Implementing BREeze - a High-Performance Regular Expression Library Using Code Generation with BuildIt

by

Tamara Mitrovska

B.S., Computer Science and Engineering, Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

© 2023 Tamara Mitrovska. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Tamara Mitrovska
Department of Electrical Engineering and Computer Science
May 12, 2023

Certified by: Saman Amarasinghe
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Implementing BREeze - a High-Performance Regular Expression Library Using Code Generation with BuildIt

by

Tamara Mitrovska

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Regular expression matching is a very common problem in software engineering, with applications in text processing, text searching, data scraping, syntax highlighting, deep packet inspection in networks, etc. Due to the varying complexity of regular expressions, having one general approach to match all types of expressions is usually not enough to get the needed performance for software applications. Many modern regular expression engines have tried to solve this problem by combining different algorithms and optimization techniques, which in most cases result in very complicated and large codebases. As a result, we introduce BREeze, a fully functional regular expression library implemented in just around 1500 lines of code with comparable performance to the modern regular expression engines. BREeze is implemented on top of BuildIt, a multi-stage code generation framework that makes it possible to generate high-performance, specialized code while keeping the implementation simple.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

I would like to express my gratitude to professor Saman Amarasinghe for accepting me in his research group and offering invaluable support and guidance throughout this project. I would also like to thank Ajay Brahmakshatriya, a Ph.D. student and author of BuildIt, for the numerous meetings and his continuous advice and suggestions that kept me motivated and helped me improve various aspects of this project. I further extend my appreciation to Alice Chen, a UROP student who helped with the implementation of important parts of BREeze.

Additionally, I would like to thank all my mentors and advisors throughout my time at MIT who have helped me attain the knowledge and skills required to tackle this project.

Finally, I express immense gratitude to my friends and family whose love and support mean the world to me. I am deeply grateful for all of their encouragement that kept me going throughout my time at MIT and shaped me into the person that I am today.

Contents

List of Figures

List of Tables

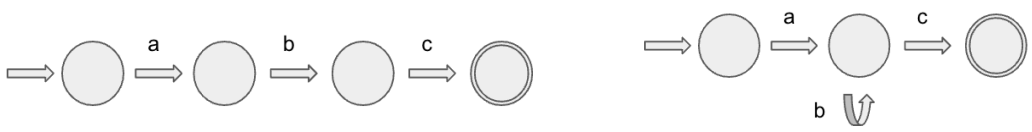
Chapter 1

Introduction

Regular expression (regex) matching is a very old problem with ubiquitous applications in software engineering. Due to the wide range of regular expressions, one general implementation approach does not always yield the best performance for matching all types of patterns. For example, regex libraries implemented with simple backtracking do not perform well for expressions with high ambiguity [10]. As a result, there have been numerous techniques throughout the years attempting to solve the problem in both time and space efficient manner ([17], [6], [18]). Most of these approaches build upon the same underlying principle that regular expressions can be represented as finite automata (FA).

1.1 Regular expressions as finite automata

There are two different types of FA: deterministic (DFA) and nondeterministic (NFA). In DFA there is at most one possible transition from each state. For example, we can represent the regex `abc` as a DFA because starting from any character, there is only one possible character that we can transition to. Matching an input string against a DFA takes linear time in the length of the string because we can simultaneously walk through each input string character and the corresponding state in the DFA (this process is called DFA simulation). However, DFA simulation does not work for nondeterministic regular expressions like `ab*c` where we can have multiple possible



(a) DFA example: `abc` (b) NFA example: `ab*c`

Figure 1-1: Regular expressions as FA.

transitions from a single state. In this case, we have to represent the regex as an NFA.

The most common way to match an input string against an NFA is backtracking implemented with depth-first search (DFS), which can have a worst-case exponential runtime in the length of the input string. Consider matching the string `abccc` with `.*b.*`. The first `.*` part of the regex will consume the entire string, so when we try to match `b` next, we will need to backtrack from the end of the input string all the way to index 1 (where `b` is). In the case of a much longer input string, this approach can result in an exponential blowup which is also known as catastrophic backtracking [10]. Despite the time inefficiency, backtracking is very easy to implement which is why it is commonly being used in the regex matchers of programming languages like Python and Java [10].

More sophisticated regex matchers try to combine the advantages of the NFA and DFA approaches by doing on-the-fly NFA to DFA conversion (also known as Thompson's algorithm or Thompson's NFA simulation [14]). This approach finds all the states that can be reached from the current NFA state and merges them into a single DFA state. This has two main advantages. First, we no longer need to store the entire NFA or DFA, it is enough to store only the current subset of reachable states which improves memory efficiency. Second, we can now process the regex in a breadth-first search (BFS) manner as opposed to depth-first search used in backtracking. Using BFS helps avoid catastrophic backtracking because after matching each character in the input string, we now consider all the possible states we can go to before getting any deeper into the search. In chapter 4 we explain how we adapt this approach for our implementation.

1.2 Code specialization

Thompson's algorithm is not always enough to achieve the regex-matching performance required by some software applications. The best-performing regex matchers try to improve performance by applying tricks like splitting the regex into deterministic and non-deterministic parts, using bit masks to represent the states ([17], [9]), etc. which often result in very long and complicated source code which is hard to understand and maintain. Despite that, due to the variety of regular expressions and their complexity, tailoring the implementation to the specific regular expression helps a lot in terms of performance. For example, one such optimization in RE2 [9] is for patterns where every possible match starts with the same character, as in `(dragon|dog)`. When looking for this pattern in a long text, the matching would normally have to be run starting from each character in the text. However, in this specific case, to avoid unnecessary looping over the text, RE2 uses `memchr` first to find an occurrence of the first character of the match (in this example `d`), and then runs the normal matching procedure from there [9].

1.3 BuildIt

To simplify the implementation while still being able to obtain specialized code, we are using BuildIt [7] - a type-based multi-stage code generation framework available as a C++ library. BuildIt allows for easy implementation of libraries and domain-specific languages (DSLs) because it does not require any compiler modifications which domain experts are not usually very familiar with. BuildIt is type-based which means that the code staging is controlled by 2 types of variables: static and dynamic. For example, a two-stage approach to generating code would look as follows: we declare all the variables that we want to be evaluated in the first stage as static, and all the variables that we want to appear in the generated code for the second stage as dynamic. Figure 1-2 shows an example of staging a simple function taken from [7] and figure 1-3 shows an example of the corresponding generated code.

```
dyn_var<int> power(dyn_var<int> base, static_var<int> exponent) {
    dyn_var<int> res = 1, x = base;
    while (exponent > 1) {
        if (exponent % 2 == 1)
            res = res * x;
        x = x * x;
        exponent = exponent / 2;
    }
    return res * x;
}
```

Figure 1-2: First-stage code for computing the power of a number. The exponent is static and the base is dynamic.

```
int power_15 (int base) {
  int res = 1;
  int x = base;
  res = res * x;
  x = x * x;
  res = res * x;
  x = x * x;
  res = res * x;
  x = x * x;
  int var3 = res * x;
  return var3;
}
```

Figure 1-3: Generated code for the power function from figure 1-2 after supplying 15 as the static exponent argument. The code raises `base` to the power of 15.

BuildIt generates code by repeatedly executing the first-stage code and following each control flow path to construct the AST of the program to be generated. Therefore, the main idea is that by declaring a regular expression as static, we are able to generate specialized code for it using BuildIt. Apart from specifying variables as static or dynamic, BuildIt does not require any other syntax changes compared to a normal C++ program which keeps the implementation very simple. This makes it very easy to add new optimizations with very minimal changes to the source code, as we will see in chapter 5.

1.4 Contribution

This work has two main contributions. First, we manage to keep our implementation very short and simple compared to the existing regex libraries while achieving comparable and in some cases even better performance. We achieve this by implementing a variation of Thompson's algorithm described in section 1.1 and staging it with BuildIt. BuildIt keeps the source code simple, and at the same time generates highly specialized code specific to a given regular expression which leads to good performance.

