile and link the C++ file with the Parabix run-time libraries in order to generate a native executable. Compiling a regular expression required running three scripts in a specific order, with the GCC compiler being used to generate the static binaries for the target machine.
Figure 3.1: Architecture of the Prototype Regular Expression Compiler
Figure 3.2 shows the architecture of icGREP, the new unified regular expression compiler. The architecture of the application has been partitioned into two distinct components: the compiler for regular expressions, and the main processing loop for the block-at-a-time processing of the target input document.

The compiler for the regular expressions transforms a regular expression into LLVM IR that is just-in-time compiled (JITted) into a function that is called by the block-at-a-time processing loop. The component has been implemented as a class that acts as a driver for the individual transformations that are used to compile the regular expression. The outputs from the transformations are semantically equivalent to the outputs from the tools in the tool chain of the prototype. As the outputs of the tools of the prototype are known to be correct, implementing the transformations to produce the same outputs helped when validating the correctness of the transformations during the construction and testing of the new compiler. Instead of being statically compiled though, with icGREP the final outputs from the transformations are JITted into machine code, which is then called by the block-at-a-time processing loop that was statically compiled.

The compiler component contains a number of additional transformations that were not included in the prototype compiler. In the prototype, the parser extracted the character classes from the regular expression, and they were passed directly to the character class compiler for the generation of the character class equations. In icGREP, an abstract syntax tree (AST) is output from the parser, but instead of going to the transform function that implements the algorithms of the character class compiler, the AST is first fed as input into the UTF-8 encoder. The AST emitted from the parser is a data structure consisting of objects that represent each of the constructs of the regular expression, including objects that represent the character classes. The character class objects in this AST contain the code points for the character classes from the regular expression. The UTF-8 encoder processes the AST from the parser, and it outputs a new AST with character classes that have been transformed from character classes with code points into character classes with UTF-8 encoding. This new AST with the UTF-8 encoded character classes is then passed through three new additional optimization transformations to optimize the regular expression before it is finally passed to the transformations that generate the character class equations and the regular expression matching parallel bit stream equations.

The component of icGREP that performs the block-at-a-time processing of the input document contains a processing loop that reads text from an input file, uses functions in the
Parabix IDISA libraries to transpose the text into parallel bit streams, and then it passes the bit streams to the JITted function for processing. After processing, the JITted function returns to the block-at-a-time processing loop a bit stream that marks the positions of the search items that were found in the input text. The lines of the input text that contained matches are then printed, or they are counted and a count of the total number of lines with matches is printed when the processing of the input file is complete.

The block-at-a-time processing component also contains the statically compiled data types of the Unicode general categories. Every character in the Unicode database has been assigned a general category classification based upon the characters type. Each of the statically compiled Unicode general category data types in icGREP contain a callable function that contains the character class equations and the regular expression matching parallel bit stream equations that are necessary to identify the characters of the category in the input text. If a regular expression contains a Unicode general category, the JITted function passes to the function for the category the block of input text that is being processed. The function processes the input text and it returns to the JITted function a bit stream that marks the locations of all of the characters in the text that are included in the category.
Figure 3.2: Application Architecture of icGREP
3.2 Parser

The parser is a recursive descent parser that emits an AST consisting of objects that represent each of the constructs of the regular expression. The leaves of the AST are objects that represent the character classes of the regular expression and the character classes contain the code points of the characters of each character class. The AST from the parser represents the regular expression over code points.

In icGREP, when a character class is parsed from a regular expression, a new character class object is created and the code points for the characters of the character class are added to the new object. The character class object of icGREP maintains a sorted list of the code point ranges. Each code point range is represented by a data structure with a high and a low code point pair. The character class \([abc\text{-}d]\) represents a contiguous range of code points from \(a\) to \(d\). A hyphen, such as in the expression \([a\text{-}d]\) can also be used in order to express the same code point range. The hexadecimal code point value for the lower case “a” is \(0x61\) and the hexadecimal value for the lower case “d” is \(0x64\); therefore, the low code point value for the range \([a\text{-}d]\) would be \(0x61\) and the high code point value for the range would be \(0x64\). A character class can contain one or more code point ranges. Single characters are represented by a code point pair where the high and low code point values are the same.

3.3 UTF-8 Encoding

The UTF-8 encoder is a transformation that walks the AST generated by the parser and emits a new AST that represents the equivalent regular expression over UTF-8 byte sequences. For each code point that is contained within each character class, the encoder determines the number of UTF-8 bytes that are required to represent it, and then it splits the code point into a sequence of bytes, with each byte containing the necessary UTF-8 prefix. Each byte is then assigned to a new character class in the new AST. Take for example the regular expression ‘€ 100’. This regular expression would match the sequence of a euro character, followed by a space, a one, a zero, and another zero. The AST from the parser would represent this as a sequence of character classes with the following code points: \(0x20AC\), \(0x20\), \(0x31\), \(0x30\), and \(0x30\). With UTF-8, all of these code points would be encoded as single bytes, except for the euro character which would be encoded as three bytes. The AST
from the UTF-8 encoder would represent the euro character as a sequence of three character classes, each with one of the three bytes 0xE2, 0x82, and 0xAC. This would be then followed by the sequence of character classes for the bytes 0x20, 0x31, 0x30, and 0x30 that make up the rest of the regular expression. Using the \u{ } notation used to specify hexadecimal values in a regular expression, the regular expression with UTF-8 encoding would be represented as a sequence of bytes in the following form: \xE2\x82\xAC\x20\x31\x30\x30.

For a more complex example, consider the following regular expression that consists entirely of multibyte Unicode characters: `\u{244}[^\u{2137}]`. The AST from the parser would represent this as a sequence starting with the character class containing the code point 0x244 followed by the character class containing the range from 0x2030 to 0x2137. After the AST has been passed through the UTF-8 encoder this AST would become considerably more complex. The first codepoint in the sequence would be encoded as the two byte sequence `\u{C9}\u{84}`. The character class containing the range that follows, is a range of three byte sequences that is expanded into a series of sequences and alternations that are necessary to specify all of the possible byte encodings that would be contained within the range. The UTF-8 encoded regular expression for the range \u{2030}-\u{2137} would be encoded as follows:

```
\xE2((\x84[^\x80-\xB7])|([^\x81-\x83][^\x80-\xBF])|[^\x80[^\xB0-\xBF]])
```

Transforming the regular expression immediately after parsing from being a regular expression over code points into a regular expression over bytes simplifies the rest of the compiler, as the compiler then only needs to be concerned with single bytes as opposed to code points, which vary in size.

