University of Manchester
School of Computer Science
Project Report 2022

Profiling-Guided String Representation Optimisations in TruffleRuby

Author: Veselin Karaganev

Supervisor: Mikel Luján

Abstract

Profiling-Guided String Representation
Optimisations in TruffleRuby

Author: Veselin Karaganev

The string data type is ubiquitous in all general-purpose programming languages. The purpose of this work is to understand in detail the implementation of the string data type in TruffleRuby, to compare and contrast that implementation with other common string implementations, to identify its strengths and weaknesses, to devise and implement optimisations to address those weaknesses, and to evaluate and compare those optimisations against the baseline TruffleRuby runtime and to other Ruby runtimes.

This work presents several optimisation for the tree-based representation of strings used in TruffleRuby. A new profiling technique which can be used to detect expensive string operations is developed. An improved implementation based on this technique is presented. By using the adaptive optimisation capabilities of TruffleRuby, the new implementation is able to switch from a tree-based to an array-based string representation at runtime in the cases where it is deemed more optimal. Benchmarks and analysis of the new implementation are presented at the end of this work.

This work addresses two outstanding performance issues on the TruffleRuby issue tracker by improving the performance in the use cases described in those issues by close to three orders of magnitude.

Supervisor: Mikel Luján

Acknowledgements

I would like to thank my parents for their help and support, my supervisor Mikel Luján for his guidance and feedback, Chris Seaton for his coverage of TruffleRuby in his blog, which was an invaluable resource, Benoit Daloze, for expressing interest and providing context for this work, as well as Andrew Nesbit and Salim Salim, who offered their help and advice whenever it was needed. Without your help none of this would have been possible.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Strings are a crucial data type and are available in almost all programming languages. Strings are used in server applications to generate web pages, to process user input, for natural language processing tasks, and many other tasks.

A user report on the TruffleRuby project's public issue tracker titled 'Raytracer benchmark is 34x slower than with MRI' [29] revealed that the implementation of the string data type significantly underperforms when compared to other Ruby implementations in some scenarios. As TruffleRuby uses a Just-in-Time dynamic recompiler that allows adaptive optimisations not easily achievable in other Ruby implementations, it should be possible to detect those use-cases at runtime and change the parameters of the existing implementation or develop new specialised solutions.

1.2 Objectives

This project has five objectives.

1. Generate a survey of the string representations in popular language implementations.
2. Identify a set of benchmarks to compare the performance of the string implementations in Ruby runtimes.
3. Develop an understanding of dynamic language runtimes, general optmisation techniques and optimisation techniques available in TruffleRuby.
4. Devise and implement an improved string implementation using these techniques which addresses the aforementioned performance issues reported by users.
5. Perform a thorough performance evaluation and report any findings.

1.3 Report structure

This report consists of 5 chapters, each of which covers a distinct portion of work on this project:

1. Introduction gives an overview of the project.
2. Background provides the background information necessary to understand the accomplishments in this work.
3. Implementation covers the separate units of the implementation work.
4. Testing and evaluation quantifies the work done in the implementation.
5. Conclusions reflects on the decisions made during the development. of this work, summarises the work and suggests areas for improvement.

Chapter 2

Background

This chapter provides the background information necessary to understand the accomplishments in this work. The main focus is on the common way of implementing the string data type, as well as on the TruffleRuby implementation of Ruby and the tools for runtime optimisations it provides.

2.1 Strings

2.1.1 Considerations

Strings are a critical data type in any programming language because they are so widely used. Strings can be used in server applications to generate HTML markup, to process/sanitise user input, to refer to other entities in the program (often called dynamic introspection/reflection), for natural language processing tasks, and many other tasks.

The choice of data structure for strings is often one that is made by the language implementor (it is seldom determined by an international standard) and one that must be appropriate for all these different scenarios. One of the most limiting factors when it comes to choosing which data structure to use is whether the source language treats strings as mutable or immutable data types.

Mutable strings are objects which allow their value to be changed. Immutable strings are fixed after creation. The example in listing 2.1 (in pseudo-code) will not be permitted in a language with immutable strings.

```
1 x = "programming"
2 x[0] = "P"
```

Listing 2.1: Replacing a character by its index in a mutable string

2.1.2 Mutability affects performance

In languages with immutable strings, alternative data types that support mutability are often provided (Java has the `StringBuilder` and `StringBuffer` data types). Even when both types of strings are available for use, programmers need to be aware of whether strings in their language of choice are mutable or immutable, because this will allow them to write more efficient code [10].