Second, we allow the user to control the code generation process by providing special scheduling options which are not available in standard regex libraries. This is again possible due to BuildIt because it lets us generate significantly different code with very minimal changes to the implementation. These changes usually include adding extra static flags or converting static variables to dynamic ones and vice versa. The scheduling options that we provide can be combined to generate many different schedules that the user can pick from to optimize the matching performance of a given regular expression.

The rest of this paper is organized as follows. In chapter 2 we give an overview of three modern regular expression libraries relevant to our implementation and performance analysis. In chapter 3 we introduce the syntax and API of BREeze. In chapter 4 we explain the implementation in detail and in chapter 5 we go over the supported scheduling options. Finally, in chapter 6 we present our implementation and performance analysis and in chapter 7 we summarize our work and talk about limitations and future steps.

Chapter 2

Related Work

This section gives a brief overview of three popular regular expression libraries that we compare our implementation and performance to. PCRE2 [12] is an updated version of PCRE - a general and widely used regular expression library. We also take a look at Hyperscan [17] and RE2 [9] which are more specialized and performance-oriented than PCRE2.

2.1 PCRE2

PCRE2 is the latest version of the Perl Compatible Regular Expressions (PCRE) library implemented in C. It is one of the most popular regex libraries and the basis of the regex features included in the matchers of many languages like PHP, R, and Delphi [11]. It supports two different matching algorithms. The standard algorithm is an NFA algorithm that performs backtracking with DFS. The alternative algorithm is considered a DFA algorithm - it keeps track of multiple active states at a time and performs BFS to find a match on the input string. Some features like choosing between greedy and lazy matching that are present in the standard algorithm are not relevant for the alternative algorithm [12]. To match a regular expression with PCRE2, it first needs to be compiled into a binary version which is passed to one of the matching functions to match against a string. Our library supports only a subset of the features available in PCRE2. Since PCRE2 is widely used, we follow its syntax

conventions very closely in our implementation. Section 3.1 and specifically table 3.1 give an overview of the regular expression syntax that we support. The PCRE2 implementation has around 130K lines of code. In chapter 6 we show that BREeze performs consistently better than PCRE2. With just around 1500 lines of code, we can obtain up to 5× speedup compared to PCRE2 when using unoptimized schedules and on average 20× speedup with tuned schedules.

2.2 RE2

RE2 [9] is a regex library developed in C++ by Google. It was implemented for production use, therefore its main goal is to be able to handle regular expressions from different types of users by limiting the amount of memory used, to avoid problems like stack overflowing [4]. It implements an optimized version of Thompson's algorithm that includes caching of already-seen DFA states. RE2 guarantees linear time execution in the length of the matching string. It first parses and simplifies the regex into a more optimal version and then compiles it into an NFA. The compiled object is then used to match an input string [9]. To prioritize efficiency, RE2 does not implement some of the features available in PCRE2 like backreferences. RE2's source code has around 26K lines of code [4]. Our library shares a lot of similarities with RE2 in terms of syntax because it is also based on PCRE. As we will see in chapter 6, we obtain a similar performance to RE2. With untuned schedules BREeze is at worst 1.6× slower than RE2; however, with tuned schedules, it is on average around 1.85× faster than RE2.

2.3 Hyperscan

Hyperscan [17] is a high-performance regular expression library developed by Intel. It is primarily optimized for use in deep packet inspection (DPI). Unlike other regular expression libraries, it is highly optimized for multi-pattern matching and streaming. Hyperscan performs graph decomposition on the regex to split it into literal strings

and finite automata components. For the literal strings, it supports multi-string shift-or matching which exploits SIMD operations for parallel matching. To match the FA components Hyperscan uses a bit-based NFA matching algorithm (similar to Thompson's algorithm) which also leverages SIMD operations [17]. These techniques result in a high-performance regex matcher; however, the source code (written in C and C++) contains around 187K lines of code which is very hard to understand [1]. Using a much simpler and more concise implementation, we were still able to support some of its features through our scheduling options, like splitting the string into blocks and parallel matching. As we will see in chapter 6, Hyperscan, in general, has the best performance among BREeze and the libraries presented above. On average BREeze is around 7× slower when using serial schedules. However, we will show that parallelizing our code significantly improves our performance and achieves up to 1.75× speedup compared to the serial Hyperscan running times.

Chapter 3

Introducing BREeze

In this section, we summarize the syntax and API of BREeze [1] and highlight the main differences from PCRE2 and RE2. Hyperscan's API has significant differences compared to other standard regex libraries and therefore we do not follow most of its conventions.

BREeze is primarily intended for matching against ASCII-encoded strings. However, we also support regular expressions with hexadecimal characters which can be used to match characters outside of the ASCII range based on their code.

3.1 Syntax

BREeze closely follows the PCRE2 and RE2 syntax. It supports a subset of their operators and expressions shown in table 3.1. BREeze supports all the basic features such as repetition, alternation, and character classes. Characters that have special meaning in the syntax like * are preceded by a backslash to be matched as literals. When a normal alphabetic character is preceded by a backslash it represents a special character class as described in table 3.2.

Our matching is non-greedy (also called lazy) by default, which means that we return the first found match based on our algorithm which is not necessarily the

[1] The first part of the name BREeze stands for BuildIt Regular Expressions (BRE). The rest of the name was chosen to hint towards the simple and easy implementation.

Expression	Description
`.`	any character except newline
`x?`	zero or one `x`
`x+`	one or more `x`
`x*`	zero or more `x`
`(x|y)`	`x` or `y`
`[xyz]`	character class
`[^xyz]`	negated character class
`[a-z]`	character range
`[^a-z]`	negated character range
`x{n}`	`x` repeated `n` times
`x{n,}`	`x` repeated `n` or more times
`x{n,m}`	`x` repeated between `n` and `m` times inclusive
`\d,\w,\s,\D,\W,\S`	special character classes

Table 3.1: Supported expressions and operators.

Expression	Character class
`\d`	`[0-9]`
`\D`	`[^0-9]`
`\w`	`[a-zA-Z0-9_]`
`\W`	`[^a-zA-Z0-9_]`
`\s`	a single space
`\S`	anything except a single space

Table 3.2: Special character classes.

longest match. Greedy matching on the other hand reports the longest found match. For example, a lazy match of `ab*` in `abbc` is `a` and a greedy one is `abb`. We support greedy matching as well, but it needs to be turned on with a special flag as described in section 4 where we go more into detail about why we choose to default to lazy matching. In contrast, PCRE2 and RE2 return greedy matches by default and provide extra notation to represent lazy matching. For example, `ab*c` is matched greedily with PCRE2 and RE2, and lazily with BREeze. Lazy matching in PCRE2 and RE2 can be turned on by adding a question mark after the repetition operator like `ab*?c`.

Another difference between our implementation and the above-mentioned libraries is in alternation. In case of multiple matches, PCRE2 and RE2 have a preference over the available options. For example, when matching `(abc|def)` they prefer the leftmost option `abc` over `def`. As a special design choice, BREeze reports the first

found match disregarding the alternation ordering.

Any other features present in PCRE2 like backreferences, lookahead assertions, named groups, etc. are not currently supported in BREeze. This however does not impede the usability of our library since we are still able to express a wide range of regular expressions as we will see in chapter 6. Most of these features like backreferences and lookahead assertions are not present in RE2 either. Moreover, from our experience implementing BREeze, these features could very easily be added in the future.

Option	**Description**
`ignore_case`	when true, caseless matching is turned on (default: false)
`dotall`	when true, `.` matches newline as well (default: false)
`greedy`	when true, greedy matching is turned on (default: false)
`block_size`	when set to a positive integer the string is split into that many blocks which are matched in parallel (default: -1)
`interleaving_parts`	matching stride for partial matches, distinct parts can be matched in parallel (default: 1)
`flags`	a string of the same length as the regex; splitting and grouping scheduling options are specified by marking specific positions with special characters like `s` or `g`

Table 3.3: Fields of the RegexOptions struct.

A full list of flags that are currently supported is given in table 3.3. The `dotall`, `ignore_case` and `greedy` flags have their equivalent or similar flags in PCRE2, RE2 and Hyperscan. The rest of the flags are specific to our library and are used to control the scheduling options described in section 5.