### 3.4 Nullable Optimizations

The nullable prefix and nullable suffix optimization transformations were added after it had been observed by one of the contributors to the prototype regular expression compiler, that regular expressions are at times created with prefixes or suffixes that may be removed, or modified while still providing the same number of matches as the original regular expression. Take for example the regular expression `([a-z]*[A-Z][0-9]*)`. Both the prefix `[a-z]*` and the suffix `[0-9]*` are nullable. The Kleene Star operator specifies a match of zero or more
of the expression that is to its immediate left. In this case the regular expression \([A-Z]\) that would result from removing both the prefix and the suffix, would provide the same number of matches as the original regular expression. The transformations also modify regular expressions by removing repetition operations from the prefix or suffix that are unnecessary. In the regular expression \(([a-z]+[A-Z][0-9]+)\) both the prefix and suffix use the + operator, which tells the regular expression engine to match either one or more of the character class that is to the operator’s immediate left. The regular expression only requires a single instance of the prefix or suffix to be matched, so the + repetition operators are unnecessary, and both can be removed. The transformations take as input an AST that represents a regular expression and output a new AST that represents a transformed regular expression without these unnecessary and computationally expensive operations.

3.5 Simplification Optimizations

The simplification optimizations are a series of four optimizations applied to the regular expression AST. The optimizations traverse the AST to flatten the sequence and alternation hierarchies, to combine sequences of character classes, and to canonicalize all forms of repetition into single repetition structures. When a regular expression is parsed, the parse tree from the parser presents a literal representation of the regular expression. Take for example the regular expression \(([0-9][0-9])/([0-9][0-9])\). This regular expression could be used to represent the date in the form of mm/dd. With regular expressions, parentheses are used to group parts of a regular expression together. This means that this regular expression would be represented in the AST as a sequence that would contain the sequence of the two characters classes of \([0-9]\) and \([0-9]\), followed by a character class for the character ‘/’, followed by another sequence of the two character classes of \([0-9]\) and \([0-9]\). The simplification optimization would flatten this structure by combining all three of the sequences into a single sequence. The simplified form of the AST for this regular expression would be of a single sequence containing the five character classes \([0-9][0-9]/[0-9][0-9]\). In the same way, alternations with sub alternations would be flattened into single alternations. Alternations such as \([a-z]|([A-Z]|[0-9])\) would become \([a-z]|[A-Z]|[0-9]\). In this case, with the character class combining simplification, the three alternate character classes in the regular expression would also be combined into the single character class of
[a-zA-Z0-9]. For regular expressions with nested repetition structures, such as in the regular expression \(([0-9]{3,5})\{3,5\}\) the repetitions would be flattened by the transformation into the following single repetition structure \([0-9]\{9,25\}\).

### 3.6 Reducer Optimization

The purpose of the reducer optimization is to reduce the number of redundant character class equations and bit streams. The redundancies are due to multiple instances of identical character classes that may be repeated within a regular expression. Without the optimization, character class equations and bit streams would be created for each instance of every character class. The optimization provides code improvements that are similar to the improvements that would be made by LLVM’s global value numbering optimization. LLVM’s global value numbering optimization uses value numbering to perform common subexpression elimination. It also performs redundant load elimination [2]. For regular expressions that contain multiple instances of single byte character classes, the LLVM IR that is generated after applying the reducer optimization is the same as the IR that would be generated if only the LLVM global value numbering optimization had been applied. For multibyte character classes though, the reducer optimization is able to provide code improvements that result in fewer instructions and less generated IR through the use parallel bit stream operations that have been added specifically for multibyte encodings. The global value numbering optimization is not able to provide these same reductions.

To implement the reducer, each character class object has a name method, and when the method is called, a name for the object is generated dynamically by concatenating the string representations of each of the code points of the character class. The input to the optimization is the AST from the simplification optimization, and when the AST is walked, a single copy of each uniquely named character class is moved into a dictionary that has been created for uniquely named regular expression objects. The optimization then creates and outputs a new AST, except with this AST all of the locations that formerly contained character classes objects now contain name objects. The name objects map directly to the single instances of the uniquely named character class objects that are now contained in the dictionary. This dictionary is then passed to the transformation that creates the character class equations so that character class equations are only generated once for each
unique named character class. The new AST with the name objects is then passed from the reducer optimization transformation as input to the transformation that creates the parallel bit stream equations to process the regular expression.

The dictionary that holds the uniquely named character class objects was designed to hold the more general regular expression objects instead of just the specific character class objects. The multi byte UTF-8 character classes are made up of sequences of character classes, with each character class of the sequence containing a single code point of a single UTF-8 encoded byte. For these UTF-8 character class sequences, named sequence objects have been created. In the same way that the character class object generates its name, the name method of the sequence object dynamically generates a name by concatenating together the names of the sequence’s character class objects. After the reducer optimization removes the named character class objects from the AST and replaces them in the new AST with name objects, uniquely named instances of the UTF-8 character class sequences are also moved to the dictionary of uniquely named regular expression objects. The named sequences are also then replaced by named objects in the new AST. After the dictionary for the uniquely named regular expression objects has been passed to the transformation to generate the character class equations for the uniquely named character class objects, it is then passed to a special method in the transformation that is responsible for the generation of the regular expression matching parallel bit stream equations to create the equations for the uniquely named UTF-8 character class sequences. Generating the parallel bit stream equations for the unique instances of the UTF-8 character class sequences in this way ensures that the marker bit streams for the multibyte characters are calculated only once. The ScanThru operation, as explained in section 3.9.1 is then used with the marker bit stream for each multibyte character class during the matching of the regular expression to combine, match, and advance the cursors for all instances of each multibyte character class all at the same time.

### 3.7 Character Class Equations

Character class equations are three-address equations that represent the parallel bit stream operations that are used to compute each character class bit stream from the set of eight basis bit streams that were transposed from the input text. The transformation that generates the character classes equations, transforms each of the the uniquely named character class
objects into a list of the Pablo statements and expressions that represent each of the bitwise operations that are required to generate the character class bit streams for each character class. The character class bit streams mark the locations of each character class in the input text. Figure 3.3 is an example of the character class equations that would be required to calculate the character class bit stream for the letter “a”.

\[
\text{INPUT: } a = [a]
\]

\[
\begin{align*}
\text{OUTPUT: } & \quad \text{temp1} = (\text{basis\_bits.bit\_1 \&} \text{\~ basis\_bits.bit\_0}) \\
& \quad \text{temp2} = (\text{basis\_bits.bit\_2 \&} \text{\~ basis\_bits.bit\_3}) \\
& \quad \text{temp3} = (\text{temp1} \& \text{temp2}) \\
& \quad \text{temp4} = (\text{basis\_bits.bit\_4 | basis\_bits.bit\_5}) \\
& \quad \text{temp5} = (\text{basis\_bits.bit\_7 \&} \text{\~ basis\_bits.bit\_6}) \\
& \quad \text{temp6} = (\text{temp5} \& \text{\~ temp4}) \\
& \quad a = (\text{temp3} \& \text{temp6})
\end{align*}
\]

Figure 3.3: Character Class Equations for the Character Class “a”

Figure 3.4 provides an example of the character class equations that would be generated for the character class that represents a digit in the range from 0-9.