Let's look at two alternative implementations of a Java program which computes the string `"foofoofoo..."` consisting of N copies of `"foo"`.

```
String result = "";
for (int i = 0; i < N; i++) {
  result = result + "foo";
}
```

Listing 2.2: Incremental string building using immutable strings

In the example in listing 2.2, the immutable string type `String` is used. The semantics of the immutable string type require that a new String instance is created for each intermediate concatenation operation, represented as $+$.

To compute the concatenation of two strings $left$ and $right$, a new string is allocated, with a character array of size $len(left) + len(right)$. All $len(left)$ characters are copied from the array of $left$ to the newly allocated string's array, followed by all $len(right)$ characters from the array of $right$.

It is easy to see that for N loop iterations, N such temporary strings are allocated. Because at most N characters per string are copied, the running time of this algorithm will have a worst-case bound of $O(N^2)$.

```
StringBuilder temp = new StringBuilder();
for (int i = 0; i < N; i++) {
  temp.append("foo");
}
String result = temp.toString();
```

Listing 2.3: Incremental string building using mutable strings

In the example in listing 2.3, the mutable string type `StringBuilder` is used. The semantics of this type allow for a growable character array to be used [10]. No new objects are allocated in the body of the loop.

To append a `String` $right$ to a `StringBuilder` $left$, the internal character array of left needs to be large enough to hold at least $len(left) + len(right)$ characters. All $len(right)$ characters are copied from the array of $right$ to the existing internal array of $left$. If the internal character array is not large enough, a new character array of size $c * len(left)$ is allocated. All n characters are copied from the old array to the new array, leaving spare capacity of $(c-1) * n$. It is important to point out that with a well-selected constant c, the process of growing the internal array of $left$ can be avoided most of the time. In fact, in this situation, amortized constant complexity is observed [6]. Using amortized compexity analysis, it can be concluded that the algorithm will run in $O(N)$ time.

2.1.3 Encoding

An encoding is a specification of how individual characters that make up a string are mapped to a binary representation. Different encodings might represent the same set of characters. In addition, variable-length encodings exist, which can map different characters to a different number of bytes.

The choice of encoding has an important implication on the running time of some algorithms. For example, accessing the N-th character of a string in a fixed-length encoding only requires a single memory access, whereas for a variable-length encoding, all $N-1$ characters in the string before the final N-th character must be decoded to determine the number of bytes used for that character. That is, random access may require a linear scan.

2.1.4 Reviewed implementations

Implementations of the string data type in 11 out of the 15 most popular programming languages according to the TIOBE index were reviewed and their primary characteristics are presented in table 2.1.

Language	Implementation	String mutability	Mechanism
1. Python	CPython	Immutable	UTF-32 code point array with Latin-1 optimisation [24]
2. C	libc	Mutable	Null-terminated byte array [25]
3. Java	OpenJDK	Immutable	UTF-16 code point array [13]
4. C++	libc++	Mutable	Null-terminated byte array with short-string optimisation [11]
5. C#	.NET	Immutable	UTF-16 code point array [16]
6. Visual Basic	.NET	Immutable	UTF-16 code point array [15]
7. PHP	Zend	Mutable	Byte array with copy-on-write [27]
10. Go	gc	Immutable	UTF-8 code point array [9]
11. Swift	Apple Swift	Mutable	UTF-16 code point array with copy-on-write [19]
14. Object Pascal	Delphi	Mutable	Code point array with copy-on-write [12]
15. Ruby	MRI	Mutable	Byte array [22]

Table 2.1: Key characteristics of strings in languages in the TIOBE index (Feb 2022)

Note: Some languages were omitted from the review: 'Assembly language', SQL, R, Matlab. These languages either do not define a string data type or are domain-specific programming languages and do not fall within the scope of this evaluation.

Array

Strings are almost always implemented as arrays of characters. All language implementation in table 2.1 use an array-based implementation for the string data type. This is so commonplace that most authors use the two interchangeably [6]. Array-based strings are also used in MRI, JRuby, Rubinius, which are three popular Ruby implementations.

Array-based implementations are simple and well-studied in literature. They exhibit constant random-access times (for byte-level access) and the amount of memory they occupy can be reasoned about easily, since that is only bounded by the number of characters in the string and the character encoding.