3.2 API

BREeze takes a two-step approach towards regex matching: it first compiles the regex into a function object and then runs the compiled function to find a match. For convenience, we provide a single `match` function that performs both of these steps. However, we also provide helper functions for running the compilation and matching steps separately. The `match` function takes the following arguments:

```
string regex = "(ab|cd)(12)*";
string text = "23cd12123";

RegexOptions default_options;
int partial = match(regex, text, default_options, MatchType::
    PARTIAL_SINGLE); // returns 1
int full = match(regex, text, default_options, MatchType::FULL); //
    returns 0

int res = 0;
RegexOptions opt;
opt.greedy = false;
string word;
res = match(regex, text, opt, MatchType::PARTIAL_SINGLE, &word);
cout << "Lazy match: " << word << endl; // prints "cd"

opt.greedy = true;
string longest;
res = match(regex, text, opt, MatchType::PARTIAL_SINGLE, &longest);
cout << "Greedy match: " << longest << endl; // prints "cd1212"
```

Figure 3-1: Example code for using the `match` function.

- `string regex` - a regular expression
- `string str` - the string in which to look for matches
- `RegexOptions options` - the available flags to be set by the user provided in table 3.3; they can be used to turn on the scheduling options given in section 5
- `MatchType match_type` - an `enum` type to choose between full and partial match; it has 2 available options: `FULL` and `PARTIAL_SINGLE`
- `string* submatch` - a pointer to a string that will hold the found match; this is an optional parameter and it is set to `nullptr` by default

The function returns an integer `1` or `0` corresponding to match or no match respectively.

3.2.1 Full match

The full match option can be specified by passing `MatchType::FULL` to the `match` function. It checks whether the regular expression exactly matches the input string.

It is equivalent to the RE2 `FullMatch` function.

3.2.2 A single partial match

This option can be activated by passing `MatchType::PARTIAL_SINGLE`. If a `submatch` string pointer is not passed, it returns `1` as soon as it finds any partial match and `0` otherwise. If `submatch` is passed by the user and the `greedy` option is false, it fills `submatch` with the first found match which is often the same as the first shortest match. If `greedy` is true, it finds the first longest match. This option is equivalent to the `PartialMatch` function in RE2.

3.2.3 All partial matches

We provide a `get_all_partial_matches` function which takes in a regex, a string, and a `RegexOptions` object and returns a vector of non-overlapping partial matches. This function works the same as repeatedly calling the `match` function with the `PARTIAL_SINGLE` option and advancing the start of the string pointer after the end of each found match to find the next one. This option gives the same results as repeatedly calling the `FindAndConsume` function in RE2.

Chapter 4

Implementation

The following steps take place when a user calls the `match` function from section 3.2:

1. The `RegexOptions` and `MatchType` are transformed into a `Schedule` struct whose options are summarized in table 4.3.

2. The regex is parsed (section 4.1).

3. The parsed regex is used to generate a state transition table (section 4.2).

4. The schedule options, parsed regex, and state transition table are passed to a BuildIt function to generate specialized code (section 4.3).

5. The generated code is compiled and loaded and a pointer is returned to the matching function.

For the rest of our discussion, we consider all of the steps above as part of the regex compilation process. After they are completed, the compiled matching code is run to find a match. The following sections go over each one of these steps in detail. BREeze is implemented in C++ [1]. The generated code is in C.

[1]The implementation is available as a public GitHub repository as part of the BuildIt-lang organization at `https://github.com/BuildIt-lang/buildit_regex`.

4.1 Regex parser

The first step is parsing the user-provided regular expression into a meaningful representation to be consumed by the matching algorithm. The parser has multiple roles.

First, it simplifies some of the operators using other operators. Specifically, matching one or more characters like `x+` is transformed into `xx*`, and bounded repetition like `x{2,5}` is changed into `xx(x(x(x?)?)?`. Therefore, the matching algorithm does not require special handling of `+` or bounded repetition; it instead reuses the logic for simpler operators like `*` and `?` which simplifies the implementation.

Second, the parser extracts metadata about the regex which is used to simplify and optimize the matching process. It produces arrays that keep track of corresponding opening and closing brackets and the locations of | characters inside the regex. This information allows us to easily jump back and forth between states when generating the state transition table.

Third, in case of any invalid characters or malformed regular expressions, the parser prevents further compilation and matching and outputs the reason for termination.

4.2 State transition table

After parsing the regular expression, the next step is generating the state transition table. This is one of the most important steps in the compilation process because code generation is completely controlled by the state transitions. It is important to note that the states that we talk about here and for the rest of this paper are slightly different from the finite automata states that we saw in the introduction. We use the following notion of states. Each normal character in a regular expression has a corresponding state. In addition to that, each expression has an end state which gets activated only when a match is found. Therefore, a pattern of length `re_len` can have at most `re_len + 1` states. A transition from a state `s` to a state `t` means that

after matching the character corresponding to `s` we can match the character at `t`. There are two main steps involved in generating the state transition table.

First, we generate a one-dimensional array `next_states` of length `len(regex)` such that `next_states[s]` is the index of the next character we should check after `s`. This array is meant to simplify the second step of the generation described below. In most cases we have `next_states[s] = s+1`; however, in case `s` is followed by a closing bracket or parenthesis we set `next_states[s] = next_states[s+1]`. Since parentheses are used mostly for grouping and are not treated as actual states, in case of multiple nested groups this lets us skip over all the closing parentheses to get to an actual state. Table 4.1 gives an example of this array.

regex	a	(	b	c	)	*	d
`next_states`	1	2	3	5	5	6	7

Table 4.1: The `next_states` array for the regex `a(bc)*d`. Note that the next character index for `c` is 5 because we skip over `)`.

	a	**(**	**b**	**c**	**)**	*****	**d**	**$**
^	1	0	0	0	0	0	0	0
a	0	0	1	0	0	0	1	0
(	0	0	0	0	0	0	0	0
b	0	0	0	1	0	0	0	0
c	0	0	1	0	0	0	1	0
)	0	0	0	0	0	0	0	0
*****	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	1

Table 4.2: The state transition table for the regex `a(bc)*d`. `^` denotes the start of the regex and `$` the end. Each row marks the states that we can transition to from the current state. For example, having just matched `a` we can either match `b` or `d`. `(`, `)`, `*` are not valid states, so their rows do not have any active transitions.

The second step is generating the actual state transition table which is a two-dimensional array of size `(len(regex)+1) * (len(regex)+1)`. We call this table the `cache`. For any two states `s` and `t`, `cache[s][t] = 1` if both `s` and `t` are valid states and there is a transition from `s` to `t`. Otherwise, `cache[s][t] = 0`. An example of this table is given in table 4.2. A state `s` is valid if `regex[s]` is not

one of the following special characters: `]()|*+?^`. Figure A-1 shows partial code for generating the table for a simplified version of the language that contains only `*` as a special character. As shown in the code, the state transitions are generated for each state i.e. cache row separately.

4.3 Matching algorithm and code generation

Our implementation builds upon an already existing implementation of a small regex compiler with BuildIt [8] which only supports matching zero or more characters with *. We kept the core of the algorithm the same, but we extended the language to support more operators and schedules. Our main algorithm follows Thompson's NFA to DFA conversion; hence, it is close to a BFS approach. First, we will go over how one would implement a simple match function with this approach, and then we will talk about how to stage the code with BuildIt.

Figure 4-1 shows the pseudocode of our implementation without any staging. First, we allocate two state arrays of length `len(regex)+1`: `current` that keeps track of currently active states, and `next` that keeps states that are reachable from the current states. The algorithm consists of 2 nested loops. The outer loop iterates over each character of the input string, and the inner loop iterates over the `current` array. If a character from the input string matches with a character from the regex corresponding to an active state in `current`, we call the `progress` function which updates `next` with all the transitions from the current state using the state transition table from 4.2. In case of a partial match, after the completion of the inner for loop we activate the first state again in the `next` array to be able to match from the beginning of the regex again for the next position in the string. After that, `next` becomes `current`, and `current` is cleared for the next iteration of the outer for loop. Finally, if the end state is active in `current` we return the end index of the found match. In case of no match, we return -1. This simple approach summarizes the main logic behind our implementation.