\[
\text{INPUT: } \text{digit} = [0-9]
\]

\[
\begin{align*}
\text{OUTPUT: } & \quad \text{temp1} = (\text{basis\_bits.bit\_0 | basis\_bits.bit\_1}) \\
& \quad \text{temp2} = (\text{basis\_bits.bit\_2 \&} \text{basis\_bits.bit\_3}) \\
& \quad \text{temp3} = (\text{temp2} \& \text{\~ temp1}) \\
& \quad \text{temp4} = (\text{basis\_bits.bit\_5 | basis\_bits.bit\_6}) \\
& \quad \text{temp5} = (\text{basis\_bits.bit\_4 \&} \text{temp4}) \\
& \quad \text{digit} = (\text{temp3} \& \text{\~ temp5})
\end{align*}
\]

Figure 3.4: Character Class Equations for the Character Class [0-9]
3.8 Regular Expression Matching Equations

The regular expression matching equations are three-address equations that represent the parallel bit stream operations that are required to support the various constructs and operations of the regular expression. The transformation for the regular expression matching equations, transforms the AST that represents the regular expression from the reducer transformation into a list of the Pablo statements and expressions that represent the parallel bitstream operations that are required to match the regular expression. Figure 2.4 illustrated the matching process to match the regular expression \(a[0-9]*z\) when using parallel bit streams. Figure 3.5 shows the parallel bit stream equations that would be generated to specify that matching process. Marker bit stream \(M_3\) is the bit stream that would be returned with the matches that have been found in the input text.

\[
\begin{align*}
\text{INPUT: Regular Expression} & = a[0-9]*z \\
M_1 & = \text{Advance}( M_0 \ & a) \\
M_2 & = \text{MatchStar}( M_1, [0-9]) \\
M_3 & = \text{Advance}(M_2 \ & z)
\end{align*}
\]

OUTPUT: \(M_3\)

Figure 3.5: Parallel Bit Stream Equations for the Regular Expression \(a[0-9]*z\)

3.9 Unicode

3.9.1 Processing of Multibyte Sequences

Each bit position in the character class bitstream of a single byte ASCII character marks either the location of, or the absence of the ASCII search character. To match the location of a character the current position of the cursor is checked to see if the bit is set and then the cursor is advanced by one position. To match the position of a multibyte search character the procedure is different. For multibyte UTF-8 characters of length \(k\), it is the last \((k-1)\)th byte of the multibyte sequence in the bitstream that marks the location of the multibyte character. Figure 3.6 illustrates the process of matching a character class of a three byte character. The locations of the first two bytes of each character in the character class
CC have been marked with zeros. To match multibyte characters, first a *nonfinal* helper bitstream must be formed. The *Nonfinal* bitstream is formed by marking the locations of the first bytes of two byte sequences, the first two bytes of three byte sequences, and the first three bytes of any four byte sequences. The \texttt{ScanThru(current, nonfinal)} operation is then applied, in order to advance all of the current cursor positions to the locations of the \((k-1)\)th final character positions. To find any matches the result is then compared with the bits that are set in the UTF-8 character class bitstream. After this, the cursor is advanced by one position to be ready for the next matching operation.

\[
\begin{align*}
CC & \quad 001\ldots001\ldots\ldots \\newline
M_1 & \quad 1\ldots1\ldots1\ldots\ldots \\newline
\text{nonfinal} & \quad 11\ldots11\ldots\ldots \\newline
T_1 = \text{ScanThru}(M_1, \text{nonfinal}) & \quad \ldots1\ldots1\ldots\ldots \\newline
T_2 = CC \land T_1 & \quad \ldots1\ldots1\ldots\ldots \\newline
M_2 = \text{Advance}(M_1) & \quad \ldots1\ldots1\ldots\ldots
\end{align*}
\]

Figure 3.6: Processing of a Multibyte Sequence

Figure 3.7 shows how the MatchStar operation can be used to find all matches of a multibyte UTF-8 sequence. The problem is to find all matches to the character class CC that can be reached from the current cursor positions in \(M_1\). First we form two helper bitstreams *initial* and *nonfinal*. The initial bitstream marks the locations of all single byte characters and the first bytes of all multibyte characters. Any full match to a multibyte sequence must reach the initial position of the next character. The nonfinal bitstream consists of all positions except those that are final positions of UTF-8 sequences. It is used to "fill in the gaps" in the CC bitstream so that the MatchStar addition can move through a contiguous sequence of one bits. In the figure, the gaps in CC are filled in by a bitwise-or with the nonfinal bitstream to produce \(T_1\). This is then used as the basis of the MatchStar operation to yield \(T_2\). We then filter these results using the initial bitstream to produce the final set of complete matches \(M_2\).
3.9.2 If Hierarchy Optimizations

With each of the Unicode general categories containing such a large number of code points, an If Hierarchy optimization has been included in the statically compiled implementation of each category. The optimization works under the assumption that most input documents will only contain the code points of the characters from a small number of writing systems. Processing the blocks of code points for characters that exist outside of this range is unnecessary and will only add to the total running time of the application. The optimization tests the input text to determine the ranges of the code points that are contained in the input text and it only processes the character class equations and the regular expression matching equations for the code point ranges that the input text contains. The optimization tests the input text with a series of nested if else statements, using a process similar to that of a binary search. As the nesting of the statements increases, the range of the code points in the conditions of the if statements narrow until the exact ranges of the code points in the text has been found.

3.10 LLVM

The final steps in the compilation process are to transform the lists of the Pablo character class equations and the Pablo regular expression matching equations into LLVM IR, and then to JIT the IR into machine code in order to prepare it for execution. JIT compilation was used in order to provide the compiler with the ability to both compile a regular expression and to process an input file in a single high-level step. It was also used to provide portability. One of the future goals for icGREP will be to allow a user to run icGREP on machines with
different architectures, where icGREP will have the ability to adapt dynamically to the unique features of each architecture.

The LLVM IR Generator component takes the lists of the Pablo character class equations and the Pablo regular expression matching equations as input, and it returns as output a function pointer to a JITed function that can be called by the main processing loop to perform the block-at-a-time processing of an input file.

The LLVM module is the top level container in LLVM that contains type definitions, global variables, and functions. The bodies of functions are made up of one or more basic blocks, and the basic blocks are made up of straight line sequences of code. In icGREP, the LLVM Generator class starts by creating a new LLVM module and then LLVM instructions from the LLVM API are used to define all of the needed types, and to declare the application’s single JITed function. When the lists of the character class and regular expression matching parallel bit stream equations are processed, LLVM’s IR Builder class is used to generate the instructions in LLVM IR that are required for each of the Pablo statements and Pablo expressions that are contained in the lists. When LLVM adds a new instruction to a function, the instruction is appended to the end of a basic block of IR instructions.

The algorithms that were implemented in LLVM IR for each of the Pablo statements and Pablo expressions were based upon the algorithms implemented in the Parabix IDISA run-time libraries. LLVM provides near direct implementations of many of the functions found in the IDISA libraries. For example, a bitwise add operation of two 128-bit SIMD registers can be achieved with the \texttt{simd128<128>:add(x, y)}: function from the IDISA library. The same operation can also be achieved by performing a bitwise add of two 128-bit i128 operands with LLVM. In the prototype compiler, the Pablo statements and expressions were transformed by the Pablo compiler into C++ function calls that called the functions of the IDISA libraries directly.