Rope

TruffleRuby's string implementation takes inspiration from the Cedar programming language [22] and is based on the rope data structure. Ropes are an immutable tree data structure, in which character arrays represent the leaves of the tree, and the operations between them (concatenation, repetition, slicing) comprise the internal nodes of the tree. By traversing the tree and applying the appropriate transformation, all characters of the rope can be enumerated.

Being an immutable data structure by design, mutability is not directly supported. In the general case, a mutation of a single character in a TruffleRuby string is represented by a tree with five new nodes (figure 2.1):

1. a node representing the left partition of the original string
2. a node representing the replaced character
3. a node representing the right partition of the original string
4. a node representing the concatenation of the left partition with the replaced character
5. a node representing the concatenation of the right partition with the previous node

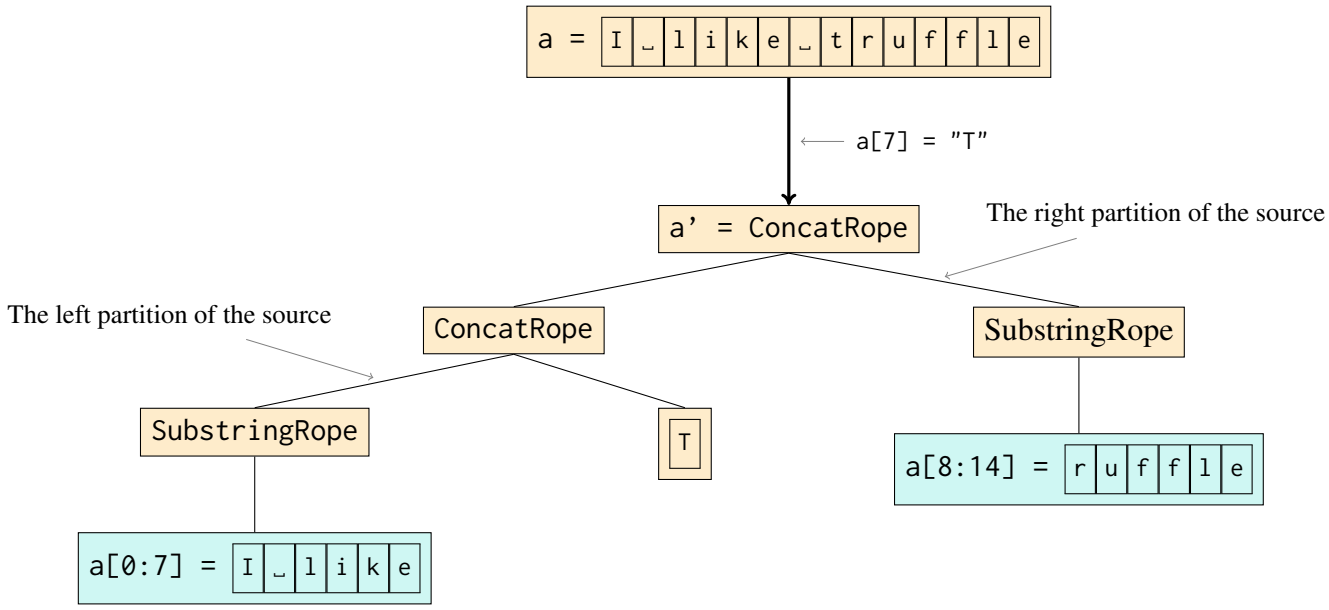

Figure 2.1: Replacing 't' with 'T' in a rope data structure

As string mutations accumulate, the tree gets deeper and deeper (figure 2.2). That is why, TruffleRuby uses heuristics to try and limit the depth of that tree. This is done by flattening the tree — essentially traversing the tree and replacing it with a single new leaf (character array) node.

Since existing nodes are immutable, ropes can be implemented as a persistent data structure, that is nodes can be shared between rope trees. As a corner case, the leaf node `a` in the figure is referenced from (shared between) two substring nodes of the same tree.

Most of the rope operations involve node allocation and are sensitive to the type of the node. The performance impacts of both are largely mitigated by the small object allocation pool in the JVM, the ability of TruffleRuby to specialise method implementation at runtime via JIT compilation, and escape analysis, which can eliminate certain cases of object allocation.

2.2 Ruby programming language

2.2.1 Dynamic programming

Ruby is described as 'a dynamic, open-source programming language with a focus on simplicity and productivity' [5] on the official language website.