Now we will talk about how to stage this approach with BuildIt. As mentioned

```
progress(regex, next, last_state)

update next with all reachable states from last_state
```

```
match(regex, string, partial)

current = next = [0, 0, …, 0]
progress(regex, current, -1) // activate the start state
for c = 0, …, len(string)-1:
    for state = 0, …, len(regex)-1:
        if current[state] == 1 and string[c] == regex[state]:
            progress(regex, next, state)
    if partial: // activate the start state again
        progress(regex, next, -1)
    current = next
    next = [0, 0, …, 0]
    if current[-1]: // we found a match
        return c
return -1 // no match
```

Figure 4-1: Pseudocode describing the basic matching algorithm.

before, BuildIt is type-based, meaning the stage that the code is evaluated in is determined by the type of variables that it uses. In our implementation, we use only 2 stages such that we declare the variables that we want to be evaluated in the first stage as static, and the ones that we want to appear in the second stage as dynamic. So starting with the simple implementation from figure 4-1, the only part left to do is decide for each variable whether it should be static or dynamic. When making this decision, the main goal we have in mind is to be able to generate specialized code for a given regular expression which can be compiled and run to find matches in different input strings. Therefore, we declare the regex as static and the string as dynamic. As a result, all the variables associated with the regex, such as `current`, `next`, `state`, etc. are also static, and the variables associated with the string such as `c` are dynamic. Similarly, the loop over the input string is dynamic, but the loop over the regex is static, and so on. As mentioned before, all the static code is evaluated in the first stage, which leaves the dynamic code to appear in the second stage. The entirety of the staged code is contained within a single dynamic function which we will refer to as `dyn_match`. Figure A-2 shows a simplified version of this function following the same logic as figure 4-1.

```
builder::builder_context context;
context.feature_unstructured = true;
auto fptr = (int (*)(const char*, int, int))builder::
    compile_function_with_context(context, dyn_match, "ab*c", ...);
int match = fptr("abcde", 5, 0);
```

Figure 4-2: Example of generating and compiling code with BuildIt.

The next step is to generate and compile the second-stage code (i.e. the specialized code) with BuildIt. This can be done by first creating a special BuildIt context object and passing it to the `compile_function_with_context` function (as shown in figure 4-2) which goes through 3 steps. First, it constructs the AST of the second-stage code by following the control flow paths of the first-stage code based on evaluating the static variables. Second, it generates code corresponding to the AST. The generated code mainly consists of if-else blocks comparing characters from the input string to characters from the regex, and labels and `goto` statements that simulate the outer for loop from the `dyn_match` function. Appendix B has some examples of generated code. Third, it compiles the generated code, loads it, and returns a pointer to the compiled function. Finally, to find a match, we just run this function on a given input string.

4.4 Code specialization for full and partial match

The code generation is controlled by the options of the `Schedule` struct which are passed to the `dyn_match` function described in the previous section. Depending on the values of these options, BuildIt can generate different variants of specialized code for the same regex to optimize for performance. The `Schedule` options are summarized in table 4.3. In this section, we are going to focus on three of these flags: `start_anchor`, `reverse`, and `last_eom` which help us distinguish between the full and partial match cases. We are going to discuss the rest of the options in section 5.

Table 4.4 shows how `MatchType` is transformed into different combinations of `Schedule` options. The only difference between a full and partial match implementation is one extra condition at the end of each iteration of the inner for loop that

Option	Description
`ignore_case`	when true, caseless matching is turned on (default: false); same as in `RegexOptions`
`dotall`	when true, `.` matches newline as well (default: false); same as in `RegexOptions`
`start_anchor`	when true, anchors the match at the beginning of the string (default: false)
`last_eom`	when true, `dyn_match` returns the last end of a match instead of the end of the first found match (default: false)
`reverse`	when true, the regex is reversed and `dyn_match` matches the string in reverse order (default: false)
`block_size`	when set to a positive integer the string is split into that many blocks which are matched in parallel (default: -1); same as in `RegexOptions` §5.5
`interleaving_parts`	matching stride for partial matches, distinct parts can be matched in parallel (default: -1); same as in `RegexOptions` §5.4
`split`	set to true if the `flags` string in `RegexOptions` contains an `s` character (default: false); it splits the regex such that there are multiple generated functions §5.1
`state_group`	set to true if the `flags` string in `RegexOptions` contains a `g` character (default: false); the grouped state transitions are kept in dynamic arrays instead of static §5.3

Table 4.3: Fields of the Schedule struct.

activates the first state of the regex in case `start_anchor` is false. This condition is given in line 30 in figure A-2. Figures B-1 and B-2 show the differences in the generated code for the same regex between full and partial matching.

As mentioned before, the `match` function from section 3.2 returns the first found match by default. We can make it return the first longest match by setting the `greedy` flag in `RegexOptions`. This triggers a two-pass matching process, where two functions get generated instead of one. The first pass is backward, which means we traverse the string from right to left and we follow the state transitions in the opposite direction from what is specified in the state transition table. That is, if we have just matched the state `t`, to update the `next` array, we look for all states `s` such that `cache[s][t] = 1` where `cache` is the state transition table. If `last_eom` is set (as is the case for the first longest match), the backward pass returns the leftmost

	full match	**any partial**	**first longest partial**	
	pass 1	pass 1	pass 1	pass 2
`start_anchor`	true	false	false	true
`reverse`	false	false	true	false
`last_eom`	true	false	true	true

Table 4.4: Specifies each type of match in terms of the relevant `Schedule` options. Finding a full match and a lazy partial match need only one pass. Finding the first longest partial match needs a two-pass approach.

match end position which is the same as the start index `si` of the first match. From that index, we run a forward pass with `last_eom` set to find the end position `ei` of the longest match anchored at `si`. Finally, the substring between `si` and `ei` in the input string is returned as the first longest match. We show examples of first- and second-pass code in figures B-4 and B-5.

If the `greedy` option is false we only need to run a single forward pass which is by far more efficient than the two-pass approach. This is the main reason why our implementation defaults to lazy matching.

Chapter 5

Scheduling options

Thompson's NFA simulation which is the basis of our code generation described above is a good general strategy for matching most types of regular expressions. As we were able to see in section 2, this approach is used in high-performance regex engines like RE2 and Hyperscan. However, it has been shown that this method does not perform very well for expressions with bounded repetition such as `(abc){50}` [15]. This is mainly because as the regex gets longer the size of the subset of currently active states during the NFA to DFA construction gets larger and widens the search. In our implementation, this means that BuildIt has to explore more code paths during the code generation stage and generate more complicated code which increases the compilation time. This inefficiency is evident in BREeze when trying to compile the following regex `(Tom.{10,15}river|river.{10,15}Tom)`. Apart from repetition, this expression also contains ambiguity due to the use of the dot character and alternation. As a result, the general code generation approach described above is not enough to compile this expression in any reasonable amount of time (it takes more than 5 minutes).

The scheduling options described in this section and summarized in table 4.3 are the result of our efforts to limit the number of currently active states at each search level and simplify the code generation process for this type of expressions. In general, in addition to improving compilation times, this also improves the running times due to the simpler matching code. More specifically, we leverage BuildIt's staging

```
string regex = "(ab|cd)(12)*";
string text = "23cd12123";
int res = 0;

RegexOptions split_options;
split_options.flags = ".s..s.......": // splitting on | groups
res = match(regex, text, split_options, MatchType::PARTIAL_SINGLE);

RegexOptions join_options;
join_options.flags = ".jj.jj..jj..": // joining literal strings
res = match(regex, text, join_options, MatchType::PARTIAL_SINGLE);

RegexOptions options;
options.interleaving_parts = 2;
res = match(regex, text, options, MatchType::PARTIAL_SINGLE);
```

Figure 5-1: Example code how to specify some of the scheduling options using the `match` function from section 3.2. All three approaches result in a match i.e. `res = 1` as expected.

capabilities to combine the BFS approach used in Thompson's algorithm with other approaches like backtracking. Table 5.1 shows the improvement in compilation times when using the scheduling options explained below.