After the LLVM IR has been generated by icGREP, an LLVM method is called to validate the IR and then LLVM optimization passes are applied. With LLVM, code improvements through optimizations can be achieved in a number of different ways. Using IR Builder to generate LLVM instructions means that optimizations, such as constant folding will be applied to the IR right when the LLVM instructions are created. If IR Builder sees an opportunity for constant folding, it will simply apply the constant fold, and instead of creating an instruction will return the constant instead [8]. Optimizations can also be applied by applying LLVM intra-procedural optimization passes to optimize the IR before it is passed
to the execution engine. An intra-procedural optimization pass provides optimizations on a function, independent of the rest of the application. The first intra-procedural optimization pass that is applied by icGREP is the mem2reg optimization pass. In addition to promoting the stack variables to SSA registers, the mem2reg pass also inserts phi nodes, and rewrites all of the loads and stores as appropriate. Using the mem2reg optimization pass provides a significant reduction in the total running time of the application. When studying the optimizations of LLVM, the source code of the LLVM opt utility was reviewed. In the source code of opt it was observed that after optimization passes such as mem2reg, that were capable of making significant transformations of the IR had been applied, a sequence of other additional optimization passes were often also applied. Based upon the patterns that were observed in opt, that were suggested in the LLVM documentation, and upon the results of performance evaluations, a sequence of additional optimization passes were also applied after the mem2reg optimization pass. First a basic alias analysis pass was added, to determine if multiple memory references point to the same memory locations. The information generated by the basic alias analysis pass is used by the global value numbering pass to help it eliminate redundant loads. The basic alias analysis pass was then followed by a instruction combining pass. The instruction combining pass was often repeated in Opt, and it works to form fewer and simpler instructions. The instruction combining pass is a peephole optimizer, that works to eliminate or reduce suboptimal adjacent instructions. After the instruction combining pass a global value numbering pass was also added to further eliminate redundant instructions and to eliminate the redundant loads.

An additional, more general way to achieve code improvement through optimizations is to adjust the code generation optimization level of the JIT execution engine. The execution engine has an option that allows the optimization level to be set to None, Less, Default or Aggressive. The LLVM code generator uses this optimization level to determine which optimization passes it should apply during the generation of the machine code. Each of the optimizations levels, except for None, cause a large number of additional optimizations to run on the generated code.

After the optimization passes have been applied, the LLVM execution engine is used to compile the generated function into machine code. A function pointer to this function is returned from the LLVM Generator class back to the main processing loop of the application. The main processing loop for icGREP is very much the same as the main processing loop of the prototype. The Parabix framework is used to transpose input text a-block-at-a time into
the basis bit streams, and the basis bit streams are passed to the JITted matcher function so that the characters in the text can then be processed in parallel.

The entire icGREP application could have been JITted all at once, but there are parts of the application that do not change, such as the code for the block-at-a-time processing of the input document. Instead, the only code that is JITted by icGREP is the code that needs to be compiled dynamically for each regular expression.
Chapter 4

Unicode Feature Evaluation

4.1 icGREP

The Unicode Consortium has published guidelines on how to adapt regular expression engines to use Unicode. The guidelines specify three levels of Unicode support. While icGREP currently does not meet all of the requirements of Unicode level 1 support the framework is in place for icGREP to become fully compliant. Currently, icGREP does not meet any of the requirements for Unicode level 2 or Unicode level 3 support.

To claim conformance with the first level of support a regular expression engine must support hexadecimal notation, properties, subtraction and intersection, simple word boundaries, simple loose matches, line boundaries, and code points.

Hexadecimal Notation: The support for hexadecimal notation is necessary in order to allow users to input all of the Unicode code points, even if a code point is not supported by a user’s keyboard. Keyboards in North America, for example, do not have a key for the euro character. The use of hexadecimal notation, such as in the notation \u{20AC} is supported by icGREP.

Properties: The support for Unicode properties by a regular expression is necessary as the Unicode character set is large. It would be impractical, and error prone, to expect a user to enter either all of the individual code points, or ranges of the code points, that exist for the characters of the different Unicode properties. Unicode properties provide a convenient way for users to express in a regular expression entire categories of Unicode
characters. There is a list of properties that a level 1 Unicode compliant regular expression engine must support. The list includes the Unicode general category property, scripts and script extensions, alphabetic, uppercase, lowercase, white space, noncharacter code point, default ignorable code point, any, ASCII, and assigned.

The most basic of the required properties is the Unicode general category property. The Unicode general category property provides a way to express in a regular expression a category such as “upper case letter”, which includes the broad category of all characters that are classified as upper case letters from any of the world’s writing systems. The Unicode general category property is currently supported by icGREP.

The scripts property, and the script extensions, provide a way to group together all of the characters that belong to each of the different writing systems. Examples of scripts include scripts for Latin, Greek, and Cyrillic. Characters that could be used with more than one script are included in the scripts called common or inherited. Characters that are used by more than one script, but that are not generic enough to be classified as common or inherited have their usage documented in script extensions. The use of scripts is not yet supported in icGREP. In the same way that the Unicode general categories have been statically compiled as was described on page 24, the scripts could also be statically compiled. A second option would be to create a dictionary that would include all of the code points and the code point ranges of each script. When a script is found by the parser, a character class would be created and the code points for the script would be added to the character class. The character class would then be passed through the UTF-8 encoder and the other transformation functions of icGREP, in the same way that other character classes are currently processed. For some regular expressions it may be beneficial to build the character classes for the scripts in this way in order to take advantage of the optimizations that the various transformation functions of icGREP are able to apply. If the scripts were generated in this way the if hierarchy optimizations would have to be added dynamically for the processing of the character class equations.

Other properties, such as the alphabetic, lowercase, uppercase properties are derived properties. The Unicode database contains either the explicit definitions of the code points or the code point ranges for these properties, or the properties can be composed from other Unicode properties. For example, the alphabetic property is derived from characters with the properties of lowercase, uppercase, letter title case, letter modified, letter other, number letter, or other alphabetic. The lowercase property itself is a derived property, made up
of characters with either the Lowercase Letter Unicode general category property or the Other_Lowercase property. There are a number of options for how these properties could be implemented in icGREP. In the same way that the Unicode general categories have been implemented, the derived properties could be explicitly defined and statically compiled. Another option would be to statically compile the nonderived components of each property, and then to use parallel bit stream equations to create predefined subexpressions for the derived properties. The lowercase property would be a predefined regular expression alternation that would be implemented internally as (Lowercase_Letter | Other_lowercase). This would simply be implemented as an extension of the mechanisms for the predefined subexpressions that currently exist within icGREP.

The parser would also have to be extended to support the additional properties that are required for Unicode level 1 support.