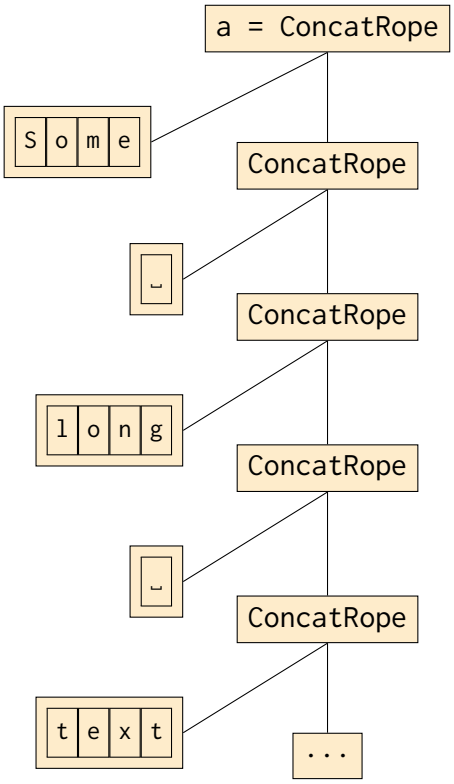

Figure 2.2: Rope representation of a list of words concatenated by whitespace

A programming language is referred to as dynamic, as opposed to static, whenever the operation of the language relies on modifications or extensions to the type system, types and objects present during the execution of the program. In static programming languages, the type system is fixed and not extended during execution.

The type system is a logical system of rules that determine the type of every value in the program. The values could be assigned to named bindings (variables) or intermediate results (resulting from the evaluation of an expression, like `a+1`). The type of a value determines what operations can be performed on it. For example, arithmetic division can be performed on two ”integer” values, but cannot be performed on one ”integer” value and one ”text” value.

The resolution of types and the set of possible operations is performed during compilation for static languages, and during program execution (runtime) for dynamic languages [1]. Dynamic languages may be preferred for many reasons, but they come at the cost of an overall reduction of execution speed, because many steps that typically happen during compilation are done during execution.

2.2.2 Popular implementations

The original Ruby implementation (reference implementation) is often called ‘CRuby’ (since it is written in C) or as ‘MRI’ (stands for ‘Matz’s Ruby Interpreter’). MRI executes Ruby code using interpretation (since Ruby 3.0, MRI includes a JIT compiler called ‘MJIT’).

JRuby is a Ruby implementation written in Java that executes Ruby either through interpretation or via JIT compilation to Java bytecode.

[1] This is called dynamic typing, but most dynamic languages are also dynamically typed

Rubinius is a Ruby implementation written in C++ based on LLVM (a compiler toolchain) and executes Ruby via JIT compilation to machine code.

TruffleRuby is another implementation of the Ruby programming language, which has quickly gained popularity for several reasons: on average, it performs much better than existing implementations for many tasks, it is backed by a large and well-known corporation (Oracle), it is implemented in a higher-level programming language (Java), and it runs on the portable Java Virtual Machine (JVM). TruffleRuby has some unique advantages that will be relevant to this work.

2.3 TruffleRuby

2.3.1 Java Virtual Machine

A Java Virtual Machine/JVM is the infrastructure on top of which Java bytecode can be executed. Java is not the only language that can be compiled to Java bytecode, so despite being called Java Virtual Machines they can be a suitable target for languages other than Java. The de-facto standard JVM is the HotSpot JVM, which uses a multi-tier optimising compiler with adaptive optimisation and de-optimisation to enable high-performance Java programs. [18]

2.3.2 Just-in-Time compilation

JVMs often execute Java bytecode by interpretation [20], which is the process of matching input program sequences to their implementations in the language of the interpreter. Almost all JVMs also employ Just-in-Time/JIT compilation [20], which consists of taking an input program sequence and producing machine code suitable for the platform of execution. This machine code is then kept in memory and executed directly by the processor.

2.3.3 Graphs for intermediate representation of code

During compilation, source code goes through multiple intermediate representations (IR) [1, 18, 20]. These IRs often take the form of directed acyclic graphs (DAGs) [1, 20]. The DAG representation is desirable, because results of expressions are represented as nodes in the graph, and the dependent subexpression of the expression are the reachable nodes in from the expression node in the graph.