Scheduling option	**Compile time**
none	$>$ 5 minutes
`split`	70 milliseconds
`join`	32 milliseconds
`interleaving`	8 seconds
`block`	42 milliseconds

Table 5.1: Times taken to compile `(Tom.{10,15}river|river.{10,15}Tom)` when using different scheduling options. In the absence of scheduling, we were not able to compile the regex in any reasonable amount of time.

The scheduling options can be specified using the `flags` field of the `RegexOptions` struct as shown in figure 5-1. The `flags` field is a string of the same length as the regex where each character is either `.` (default), `s`, `g` or `j`. The following sections go more into detail about each of these options.

5.1 Splitting the regex

This option combines the basic BFS approach with backtracking. More specifically, patterns with alternations can be compiled such that each alternation part is matched separately. This resembles backtracking because if one alternation option fails, we have to go back and try another one. To demonstrate how this works, consider the regex from above `(Tom.{10,15}river|river.{10,15}Tom)`. For this expression, the normal approach generates one function that performs BFS. Instead, we can generate 2 BFS functions `f1` and `f2` for matching `Tom.{10,15}river` and `river.{10,15}Tom` respectively. Finally, to get a match, we run `f1` first, and if it fails we run `f2`. To denote a split like this, we pass the following string as the `flags` option in `RegexOptions`: `.s..............s.....`. The `flags` string is the same length as the regex. To denote that we want to split the expression at index `i`, we set `flags[i] = 's'`. If there are no special options for index `j`, we just set `flags[j] = '.'`. When the user provided `RegexOptions` get parsed into a `Schedule` struct, the `split` option is set to true if there are any `s` characters in `flags`.

Although the `split` option was inspired by alternation expressions, it works in general for splitting at any position in the pattern even with no alternations.

Compiling `(Tom.{10,15}river|river.{10,15}Tom)` with this option takes only 70 milliseconds (compared to more than 5 minutes without any scheduling). The generated code has 3 functions: 1 function for each of the alternation options (we will call these functions helpers) and 1 main function that calls the helper functions. Only the main function implements the partial match logic. The helper functions implement an anchored match at a specific position in the string passed as an argument from the main function. We achieve this by keeping a working set of all the functions that need to be generated during the code generation stage. BuildIt first starts generating the main function. When the main function calls another dynamic function, we add that function to the working set and BuildIt generates it once it is done with the previous function. The number of active states at any time during the generation of each of these functions is generally lower than in the basic approach

because the work is distributed across multiple functions. This results in generating more compact code which improves both the compilation and running times.

5.2 Matching multiple characters at once

In the normal approach, literal strings that appear as part of a regex, such as `Tom` are matched one character at a time. As mentioned before, this results in a lot of states being active at the same time which slows down the code generation process. One reason for this is that each character results in a separate if condition in the generated code. Combined with different paths the code can take based on the outcome of the if statements, the matching code can become very long and complicated.

To decrease the total number of active states and simplify the code generation we introduce the `join` option to compare multiple consecutive characters at once with `memcmp`. For example, with this option, the string `Tom` is represented with only one state instead of 3. Similarly, the generated code has only one if condition instead of 3. This option not only simplifies the compilation process, but also has some benefits in terms of running times. Namely, when compiling the generated code, the backend C compiler can represent `memcmp` as a single instruction which adds an extra optimization of being able to compare multiple bytes with a single instruction call. In the normal approach, one instruction would compare only a single byte.

The `join` option can be specified as part of the `flags` string in `RegexOptions` just like the `split` option. To mark that we want to match multiple characters as a single string, we put the character `j` in `flags` at the positions of those characters. For example, if we want `Tom` and `river` from `(Tom.{10,15}river|river.{10,15}Tom)` to be matched with this option, we set `flags` to `.jjj.......jjjjj.jjjjj.......jjj..` Since we are using `memcmp`, the `join` option can only be used for normal characters (characters that are not escaped) and with the `ignore_case` flag set to false.

Compiling `(Tom.{10,15}river|river.{10,15}Tom)` as described above takes only 32 milliseconds. The generated code consists of 5 functions: 1 main function and 4 helper functions corresponding to each one of the 4 literal strings appearing in the

regex. This is because apart from `memcmp` the `join` option utilizes a similar splitting technique as in section 5.1. It works as follows. Just like in section 5.1, we keep a working set of functions that need to be generated. BuildIt starts generating the main function. When the main function calls `memcmp` on `Tom` if the call results in a match, it calls a new dynamic function that starts matching right after `Tom` i.e. `.{10,15}river`. That dynamic function is pushed to the working set to be generated later. The same procedure happens for all the literal strings that are marked with `j`, hence we have 4 helper functions. Note that this approach is also similar to backtracking. An example of using this option is given in figure B-3.

5.3 Dynamic grouping

As described in section 4, the active states are kept in two static arrays `current` and `next`. Because both of these arrays are static, all the state transition work happens in the first stage of code generation. We can spread this work out over two stages by introducing the dynamic grouping option. The user can mark multiple consecutive states in the `flags` string from `RegexOptions` with the character `g`. For example, the regex `Tom.{10,15}river` can be compiled as `ggggggggggg.....`. The state transition for all the states marked with `g` is handled dynamically in the second stage, which means it appears in the generated code. If there are any `g` characters present in `flags`, the `state_group` flag is set to true when translating `RegexOptions` into the `Schedule` struct.

This option is implemented by keeping two extra dynamic arrays `dyn_current` and `dyn_next` in addition to the static arrays. We keep track of the grouped states (the states marked with `g`) in the dynamic arrays and of the rest of the states in the static arrays as usual. One would expect that this would improve the compilation times because some of the first-stage work is shifted to the second stage. However, it does not help with compiling complex patterns like `(Tom.{10,15}river|river.{10,15}Tom)`. This is mainly because introducing extra dynamic variables in the first stage code results in generating longer and more complicated code which naturally takes a longer

time. Due to the bad performance, we do not further explore this option and do not include it in the results section.

5.4 Interleaving

As mentioned in section 4.4 the partial match code is generated by activating the first state of the regex for each position in the dynamic string that we are matching. This is the main reason behind having many states being active at the same time. To control the frequency at which we activate new states to start matching from the beginning of a pattern, we introduce the `interleaving` option. The user can specify the interleaving frequency with the `interleaving_parts` option in `RegexOptions` which later gets copied to the `Schedule` struct.

Let n be the number of interleaving parts. Interleaving works as follows. There are n different functions generated, such that the i^{th} function is responsible for matching the pattern starting at every position p in the dynamic string where $p \bmod n = i$. These functions are independent of each other because they operate from different starting positions in the dynamic string. Therefore, we can run them in parallel. We parallelize the code by adding `#pragma omp parallel for` to the for loop that calls each of the n functions.

This option lets us compile `(Tom.{10,15}river|river.{10,15}Tom)` with 16 interleaving parts in around 8 seconds. The compilation time can be improved by varying the number of interleaving parts or combining these options with the `split` and `join` options. Table 6.6 shows how changing the number of interleaving parts affects the running times.

5.5 Splitting the string

Although `interleaving` helps with the compilation times, it does not in general improve the running times as we will see in section 6.2.2. The main reason for this is that each interleaving function still has to traverse the entire string to find a match.

To avoid iterating over the entire dynamic string serially, we split it into blocks that can be traversed in parallel. The user can specify the block size with the `block_size` option in `RegexOptions` which gets copied into the `Schedule` struct. For the rest of the paper, we refer to this scheduling option as the `block` option.

The generated code with this option consists only of one function, assuming we are not using any of the other scheduling options from above. Among other arguments, this function takes in an index s in the dynamic string that we need to start matching from. Let L be the length of the dynamic string and B the `block_size` specified by the user. Then, the number of blocks is $N = \lceil \frac{L}{B} \rceil$. We simulate splitting the string into N blocks by calling the generated function N times such that in the i^{th} call we pass $s = B * i$ for the string start position. Internally, the generated function continues looking for partial matches only until their starting positions correspond to a position in the dynamic string p such that $s \leq p < s + B$. The N function calls can be run in parallel.