**Subtraction and Intersection:** The requirement for the support of subtraction and intersection is also due to the large size of the Unicode character set. The requirement requires mechanisms to be in place for set difference, union and intersection. An example of this would be of a user wanting to search for all uppercase letters that are not in the English language. The syntax for the Unicode general category property to specify all uppercase letters is \p{Lu} and the syntax to specify all characters that are English is \p{IsBasicLatin}. The user should be able to obtain the set difference between the two categories by subtracting with the notation \{\p{Lu} -- \p{IsBasicLatin}\} the set of all characters that are English from the set of all characters that are uppercase letters. To support the operators for subtraction and intersection the parser in icGREP would have to be extended. Also, new subclasses of the regular expression object would have to be added to icGREP to support the operations, but the Pablo expressions to support these operations are already in place. The subtraction operation is supported with the Pablo Not, which is implemented as a bitwise xor 1. The union operation is supported by the Pablo Or, which is implemented as a bitwise or, and the intersection is supported with the Pablo And, which is implemented as a bitwise and.

**Simple Word Boundaries:** To test for simple word boundaries, a regular expression engine must be able to test for the transition from word characters to non word characters. There must also be a way to test for end-of-line and start-of-line. With Unicode, the usual
class of word characters is extended to include characters that have either the alphabetic property, that belong to the Unicode decimal general category, are zero width joiners, or are zero width nonjoiners. Zero width joiners are nonprintable characters that indicate that two adjacent characters should be joined and treated as a single character. Zero width nonjoiners indicate that adjacent characters should not be joined. Creating a word characters class in icGREP to test for word boundaries could be achieved by creating a derived class for word characters, in the same way that the character classes for the derived character properties could be achieved. To support this, the grammar of the parser would have to be extended to enable it to accept a metacharacter such as \b which is used to indicate a word boundary in Perl. Testing for the end-of-line and start-of-line is currently supported in icGREP.

**Simple Loose Matches:** The requirement for simple loose matches applies to regular expression engines that support caseless matching. To support Unicode caseless matching, a regular expression needs to expand the search literal into a sequence of expressions that are a simple case-folding of the same value. The Unicode database contains a file that maps all simple case foldings. The code points for the mappings could be statically compiled into icGREP so that the alternate code points could be added to the character classes when the character classes of the regular expression are parsed.

**Line Boundaries:** To meet the requirement for line boundaries, regular expression engines that support line boundaries must support LF, CR, CRLF, NEL, PARAGRAPH SEPARATOR, and LINE SEPARATOR. Currently, icGREP only supports LF. The regular expression character class object that represents the LF is a predefined object that has been hard coded into icGREP. This character class object could be replaced with a regular expression sequence object, and each of the character class objects within the sequence could be used to represent each of the required line boundaries.

**Code Points:** The last requirement for a regular expression engine to be compliant with Unicode level 1 support, is that regular expression Unicode text must be interpreted as code points and not as code units. The character encoding is to be completely transparent to the user. To match a multibyte character, a user must be able to simply specify the code point for the character without having to specify the individual code units for the encoding that is being used. For example, to match the euro character the user should be able to
enter just the € character or the hexadecimal value 0x20AC instead of entering the encoded sequence of 0xE2 0x82 0xAC. This requirement has been met by icGREP.

The requirements for Unicode level 2 support for regular expression engines, is considered as extended support for Unicode. The requirements to meet Unicode level 2 support are requirements that were put in place in order to meet user expectations for sequences of Unicode characters. The primary requirements for this level of support are the support for canonical equivalence, extended grapheme clusters, and for the better detection of word boundaries.

Canonical equivalence is established when two strings are equivalent when normalized to Normalized Form Decomposition (NFD). Often a character has a single code point of its own, or the character can be assembled from a number of different code points. For example, the character ˜A has the code point 0x00C3. The sequence of the code point 0x0041 for the capital letter A followed by the code point 0x0303 for the combining tilde is considered to be canonically equivalent to the code point 0x00C3 as both have the same appearance when printed. A regular expression engine that supports canonical equivalence will treat either the single code point or the assembled sequence of code points as being the same.

Grapheme clusters are sequences of Unicode code points that a user will always identify as a single character. To implement this level of support the regular expression engine must provide the user with a mechanism to match against an arbitrary extended grapheme cluster, a literal cluster, and extended grapheme cluster boundaries.

Unicode level 2 support also has requirements for default case conversion, and wild cards in property values. There are also additional properties that must be supported such as name properties, properties that specify numeric attributes, shape and rendering, case, and direction.

Unicode level 3 specifies requirements for tailored regular expression support. A regular expression engine may be tailored for a specific locale, to be used by a specific groups of end users.

With what is now known about building a regular expression engine for grep using Parabix and LLVM, it is no longer a question of if these additional features can be added to icGREP. The question is whether or not the performance advantage of icGREP that has been observed in the performance evaluations can be maintained as the remaining features required for the three levels of Unicode compliance are added.
4.2 Alternate grep Implementations

The Perl regular expression engine is close to being fully compliant with the requirements for Unicode level 1 support. Perl supports hexadecimal notation, properties, simple word boundaries, and simple loose matches, but its support for subtraction and intersection is listed as experimental, and it does not include support for line boundaries. For Unicode level 2 support, Perl supports the requirement for name properties. Perl does not implement any of the features required for Unicode level 3 support [6]. Perl does have an implementation of grep, but it was not included in the comparative evaluations of this study as it can only be used to perform grep on lists rather than on files [18].

PCRE grep supports a subset of Perl’s Unicode support for regular expressions. PCRE supports hexadecimal notation, simple loose matches, the Unicode general category properties, scripts, and a number of the derived properties [5].

gre2p supports hexadecimal notation, simple loose matches, scripts, and the Unicode general category properties. gre2p does not implement any of the remaining Unicode properties that are required to be compliant with Unicode level 1 support [7].

egrep supports the Unicode general category properties and it supports the input of multibyte Unicode characters, but it does not support hexadecimal notation.
Chapter 5

Performance Evaluations

The performance evaluations of icGREP were conducted using an Intel Core i7-2600 (Sandy Bridge) processor (3.40GHz, 4 physical cores, 8 threads, 32+32 KB L1 cache, 256 KB L2 cache, 8 MB L3 cache) running the 64-bit version of Ubuntu 12.04 (Linux).

The performance metric reported in the evaluations is the cycles per byte used for each application run. The Ubuntu profiler tool called Perf was used to obtain the total number of processor cycles which was then divided by the number of bytes of the input document to determine the cycles per byte. The cycles per byte provides a performance metric that is independent of clock frequency. The fewer the number of cycles per byte, the better the performance of the grep implementation.

For all of the evaluations, the applications were run for a minimum of five times and the lowest number of cycles used was reported. For evaluations that provided a higher degree of variability in the results the number of application runs was increased up to the point where the lowest number of cycles reported became consistent.

The evaluations were conducted in a number of stages. In the first stage, the optimizations applied within icGREP were evaluated in order to determine the effectiveness of the optimizations on the dynamic compilation of regular expressions and on cycles per byte required to process the input document. The optimization transformations were evaluated, and the LLVM optimization passes were evaluated as well. The second stage of the evaluations, compared the performance of icGREP with the performance of the prototype regular expression compiler. The goal for this stage was to determine if the implementation of icGREP had been able to deliver upon the promise that had been shown by the prototype. The final stage of the evaluations, compared the Unicode matching performance of icGREP
CHAPTER 5. PERFORMANCE EVALUATIONS

45

with the Unicode matching performance of alternate grep implementations. The evaluations
did not compare the regular expression matching performance of icGREP with other grep
implementations when using ASCII, as these evaluations have already been conducted with
the prototype and have been documented in the paper Bitwise Data Parallelism in Regular
Expression Matching (Cameron et al., 2014) [21].