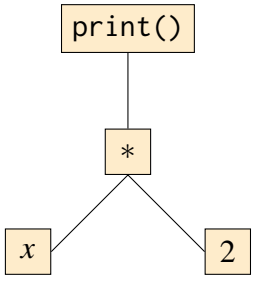

Figure 2.3: `print(x*2)` represented as a DAG

2.3.4 GraalVM

GraalVM is a novel JVM implementation which includes the Graal compiler. GraalVM ships with the GraalVM SDK, which includes the Truffle language framework. The Truffle language framework is a 'smart' interpreter integrated with GraalVM. The Truffle interpreter interprets the DAG representation of the program logic (called Graal IR). When run on the GraalVM, the Truffle interpreter can compile code paths in the graph into machine code via JIT compilation [30].

2.3.5 Partial evalution

"Partial evaluation (PE) is the process of creating the initial high-level compiler intermediate representation (IR) for a guest language function from the guest language interpreter methods (code) and the interpreted program (data)." ([31])

Ruby has fairly complex semantics. The symbol for addition + could mean the arithmetic operation of addition, string concatenation or can have its behaviour overridden by user code.

These complex behaviours can require more machine code to be emitted to preserve the full semantics of the operation. Partial evaluation is a technique which allows only a subset of specialised implementations (specialisations) guided by the program flow and data inputs to be enabled [30].

For example, TruffleRuby can specialise methods based on argument types. This is done for the different types of rope nodes used in the string implementation [22].

2.3.6 Graph rewriting

Finally, TruffleRuby can revert any of the specialisations mentioned above. If the number of specialisations becomes too large or if a code path that is deemed unlikely by the TruffleRuby developer is hit, the developer can invalidate the code generated for the specialisation, disable the specialisation and let the runtime select a different specialisation [30].

CPUs contains a unit called the ALU (arithmetic logic unit), which performs fixed-width arithmetic. Ruby supports arbitrary-precision integer arithmetic, but because the fixed-width arithmetic is implemented in hardware, it is much faster than software-based arbitrary-precision arithmetic. An efficient way to perform addition, could be to use fixed-width arithmetic and assume the result does not require more than the fixed number of bits to be represented. If that assumption is broken, this specialisation can be replaced with another which uses arbitrary-precision integers, but which is also slower to execute (hence not enabled by default) (figure 2.4).

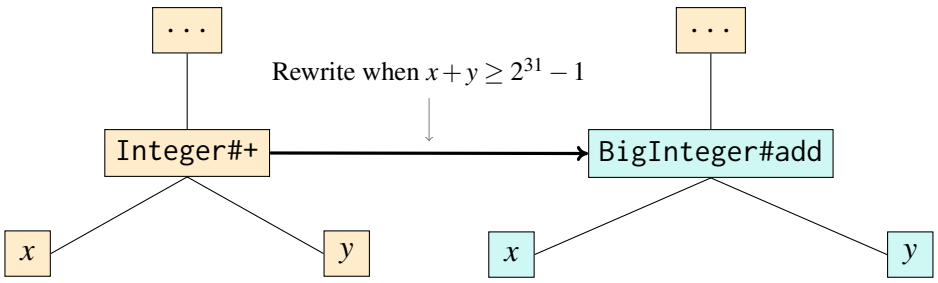

Figure 2.4: Graph rewriting

Rewriting is an expensive operation, because it requires for machine code that has been generated to be pruned, and for the DAG to be modified before the new specialisation can be executed. The new specialisation will be executed using the interpreter until the runtime decides it would be beneficial to compile it to machine code using JIT compilation. The ability to do this is often beneficial for long running applications. As the same code path is exercised many times over a single execution of a program, the most-complete specialisations will be selected, and no new code will need to be generated. This is often referred to as the warm-up period.

Unfortunately, it is also possible for re-specialisation loops to occur. In that case, program execution causes repeated de-optimisation and re-specialisation, without reaching a stable state. Because this leads to unpredictable execution speeds, it can also make benchmarking very difficult.

2.3.7 Caching

Caching is a technique in which the results of previous operations are stored in a table and reused when possible. For example, a cache can be used to reduce the number of objects allocated when the object identity is irrelevant.

Inline caches

Inline caches are caches that are local to a call-site and are implemented in TruffleRuby by storing the state of the cache in the node of the DAG defining the semantics of that operation. For example, the result of a method lookup can be cached in the method call node so that the call target is quickly accessible the next time that line of code is executed [17].