The `block` option can be used in combination with the other scheduling options described above. Generally, it has better compilation times compared to `interleaving`. We can compile `(Tom.{10,15}river|river.{10,15}Tom)` with the `join` and `block` options together in around 42 milliseconds. Moreover, `block` significantly improves the running times, as we will see in section 6.2.2.

Chapter 6

Evaluation

One of the main goals of this project is to keep the implementation simple while achieving comparable performance to the existing regex libraries. In the following sections, we evaluate both of these aspects of our implementation.

6.1 Implementation complexity

Implementing BREeze on top of BuildIt allowed us to generate highly specialized code with very simple and concise implementation. To demonstrate this, we compare the number of lines of code in our codebase to the total lines of code in Hyperscan [1], RE2 [4], and PCRE2 [2]. For fairness, we only counted the lines of code inside the `src` directory of the official GitHub repositories of each of these libraries, excluding any testing or timing code. Table 6.1 summarizes our findings.

Library	**LOC**
BREeze	1,564
BREeze + BuildIt	9,290
RE2	26,587
Hyperscan	187,033
PCRE2	131,995

Table 6.1: Number of lines of source code used for implementation of each of the libraries. The count excludes the code used for testing and benchmarking.

Even if we consider the lines of code in the BuildIt implementation, our imple-

mentation does not get longer than 10000 lines of code, which is significantly less than the number of lines in the libraries shown in table 6.1. This is mainly because it is very easy to add new features to BREeze with very minimal changes due to the specialized code generation with BuildIt.

6.2 Performance

In this section, we analyze the compilation and running times for finding a single partial match in a long string. We ran the experiments on an Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz machine with 128GB memory, 48 cores, and 2 threads per core. We used the `teakettle_2500` and `snort_literals` regular expressions sets with the `gutenberg` and `alexa200` texts respectively from a Hyperscan performance analysis blog [16]. Additionally, we used the Twain benchmark [5] to show our performance on a small set of regular expressions compiled with hand-optimized schedules.

6.2.1 Benchmarks

`Teakettle`

The `teakettle_2500` pattern set consists of 2500 synthetically generated regular expressions. Each expression contains literal strings separated by character class repetitions [16]. Some of these patterns are compiled with the `ignore_case` and `dotall` flags set. We show a few of these expressions in table 6.2. For our experiments we picked 50 patterns at random and matched them against the `gutenberg` text from [16]. `gutenberg` is a 6.7M character long string - a collection of texts in English taken from Project Gutenberg [3].

`^backfields.*lipstick.*curers`
`outstriding.{9,11}dislike.{6,6}pout.{1,8}sterigmata`
`leaderless[^\r\n]+doubtlessnesses[^\r\n]+lummox`

Table 6.2: Example regular expressions from `teakettle_2500`.

`Snort`

The `snort_literals` dataset consists of around 3000 regular expressions extracted from a ruleset provided by the Snort 3 network intrusion detection system [16]. Some of these patterns are using the `ignore_case` flag. In contrast to `teakettle`, the patterns in this dataset contain a large number of hexadecimal and escaped characters. Some examples are shown in table 6.3. For our experiments, we picked 20 expressions from this dataset and matched them against the `alexa200` text [16]. This text is around 142.3M characters long and represents a sample of a PCAP capture of browsing a list of websites.

B09DE715-87C1-11D1-8BE3-0000F8754DA1
f\xB9\xFF\xFF\xEB\x19\^\x8B\xFE\x83\xC7
PARENTNODE\.REMOVECHILD\(DOCUMENT\.GETELEMENTSBYTAGNAME\('SELECT'\)\)

Table 6.3: Example regular expressions from `snort_literals`.

`Twain`

`Twain` consists of a small set of regular expressions with varying complexity shown in table 6.8. It is a widely used dataset for benchmarking regex engines [13]. The text used for matching is a collection of Mark Twain's works [5] and it is around 16M characters long.

6.2.2 General schedules

In this section, we are measuring our performance against 50 patterns from `teakettle` and 20 patterns from `snort`. Due to the size of these datasets, we were not able to manually find the optimal schedule for each expression. Instead, we picked one general schedule for all of the patterns as discussed below.

Serial matching

Table 6.4 shows the running times when matching each of the expressions when compiled with a serial schedule. A serial schedule is any schedule that does not

include the `interleaving` or `block` options. The times in the two rows represent the total times required to match the 50 `teakettle` and 20 `snort` patterns respectively. For the BREeze results, the patterns without escaped characters and `ignore_case` enabled were compiled with the `join` schedule option by grouping all the characters of the literal strings. For example, `^backfields.*lipstick.*curers` was compiled as `.jjjjjjjjjj..jjjjjjj..jjjjjj`. We compiled the rest of the patterns with the `split` option by splitting on every 8th character whenever possible. For example, `ameliorator.*syncretistic` was compiled as `.......s.......s.......s..`

For BREeze we used the `PARTIAL_SINGLE` flag which stops matching as soon as any match is found. For RE2 we used the `PartialMatch` function. For PCRE2 we used `pcre2_match`. Since RE2 and PCRE2 match greedily by default, we converted the greedy operators into lazy ones to compare against our default lazy matching. We ran Hyperscan in block mode with `HS_FLAG_SINGLEMATCH` enabled to stop scanning as soon as a match is found. For a fair comparison, we did not use Hyperscan's multi-pattern matching mode since BREeze, PCRE2, and RE2 do not support it.

	`teakettle`	`snort`
BREeze	211.64	1798.37
RE2	181.81	1100.32
Hyperscan	34.10	249.06
PCRE2	1066.23	2315.46

Table 6.4: Serial running times for finding a single partial match. The times are given in milliseconds. Each time is a total of matching all the 50 (`teakettle`) and 20 (`snort`) expressions.

From table 6.4 we observe that BREeze performs better than PCRE2 and worse than Hyperscan and RE2. We believe that the bad serial performance is mainly because our code is not very well optimized for the case when the pattern does not exist in the text, which is the case with most of the selected patterns in this experiment. While RE2 and Hyperscan have special mechanisms for looking for the start of the literal strings in the text ahead of matching, we do not have anything similar. Instead, our code tries matching starting from each character in the text which is very expensive if we are searching in a very long text such as `alexa200`.

From the results, it is also notable that BREeze performs better for `teakettle` than for `snort`. From our experience, this is due to two main reasons. First, the `alexa200` text used with `snort` is much longer than the `gutenberg` text so in case there is no match, `snort` will naturally take more time. Second, the `snort` patterns are more complicated because they contain special characters that need to be escaped which prevents us from using the `join` option.

	`teakettle`	`snort`
BREeze	9461.64	3803.61
RE2	2.07	0.90
Hyperscan	57.47	1.60
PCRE2	0.50	0.19

Table 6.5: Compilation times for the serial experiments from table 6.4. The times are given in milliseconds.

Table 6.5 summarizes the compilation times for the same experiment. It is expected that our compilation process takes more time than the existing libraries because it includes the code generation stage with BuildIt. Although there is some room for improvement, we are not very concerned about the compilation times at the moment. This is because in most cases we would expect the user to compile the pattern once and reuse the generated code to match against different strings. Hence, for most of our work, we prioritized improving the running times.

Parallel matching

To improve the running times from table 6.4 we tried using the `interleaving` and `block` scheduling options to parallelize the matching. We still keep the `split` and `join` options on in this part.

The results for the interleaving schedule are given in table 6.6. The number of interleaving parts is equal to the number of threads that we run in parallel. According to the results, increasing the number of interleaving parts increases the matching times. Although this might seem surprising, it is expected. With interleaving enabled we only increase the stride at which we start matching from the beginning of the pattern in the string. However, each thread still has to loop through every character

interleaving parts	`teakettle`	`snort`
2	436	2448.79
4	561.11	3753.61
8	846.88	6347.58
16	1247.53	11050.6

Table 6.6: The running times when we search for a partial match in parallel using the interleaving schedule option. The times are in milliseconds.

in the string to check if the pattern matches. Moreover, when the pattern results in a match in one of the threads, we currently do not have a way to stop the other threads from running. Instead, we have to wait for them to complete before returning the match.