The regular expressions used to compare icGREP with the prototype were primarily the
same as the regular expressions that were used when the prototype was originally evaluated.
The regular expressions are listed in Table 5.1. The regular expressions are modified versions
of regular expressions that were sourced from the Benchmark of Regex Libraries. The
one new regular expression that was included in these evaluations was the email regular
expression sourced from www.regular-expressions.info. The email regular expression was
added in order to include a regular expression with an anchor and with boundaries. The
input files used were concatenated versions of the Linux 3Dfx howto file that was also used
for the original evaluations of the prototype.

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td><a href="0x"> </a>?([a-fA-F0-9][a-fA-F0-9])+[.,?! ]</td>
</tr>
<tr>
<td>Date</td>
<td>((0-9)(0-9)?/((0-9)(0-9)?/((0-9)(0-9)?(0-9)?))</td>
</tr>
<tr>
<td>Email</td>
<td>([^[])[a-zA-Z0-9-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,4}($</td>
</tr>
<tr>
<td>StarHeight</td>
<td>[A-Z](((a-zA-Z)<em>a[a-zA-Z]</em>[ ])[ ]<em>[a-zA-Z]<em>e[a-zA-Z]</em>[ ])</em>[a-zA-Z]* s[a-zA-Z]<em>[ ]</em>[.?!]</td>
</tr>
</tbody>
</table>

Table 5.1: ASCII Regular Expressions

The input files used for the optimization and Unicode evaluations were XML database
dumps from www.wikibooks.org. The database dumps are available for many different
languages.

Four regular expressions were used in the evaluations that compared the Unicode matching
performance of icGREP with the alternate Unicode capable grep implementations.
While most standard patterns use only ASCII, Unicode can be found in patterns that
search for specific Unicode characters or for sequences of Unicode characters. Three of the
regular expressions for this evaluation therefore were created to evaluate and compare the
performance of each grep implementation when using the standard regular expression oper-
ations of alternation, concatenation, and repetition. To obtain the specific characters used
in the regular expressions, the Unicode characters were selected at random from the 57 MB Japanese WikiBooks XML database download file that was used as the input file for the tests. The fourth regular expression was designed specifically to test the performance of the grep implementations when using a regular expression that contained Unicode general categories. The four regular expressions are listed together in Figure 5.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternation</td>
<td>(\u{707c}</td>
</tr>
<tr>
<td>Repetition</td>
<td>(\u{65e5}+)</td>
</tr>
<tr>
<td>Sequence</td>
<td>(\u{8fd4}\u{3057}\u{306b}\u{306a}\u{3063})</td>
</tr>
<tr>
<td>Unicode Categories</td>
<td>([^-.]\p{Lu}\p{Ll}+[!.?])</td>
</tr>
</tbody>
</table>

Table 5.2: Unicode Regular Expressions

5.1 Regular Expression Optimization Transformations

The following evaluations were performed to determine the effectiveness of the optimization transformations applied by icGREP to the regular expression AST. Two regular expressions were used for these evaluations: the URI regular expression, and the date regular expression. The regular expressions can be found in Table 5.1. The input file that was used was the 722.5 MB enwikibooks-20140606-pages-meta-current.xml input file from www.wikibooks.org. The cycles per byte used during the dynamic compilation and the cycles per byte used during the total running time of the application were both recorded. To perform the evaluations, all of the optimization transformation functions were first disabled, and then icGREP was run with each of the regular expressions in order obtain the application’s baseline performance. After this, each of the transformation functions were enabled, one after the other, and the effect of each additional transformation on the performance of the application was evaluated. The results of the evaluations can be seen in Figure 5.1.

The cycles per byte used during both the dynamic compilation and for the total running time of the application were the highest for both of the regular expressions when all of the optimization transformations were disabled. For both regular expressions, the cycles per byte for the dynamic compilation was more than double the number that was recorded after applying the first transformation. Much of this was due to the extra time that it took to generate the LLVM IR, and to JIT the IR to machine code. A significant penalty was being
paid for not applying the optimization transformations before generating the IR.

Figure 5.1: Regular Expression Optimization Transformations

5.1.1 Nullable

The first of the optimization transform functions applied to the regular expression AST by icGREP are for the nullable optimizations. For the URI regular expression, the nullable optimizations provided a 57% decrease in the cycles per byte used during the total running time of the application, and a 53% decrease in cycles per byte used during the dynamic compilation. The URI regular expression has the suffix of \((\[^\ ]+)/(\[^\ ]+)?|([\^\@]+)@([\^\@]+)\), which contains the optional Kleene star operation \\
(\[^\ ]*)? before the final alternation. The optional ? operator matches zero or more of the expressions to its left. The URI regular expression provides the same matches with or without this suffix, so the suffix is considered
to be nullable and is removed. In this case, the suffix contained a negated character class which is a computationally expensive operation, both during compilation and during the matching of the input document. The second part of the suffix, after the alternation contains the \((\[^@\]+)\) expression with the + operator at the end of the suffix. This expression only needs to match a single instance to return a positive match, so the + operator is also removed, further reducing the amount of work that icGREP must do while matching the input document.

For the date regular expression, the suffix \((\[0-9\] \[0-9\])?\) is optional, so this too is considered nullable and is removed. For the date regular expression, removing the nullable suffix provided a 4% decrease in the cycles per byte used during the total running time of the application, and a 9% decrease in the cycles per byte used for the dynamic compilation. Neither the URI regular expression or the date regular expression have nullable prefixes.

### 5.1.2 Simplification

The simplification transformation flattens the sequence and alternation hierarchies of the regular expression AST. It combines all of the repetition structures, and combines sequences of character classes. For the URI regular expression, the majority of the transformations occurred within the negated character class, which had been expanded by the UTF-8 encoder into a large hierarchy of sequences, alternations, and character classes. For the entire AST, the optimization reduced the number of sequences from 62 to 22, the number of alternations from 32 to 5, and the number of character classes from 88 to 76. The number of repetition structures in the AST remained the same. Applying the optimization reduced the cycles per byte used during the dynamic compilation by 16%, and the cycles per byte used during the total running time of the application was reduced by 23%.

The AST for the date regular expression was less complex than the AST for the URI regular expression, but the AST was still simplified by the simplification transformation. The number of sequences was reduced from 5 to 1, and the character classes were reduced from 10 to 8. The optimization resulted in a .4% increase in the cycles per byte used for the dynamic compilation, but it also provided a .01% decrease in the cycles per byte used during the total running time of the application.
5.1.3 Reducer

The AST that results from the simplification transformation often contains multiple instances of identical character classes. The reduction optimization takes as input the AST from the simplification optimization and it populates a new data structure with a single instance of each unique character class. The new data structure is then passed to both the character class transformation for the generation of the character class equations, and to the transformation for the generation of the regular expression matching parallel bit stream equations for the generation of the equations for the UTF-8 multibyte sequences. The reducer optimization is often responsible for a significant reduction in the total number of character class equations. For the URI regular expression, the number of character classes that were passed to the character class transformation was reduced from 76 to 43, and for the date regular expression the number was reduced from 8 to 2. The reduction resulted in a 10% increase in the cycles per byte used for the dynamic compilation of the URI regular expression, but it also provided a 3% reduction in the cycles per byte used during total running time of the application.