Polymorphic inline caches

Since Ruby is a dynamic programming language, the result of a method lookup for 'eql?' [2] for an object of class Array will be different from the result of that lookup on an object of class String. This means multiple entries might be necessary for an effective inline cache. Polymorphic inline caches solve this problem by using multiple entries per call-site.

The several different types of ropes in the String implementation in TruffleRuby rely on the use of polymorphic inline caches to achieve good performance [17, 22].

2.3.8 Benchmarking

Benchmarking is the process of evaluating the performance of a piece of software. For the results to be viable, one must ensure minimal interference from other processes running on the same machine. In the case of dynamic languages which use multi-tier JIT compilation, one must also ensure that JIT compilation has largely stopped.

Microbenchmarking

Microbenchmarking usually refers to evaluations performed on a small piece of code that can be executed relatively quickly. The idea is that the local performance of the system can be an indicator of overall system performance.

[2] The method which establishes equality between two objects.

The many optimisation capabilities of TruffleRuby like JIT compilation, inline caching, and escape analysis can make the results of microbenchmarks unusable [7].

2.3.9 Flame graphs

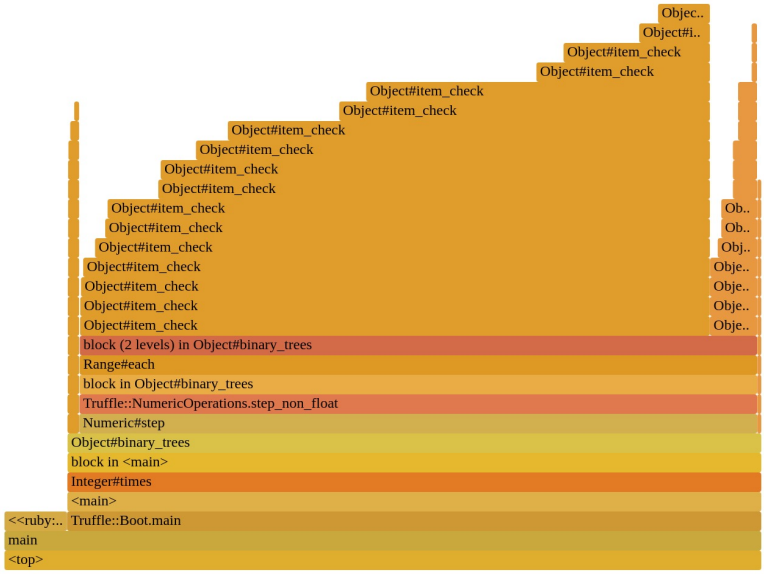

Figure 2.5: Flame graph of the execution of a divide & conquer algorithm

Flame graphs are one way of visualising the set of stack traces captured during the execution of a process [14]. Flame graphs are informationally dense and can be interpreted in both axes. Only two properties are relevant in the context of this project – (1) the block at the top represents the function that was active when the sample was collected and everything below it is the call stack, and (2) the width of the block indicates the how frequently that function was active whenever stack traces were collected.

Figure 2.5 shows an example flame graph generated from the execution of a divide & conquer algorithm. One can see how the recursive calls to `Object#item_check` are stacked on top of each other, with the topmost block indicating the recursion bottoming-out.

2.4 Related work

2.4.1 Specializing Ropes for Ruby

In a comparison between array-based and rope-based implementation of Strings in TruffleRuby, Menard et al. [22] looked at the results of three benchmarks. Two of those benchmarks were microbenchmarks. The third benchmark measured the performance of ERB, which is an HTML templating framework written in Ruby.

The authors concluded that the specialised ropes implementation performs within 0.9x-9.4x for the select use cases used in the evaluation.

Note on the microbenchmarks

Menard et al. [22] argues that in the specific case of the second microbenchmark 'micro-string-index', the observed 6.8x speed up is due to how the string in this specific benchmark is constructed as a concatenation of two leaf ropes, and how some of the work otherwise necessary due to the variable-length encoding can be skipped thanks to the underlying partitioning of the string. The authors then clarify that when the rope is flattened, the performance differential between TruffleRuby's Rope implementation and MRI is 2x, which is also within the range of the TruffleRuby Array implementation (also 2x the performance of MRI).