When there is no match, in the serial case we have an optimization to stop matching as soon as there are no more active states. This optimization cannot be used with interleaving because even if there are no more active states at the moment, we might insert a new active state in the future marking the beginning of the pattern which may result in a match. This is another reason behind the increased running times.

number of blocks	`teakettle`	`snort`
10	75.57	329.1
20	40.72	176.663
30	33.91	184.839
40	29.9	141.05
50	33.97	176.33

Table 6.7: The running times when we search for a partial match in parallel using the `block` option. The times are in milliseconds.

To avoid looping serially through the entire string we tried processing contiguous blocks of the string in parallel using the `block` schedule option. The results from this approach are given in table 6.7. The number of blocks is equal to the number of threads we run in parallel. The running times decrease as we increase the number of blocks to 40 and then they stay around the same for 50 blocks or more. This is expected due to the number of cores on our machine. Overall, this schedule improved the `teakettle` running times from 211.64ms to 29.9ms and the `snort` ones from 1798.37ms to 176.33ms. The new times are better than the serial times for Hyperscan

from table 6.4. This does not mean that our library is more optimized than Hyperscan since we have not utilized Hyperscan's parallel functions. However, it shows that our scheduling options are enough to generate code that runs in a reasonable amount of time compared to state-of-the-art libraries.

6.2.3 Tuned schedules

To generate the results for this section we tried different schedules for each regular expression and picked the one which resulted in the best performance. After compiling each expression with its corresponding schedule, we ran the generated code to find the first partial match. Table 6.8 shows the running times. The schedules that were used for getting these results mostly involved the `split` and `join` options from section 5. We tried parallelizing the code with the `interleaving` and `block` options, but we did not get a big performance improvement. This is because most of the `Twain` patterns result in a match very early in the text at which point the matching stops, so processing the text in parallel does not add any advantages.

regular expression	BREeze	RE2	HScan	PCRE2
`Twain`	**0.0001**	0.0006	0.0020	0.0005
`(?i)Twain`	**0.0001**	0.0004	0.0020	0.0004
`[a-z]shing`	**0.0022**	0.0051	0.0025	0.0529
`(Huck[a-zA-Z]+|Saw[a-zA-Z]+)`	1.539	3.999	**0.623**	2.052
`[a-q][^u-z]{13}x`	0.131	0.066	**0.011**	0.110
`(Tom|Sawyer|Huckleberry|Finn)`	0.027	0.035	**0.003**	0.160
`(?i)(Tom|Sawyer|Huckleberry|Finn)`	0.015	0.012	**0.002**	0.160
`.{0,2}(Tom|Sawyer|Huckleberry|Finn)`	0.041	0.045	**0.003**	3.734
`.{2,4}(Tom|Sawyer|Huckleberry|Finn)`	0.041	0.035	**0.003**	3.594
`(Tom.{10,25}river|river.{10,25}Tom)`	5.195	13.349	**0.815**	26.962
`[a-zA-Z]+ing`	**0.0015**	0.0029	0.0025	0.0536
`\s[a-zA-Z]{0,12}ing\s`	0.011	0.0048	**0.0047**	0.0362
`([A-Za-z]awyer|[A-Za-z]inn)\s`	5.796	2.964	**0.336**	87.339
`["'][^"']{0,30}[?!\.]["']`	**0.027**	0.030	0.034	0.058

Table 6.8: Running times in milliseconds for the Twain dataset.

From table 6.8 we observe that BREeze is consistently faster than PCRE2, and faster than RE2 for most of the expressions. Although we are doing better for some

expressions, Hyperscan in general performs the best for this benchmark. These results show the importance of tuning the schedules to their corresponding regular expressions for getting the best performance. We can see that BREeze generally performs better in this case than when using untuned schedules as in section 6.2.2. This experiment also shows that our scheduling options are enough to generate code with similar and in most cases better performance compared to the existing libraries.

Chapter 7

Conclusion

7.1 Summary

In this paper we introduced BREeze, a regular expression library implemented on top of BuildIt that generates specialized regular expression matching code. We demonstrated that BREeze supports a reasonable set of characters and operators that make it possible to represent a wide range of regular expressions. Moreover, it has a fully functional API that supports a variety of matching modes present in modern regular expression libraries. BREeze is distinguishable from the existing libraries because it provides the user with a unique set of scheduling options that can be used to further specialize and optimize the generated code. Next, we showed that combining these scheduling options is enough to generate code with comparable and in some cases even better performance than the state-of-the-art regex engines. Finally, we made all of the above possible with only 1564 lines of code.

7.2 Future work

From the results in section 6 it is evident that BREeze performs better for regular expressions compiled with tuned schedules. However, currently, we do not have a good approach to finding the most optimal schedule. More specifically, the hand-tuning approach used to generate the schedules for the Twain results in table 6.8

looks as follows. First, we generate a subset of schedules that we expect to have good performance (for example, using the `join` option on short literal string components). We compile the regex separately with each one of the schedules. We measure the running time for each of the compiled match functions on the text. Finally, we pick the schedule which results in the best running time. This approach is very inefficient, especially because it requires generating code for every possible schedule which takes a long time.

Tuning the Twain regular expressions from table 6.8 with the above approach exposed some common patterns shared among the most optimal schedules. First, when a regular expression contains short literal string components the best schedule almost always involves grouping the literal string characters with the `join` option. For example, the best performing schedule for `(Tom.{10,25}river|river.{10,25}Tom)` is `.jjj.......jjjjj.jjjjj.......jjj.`. Second, for expressions with repeated character classes, the schedule usually involves the `split` option for some of the classes. For example, `s...s.........s.....s...` is the best schedule for the regex `["'][^"']{0,30}[?!\.]["']`. Although these patterns are not instantly obvious from a user's perspective, they should be very easily detectable by a machine learning model. Therefore, the main focus of our future work is to automate the schedule tuning process by introducing a machine learning model to predict the most optimal schedule for a given regular expression. This involves two main tasks.

The first task is to generate a training dataset mapping regular expressions to their optimal schedules. The `teakettle` and `snort` datasets from [16] contain around 5500 regular expressions in total which can be used for training. One could generate the most optimal schedules for the training data using the tuning approach described at the beginning of this section. Generating the training dataset will likely take a long time; however, it needs to be done only once.

The second task is to implement and train a machine learning model that takes a regular expression string as input and outputs a string of the same length as the regular expression representing the `flags` of the most optimal schedule. Following the previous examples, a trained model that takes in `(Tom.{10,25}river|river.{10,25}Tom)`

as input should output `.jjj.....jjjjj.jjjjj.....jjj..`

Using a machine learning model greatly simplifies the tuning process. Once the model is trained, it can be used to pick one specific schedule per regular expression which eliminates the need to repeat the compilation multiple times to try different schedules. As a result, this will allow users to easily get the best performance out of BREeze.

Appendix A

Implementation details

```
void fill_cache_row(char* re, int cs, int* next_states, int* row) {
    int ns = (cs == -1) ? 0 : next_states[cs];
    if (strlen(re) == ns) {
        row[ns] = 1;
    } else if (is_normal(re[ns]) || re[ns] == '.') {
        if (re[ns+1] == '*') {
            // skip the current state...
            progress(re, ns + 1, next_states, row);
        }
        // ... or keep it
        row[ns] = 1;
    } else if (re[ns] == '*') {
        row[cs] = 1; // repeat the last state
        progress(re, ns, next_states, row); // or skip
    } ... // other conditions
}

void fill_cache(string re, int* next_states, int** cache) {
    // generate the state transition table row by row
    for (int s = -1; s < strlen(re); s++) {
        fill_cache_row(re, s, next_states, cache[s+1]);
    }
}
```

Figure A-1: Parts of the code used for state transition generation.

```
dyn_var<int> dyn_match(const char* re, dyn_var<char*> str,
        dyn_var<int> str_len, Schedule options, int* cache,
        bool partial_match, dyn_var<int> to_match, ...) {

    // allocate two state vectors
    static_var<char[]> current, next;

    dyn_var<int> no_match = to_match - 1; // no_match is usually -1
    dyn_var<int> last_end = no_match; // last end of match

    // activate the initial states
    update_states(current, cache, -1, ...);

    // iterate over str
    while (to_match < str_len) {
        // check each state
        for (static_var<int> state = 0; state < re_len; state++) {
            if (current[state])
                if (is_normal(re[state])) {
                    if (str[to_match] == re[state])
                        update_states(next, cache, state, ...);
                } else if ('.' == re[state]) {
                    update_states(next, cache, state, ...);
                } ... // other cases
                } else {
                    // invalid character
                    return no_match;
                }
            }
        if (partial_match) {
            // start matching again from the first state
            update_states(next, cache, -1, ...);
        }
        // swap current and next, clear next
        ...
        to_match = to_match + 1;

        // check if there is a match so far
        if (current[re_len]) {
            last_end = to_match;
            // if lazy match return last_end
            ...
        }
        ...
    }
    return last_end;
}
```

Figure A-2: Parts of the code for the `dyn_match` function.