The URI regular expression contains multiple negated character classes which are expanded for the negation in order to select all of the characters in the Unicode code point range that are not included in the character class. The reducer reduced the multiple instances of the identical multibyte character classes that resulted from the expansion into single instances, with the result that each of the multibyte character classes equations only needed to be calculated once. During the matching of the input document, the ScanThru operation was then used to combine, match, and advance the cursors for all instances of each multibyte character class all at once, which has provided the observed increase in the matching performance. The increase in the cycles per byte used for the dynamic compilation occurred as a result of the additional overhead that was incurred for the elimination of the redundant multibyte character class sequences and for the generation of the parallel bit stream equations required to match each of the multibyte sequences.

For the date regular expression, the reducer optimization provided a 7.5% decrease in the cycles per byte used for the dynamic compilation, but it did not provide a significant change in the cycles per byte used during the matching of the input document. The date regular expression contains only single byte ASCII characters, so the LLVM global value numbering optimization would have reduced the number of character class equations before
the IR had been passed to the JIT execution engine even if the equations hadn’t already been reduced by the reducer optimization. The reduction in the cycles per byte used for the dynamical compilation of the regular expression was due to the reduction in the number of the character class equations that required IR generation, and the reduction in the amount of IR that had to pass through the LLVM optimizations up to and including the LLVM global value numbering optimization.

5.2 LLVM Optimizations

To evaluate the LLVM intra-procedural optimization passes, a baseline was first obtained by disabling all of the optimization passes, and then icGREP was used to process the 722.5 MB enwikibooks-20140606-pages-meta-current.xml input file with the email regular expression. After this, each of the optimization passes were enabled, one at a time, and the performance of the application was measured after the addition of each pass. During the evaluations of the passes, the generated IR was output to disk after each pass had been enabled, and the outputs were compared in readable form in order to observe the transformations that had been applied by each optimization. The first optimization pass that was enabled was the mem2reg pass, followed by the basic alias analysis pass, the instruction combining pass, and then the global value number optimization pass.

The optimization pass that provided the greatest increase in performance was the mem2reg optimization pass. Before applying the mem2reg pass the generated IR was 912 lines in length and contained 225 loads and 221 stores to memory. After applying the mem2reg pass, this was reduced to 632 lines of IR with 119 loads and 93 stores. icGREP used 3.47 cycles per byte to process the input file before applying the mem2reg optimization pass and 2.178 cycles per byte were used after the optimization had been applied. The application of the mem2reg optimization alone, without applying any of the other LLVM optimization passes provided a 37% decrease in the cycles per byte used when processing the file. The instruction combining pass was applied after the mem2reg optimization pass, and it reduced the number of generated lines of IR to 536 lines with 85 loads and 65 stores. The cycles per byte used to process the input file was further reduced down to 2.177 cycles per byte. The global value numbering optimization pass was applied last, and it provided a final reduction down to 419 lines of IR with 65 loads. The number of stores did not change after applying the global value numbering optimization pass. The global value numbering optimization
pass provided only a small additional reduction in the cycles per byte used to process the input file. After adding the global value numbering optimization pass, icGREP used 2.172 cycles per byte to process the input file with the regular expression.

Evaluations were also conducted with the code generation optimization levels that could be applied to the JIT execution engine. During the evaluations, different optimization levels proved more effective than others depending upon the regular expression that had been used. Tests were therefore conducted with all of the regular expressions from Table 5.1. The enwikibooks-20140606-pages-meta-current.xml file was the input file that was used for these evaluations. The optimization levels that provided the lowest number of cycles per byte proved to be either the optimization levels of None or Less. With None the execution engine does not apply any additional optimizations. The pattern that emerged from the evaluations was that the regular expressions that were either large or complex required the optimization level of Less, while the rest of the regular expressions performed better with None. The email and the date regular expressions performed better with no optimizations, while the hex, URI, and starheight regular expressions performed better with the Less level of optimizations. The starheight regular expression, in particular with its nested repetitions used 6.06 cycles per byte when no optimizations were applied, and 4.4 cycles per byte when the optimization level was set to Less.

5.3 icGREP vs. the Prototype

5.3.1 Asymptotic Performance and Dynamic Compilation

For the evaluations that compared the performance of icGREP with the performance of the prototype, the input files used were concatenated versions of the Linux 3Dfx howto file. Tests were conducted with ten different file sizes for each regular expression. The files ranged from 75 MB to 750 MB in size. All of the evaluations compared the relative performance of the two applications in reported CPU cycles per byte. For icGREP the total number of cycles used was recorded both for each complete run of the application, and for just the dynamic compilation. The number of cycles used for the dynamic compilation was then subtracted from the cycles used during the complete application run in order to determine the cycles used for the matching of the input document. It is the cycles per byte for the matching that provides an “apples-to-apples” comparison of the two applications. The compilation of the regular expression for the prototype occurred off-line and was not
included in the evaluations. The values reported for the prototype only represent the cycles per byte used during the matching of the input document. Figure 5.2 shows the results of the comparisons between icGREP and the prototype.
Figure 5.2: icGREP vs the Prototype Regular Expression Compiler

For all but one of the regular expressions, the cycles per byte used during the matching of
the input file was less with icGREP than with the prototype. For the URI regular expression, during the matching of the input file, icGREP used almost twice the number of cycles per byte. What accounted for this difference was the use of the negated character class in the URI regular expression. For a negated character class such as \[^\ ] icGREP will match every code point except for the code point of the empty space. When there are only 127 code points, such as with ASCII, this is not a problem, but with Unicode icGREP will search for more than a million different code points. Figure 5.3 shows a comparison of icGREP with the prototype when matching the URI regular expression with the UTF-8 encoding in icGREP disabled. With the UTF-8 encoding disabled, the number of cycles per byte used by icGREP was less than the cycles per byte used by the prototype. For the other regular expressions the use of the UTF-8 encoding only had a minimal impact upon the matching performance of icGREP. When compared with other grep implementations, such as egrep, nr-grep, pcregrep, and gre2p, icGREP provided the fastest matching performance when matching the URI regular expression, even when the UTF-8 encoding was enabled. When processing the 722.5 MB input file enwikibooks-20140606-pages-meta-current.xml with the URI regular expression, icGREP used 4.56 cycles per byte, while icGREP’s closest competitor was nr-grep which used 11.09 cycles per byte.