Appendix B

Generated Code Examples

```
int match_0 (char* arg4, int arg5, int arg6);
int match_0 (char* arg4, int arg5, int arg6) {
  int var15;
  int var0 = arg6;
  int var8 = (var0 * 1) - 1;
  int var9;
  char var10;
  if ((var0 >= 0) && (var0 < arg5)) {
    var10 = arg4[var0];
    var9 = var10 == 97;
    if (var9) {
    } else {
      goto label1;
    }
    var0 = var0 + 1;
    if (var0 > var8) {
    } else {
      goto label0;
    }
    var8 = var0;
    label0:
    if ((var0 >= 0) && (var0 < arg5)) {
      var10 = arg4[var0];
      var9 = var10 == 98;
      if (var9) {
        var0 = var0 + 1;
        if (var0 > var8) {
          var8 = var0;
        }
        goto label0;
      }
      label1:
      var0 = var0 + 1;
      var15 = var8;
      return var15;
    } else {
      var15 = var8;
      return var15;
    }
  } else {
    var15 = var8;
    return var15;
  }
}
```

Figure B-1: Full match code for `ab*`.

```
int match_0 (char* arg4, int arg5, int arg6);
int match_0 (char* arg4, int arg5, int arg6) {
  int var13;
  int var0 = arg6;
  int var8 = (var0 * 1) - 1;
  int var9;
  char var10;
  label0:
  if ((var0 >= 0) && (var0 < arg5)) {
    var10 = arg4[var0];
    var9 = var10 == 97;
    if (var9) {
    } else {
      var0 = var0 + 1;
      goto label0;
    }
    var0 = var0 + 1;
    if (var0 > var8) {
    } else {
      goto label1;
    }
    var8 = var0;
    label1:
    var8 = var0;
    var13 = var8;
    return var13;
  } else {
    var13 = var8;
    return var13;
  }
}
```

Figure B-2: Partial match code for `ab*`.

```c
#include <string.h>

int match_0 (char* arg4, int arg5, int arg6);
int match_0 (char* arg4, int arg5, int arg6) {
  int var0 = arg6;
  int var8 = (var0 * 1) - 1;
  int var9;
  label0:
  if ((var0 >= 0) && (var0 < arg5)) {
    var9 = memcmp((arg4 + var0) - 0, "abcd", 4);
    if (var9 == 0) {
    } else {
      goto label1;
    }
    int var11;
    var11 = var0 + 4;
    if (var11 > ((var0 + 4) - 1)) {
    } else {
      label1:
      var0 = var0 + 1;
      goto label0;
    }
    return var11;
  } else {
    return var8;
  }
}
```

Figure B-3: Partial match code for `abcd` using the `join` schedule as `jjjj`.

```
int match_4 (char* arg4, int arg5, int arg6);
int match_4 (char* arg4, int arg5, int arg6) {
  int var17;
  int var0 = arg6;
  int var8 = (var0 * 1) + 1;
  int var9;
  char var10;
  label0:
  if ((var0 >= 0) && (var0 < arg5)) {
    var10 = arg4[var0];
    var9 = var10 == 98;
    if (var9) {
    } else {
      goto label3;
    }
    label1:
    var0 = var0 + -1;
    if (var0 < var8) {
    } else {
      goto label2;
    }
    var8 = var0;
    label2:
    if ((var0 >= 0) && (var0 < arg5)) {
      var10 = arg4[var0];
      var9 = var10 == 97;
      if (var9) {
        var0 = var0 + -1;
        if (var0 < var8) {
          var8 = var0;
        }
        goto label2;
      }
      var10 = arg4[var0];
      var9 = var10 == 98;
      if (var9) {
        goto label1;
      }
      label3:
      var0 = var0 + -1;
      goto label0;
    }
    var17 = var8;
    return var17;
  } else {
    var17 = var8;
    return var17;
  }
}
```

Figure B-4: First pass code for finding the first longest match for `a*b`.

```
int match_0 (char* arg4, int arg5, int arg6);
int match_0 (char* arg4, int arg5, int arg6) {
  int var15;
  int var0 = arg6;
  int var8 = (var0 * 1) - 1;
  int var9;
  char var10;
  label0:
  if ((var0 >= 0) && (var0 < arg5)) {
    var10 = arg4[var0];
    var9 = var10 == 97;
    if (var9) {
      var0 = var0 + 1;
      goto label0;
    }
    var10 = arg4[var0];
    var9 = var10 == 98;
    if (var9) {
    } else {
      goto label2;
    }
    var0 = var0 + 1;
    if (var0 > var8) {
    } else {
      goto label1;
    }
    var8 = var0;
    label1:
    if ((var0 >= 0) && (var0 < arg5)) {
      label2:
      var0 = var0 + 1;
      var15 = var8;
      return var15;
    } else {
      var15 = var8;
      return var15;
    }
  } else {
    var15 = var8;
    return var15;
  }
}
```

Figure B-5: Second pass code for finding the first longest match for `a*b`.

Bibliography

[1] Hyperscan code. https://github.com/intel/hyperscan.

[2] PCRE2 code. https://github.com/PCRE2Project/pcre2.

[3] Project Gutenberg. https://www.gutenberg.org/.

[4] RE2 code. https://github.com/google/re2.

[5] Project Gutenberg complete works of Mark Twain. 2021. https://www.gutenberg.org/files/3200/.

[6] Michela Becchi and Patrick Crowley. A-DFA: A time- and space-efficient DFA compression algorithm for fast regular expression evaluation. *ACM*, 2013.

[7] Ajay Brahmakshatriya and Saman Amarasinghe. BuildIt: A type based multi-stage programming framework for code generation in C++. *CGO*, 2021.

[8] Ajay Brahmakshatriya and Saman Amarasinghe. GraphIt to CUDA compiler in 2021 LOC: A case for high-performance DSL implementation via staging with BuilDSL. *CGO*, 2022.

[9] Russ Cox. Regular expression matching in the wild. 2010. https://swtch.com/~rsc/regexp/regexp3.html.

[10] James Davis. Rethinking regex engines to address ReDoS. *ACM*, 2019.

[11] Jan Goyvaerts. The PCRE open source regex library. 2021. https://www.regular-expressions.info/pcre.html.

[12] Philip Hazel. pcre2matching man page. 2021. https://www.pcre.org/current/doc/html/pcre2matching.html.

[13] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. Symbolic regex matcher. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–378, Cham, 2019. Springer International Publishing.

[14] Ken Thompson. Regular expression search algorithm. In *Communications of the ACM*, 1968.

[15] Lenka Turonová, Lukáš Holík, Ivan Homoliak, Ondrej Lengál, Margus Veanes, and Tomáš Vojnar. Counting in regexes considered harmful: Exposing ReDoS vulnerability of nonbacktracking matchers. *USENIX*, 2022.

[16] Justin Viiret. Hyperscan: Performance analysis of Hyperscan with hsbench. `https://www.intel.com/content/www/us/en/collections/libraries/hyperscan/performance-analysis-hyperscan-hsbench.html`.

[17] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[18] Zhi Liu Zhe Fu and Jun Li. Efficient parallelization of regular expression matching for deep inspection. *IEEE*, 2017.