![Figure 5.3: URI with icGREP and ASCII encoding](image-url)
5.4 Unicode Performance

5.4.1 Comparisons with other Unicode Capable GREP Implementations

The results of the Unicode performance evaluations can be found in Figure 5.4. The regular expressions for the evaluations are in Table 5.2. In all cases icGREP used fewer cycles per byte than the alternate grep implementations, except when using the Unicode sequence regular expression where egrep was faster. The Boyer-Moore algorithm is used by egrep. The Boyer-Moore algorithm uses character skipping, which provides egrep with an advantage for fixed string searches [39]. In all other cases though, the performance of icGREP proved superior. For the regular expression that used repetition, icGREP used fewer than half the cycles per byte of its closest competitor. For the regular expression that used alternation, the next closest competitor used five times the cycles per byte, and for the regular expression that used the Unicode general category, icGREP used 6.4 cycles per byte while the closest competitor used over 100 cycles per byte.

Figure 5.4: UTF-8 Processing Comparisons
5.4.2 The If Hierarchy Optimization of Named Classes

The If Hierarchy Optimization has been included in the Unicode general category implementation so that the only character class equations and regular expression matching equations that are processed for a Unicode category are the equations for the ranges of the characters that are found in the input text. In the Unicode database, the general category “Lu”, for Uppercase Letter contains 1441 code points. If an input document contains text that is only in ASCII, the text will contain a maximum of 26 code points for upper case letters. Processing the character class equations for the other 1415 code points would not be necessary. In an attempt to determine the optimal selection of the code point ranges for the if test conditions, three different if hierarchy configurations were evaluated. The results of the evaluations can be seen in Figure 5.5. To determine the base line performance, a Unicode general category implementation was created that did not include the if hierarchy optimization. The evaluations without the if hierarchy have been labeled as “Flat”. The second implementation, that is labeled “Simple”, included an if hierarchy that only tested for characters that were in the single byte, two byte, three byte, or four byte ranges. The third implementation, labeled “Full”, used a more granular approach in its if hierarchy implementation. The conditions in the “Full” if hierarchy tested for the code point ranges of Unicode code blocks that would be used by individual writing systems. Two different input files were used for the evaluations: a 19 MB Greek WikiBooks XML database download file, and the 57 MB Japanese XML database download file. As both of the files are XML files, both contained characters that are in ASCII, or Basic Latin, as well as characters that are in Greek or Japanese. The regular expression that was used for the evaluations was the Unicode Categories regular expression from Table 5.2. The cycles per byte count was recorded for the total running time of the application. For both input files, it was found that a cycles per byte count could be achieved with the more granular “Full” if hierarchy approach that was half of the cycles per byte count that was achieved when the if hierarchy wasn’t used at all. It was interesting to note that even though the cycles per byte used by the implementation without the if hierarchy was over twice that of the implementation with the “Full” if hierarchy, that the implementation without the if hierarchy was still much faster than the next closest alternate grep implementation. With the Japanese XML input file and with the Unicode Categories regular expression pcregrep used 101 cycles per byte while processing the input file while the icGREP implementation without the if hierarchy
used only 14.4 cycles per byte. This occurred despite the fact that during the processing
of the input file, that icGREP had to process the character class equations for 1441 code
points for every block of input text.

![Figure 5.5: Unicode If Hierarchy Optimization](image_url)
Chapter 6

Conclusion

6.1 Summary

This thesis set out to show that a unified, Unicode capable regular expression matching application built with LLVM and Parabix technologies could be performant. The goal was to build upon the potential that had been demonstrated by the prototype research compiler that used the Parabix framework to apply bitwise data parallelization to the search for string patterns in text. The new compiler unified the tool chain of the prototype through the use of LLVM, and new algorithms were developed in order to add Unicode support.

Performance evaluations of the new compiler, demonstrated that the matching performance of the compiler was able to meet and in most cases exceed the pattern matching performance of the prototype. For input files that were 300 MB or larger, the cycles per byte used by the new compiler for both the dynamic compilation and the matching of the patterns in the input text, proved to be less than the cycles per byte used by the prototype for the matching of the input text alone. The one case where the new compiler did not match the performance of prototype, was when UTF-8 encoding was enabled and the regular expression contained a negated character class. Even with the UTF-8 encoding enabled though, and with the same regular expression, the cycles per byte used by the new compiler was still less than the cycles per byte used by other grep implementations. The other grep implementations that were used during the evaluations were implementations that were all based upon the traditional methods of searching for string patterns in text using finite automata.

Performance evaluations that compared the Unicode matching performance of the new
regular expression compiler with the other Unicode capable grep implementations demonstrated that the new compiler had a distinct performance advantage. The only time that the performance of the compiler was bested was when the regular expression consisted solely of a sequence of multibyte Unicode characters. In this case, the grep implementation that used the Boyer-Moore algorithm that allowed for character skipping had an advantage. In all other cases though, the performance of the new compiler proved superior.

### 6.2 Future Work

One aspect of the design of icGREP that emerged during the comparative evaluations as an area that could be improved was the strategy that had been used to implement the negation of a character class. The strategy had worked well when applied to ASCII, but it proved inefficient when applied to Unicode. The strategy expands a negated character class to select every character in the Unicode code point range except for the character, or characters contained within the character class. There are a number of potential resolutions to this that should be explored. One way would be to apply the *if hierarchy* optimization to the expanded character class, similar to way that the optimization has been applied to the Unicode general categories. Another option would be to explore the possibility of negating a character class using the same strategy that has been used to negate the Unicode general categories. The Unicode general categories use the parallel bit stream equations to apply a bitwise XOR operation to the bit stream that marks the locations of the characters that are contained in negated category.

There are also additional transformations such as factorization, that could be applied to the regular expression AST. For a regular expression such as `abcd|afgh|awx|bk|blt`, the regular expression could be refactored into the following: `a(bdc|fgh|wx)|b(k|lt)`. With the regular expression in its original form, regular expression matching parallel bit stream equations would be generated to check for the letter `a` at the beginning of three sequences and to check for letter `b` at the beginning of two sequences. In the refactored form, parallel bit stream equations would only need to be generated to check once for both the `a` and the `b`.

Portability is also an issue that should be addressed in the future. One of the reasons that LLVM was used was that LLVM with JIT compilation provides the opportunity to implement a solution that is portable. The current implementation of icGREP is hard
coded to use 128-bit SSE2 SIMD technology. Now that it has been demonstrated that it is possible to build a high performance application for grep using Parabix and LLVM changes should be made so that the icGREP can transparently adapt to other architectures.

When building the components of icGREP that provide for the generation and the JITting of LLVM IR, the decision was made to allocate all memory variables on the stack and then to use the LLVM mem2reg optimization to promote the memory references to SSA registers. This strategy proved to be effective at simplifying the construction of an LLVM based system, but it necessitated the use of the mem2reg optimization pass and the other LLVM optimization passes that followed. The components of icGREP that generate the LLVM IR should be rewritten so that all variables are allocated to SSA registers directly, in order to eliminate the need for the additional LLVM optimizations passes.

Full Unicode level 1 support should be added to icGREP, followed by Unicode level 2 support. A framework should also be put in place in order to add Unicode level 3 support for individual user locales.
Bibliography