At some point after the publication of the paper, the implementation of the `String#index` method exercised in that benchmarks have been amended to always flatten the rope, eliminating that performance improvement.

The Ruby language features mutable strings, but string mutation is not featured in the evalution performed by the authors.

HTML template rendering

The HTML templating benchmark is the third benchmark included in the paper. It compares the performance of the two implementations in a realistic scenario. HTML templating is a concatenation-heavy process and is where the lazy nature of ropes gives them an edge.

After speaking to the TruffleRuby project lead, Benoit Daloze, it became clear that HTML templating is a core use-case for the TruffleRuby team. The same type of benchmark was used in the development process of this project, to detect performance regressions.

Chapter 3

Implementation

This chapter provides an in-depth look at the key issues with the baseline TruffleRuby implementation. An improved design of the rope data structure and two implementations of this design are presented. The first implementation is an object-oriented implementation which does not take full advantage of the TruffleRuby interpreter for advanced optimisations at runtime, but serves to confirming the benefits of the proposed design. The second implementation is uses the runtime optimisation features of the TruffleRuby interpreter. It employs a new profiling technique to intelligently select between string implementations and contains additional optimisations, only made possible by the tighter integration with TruffleRuby.

3.1 Initial observations

The Raytracer benchmark measures the performance of Ruby runtimes on the task of ray tracing an image [8]. The benchmark uses the *imageruby* image processing library for Ruby. Imageruby uses a string to encode and process the image, and the `String#[]=` method to change the values of individual pixels.

The poor performance on this benchmark was reported in the GitHub issue 'Raytracer benchmark is 34x slower than with MRI' [29] created on the TruffleRuby project source repository.

By using a flame graph (figure 3.1), it was possible to verify that the TruffleRuby implementation spends the majority of the execution time in the `String#[]=` method, which is the method performing the string mutation.

These observations are a primary motivator behind this project.

3.2 Implementation requirements

The results of the Raytracer benchmark show that TruffleRuby performs poorly in string mutation workloads and that the core string mutation method `String#[]=` is where most of the execution time is spent.

The TruffleRuby implementation of `String#[]=` is implemented by replacing the rope representing the original string with a rope with more rope nodes (figure 2.1). This larger rope structure requires additional memory allocation. In contrast, the MRI implementation uses an array-based string (table 2.1) implementation and simply overwrites the character in the existing array.

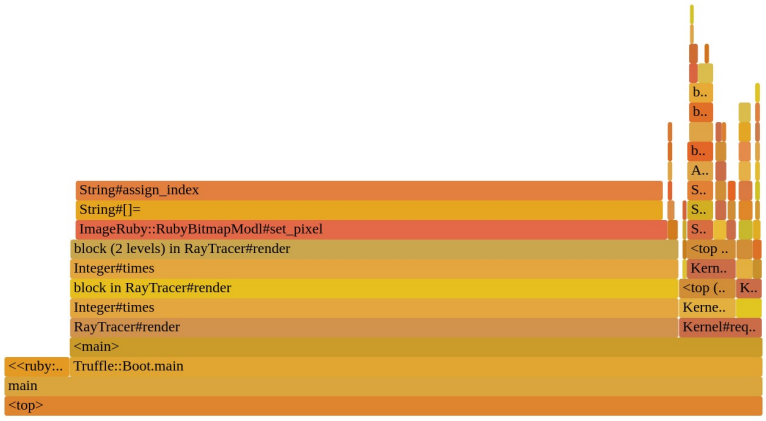

Figure 3.1: Flame graph of the TruffleRuby execution of Raytracer

The TruffleRuby string implementation outperforms MRI on other important workloads (section 2.4.1). The perfect solution will improve the performance of `String#[]=` without compromising the performance of other operations.

The existing implementation of ropes and strings in TruffleRuby amounted to $\sim$16,000 lines of code at the start of this project[1]. A great deal of engineering effort has gone into developing and testing this code. The perfect solution to the slow string mutation problem would not require major portions of this code to be modified.

3.3 New mutable rope design overview

This paper presents a design and implementation that aims to solve the problem of string mutation being slow, without sacrificing performance in other areas. This section presents the key design idea of mutable leaf ropes. An implementation of this design is presented in section 3.4. This implementation is then improved in section 3.5. The latter implementation addresses some key issues by making use of features available in the Truffle runtime.

3.3.1 Limitations of the existing implementation

The TruffleRuby string implementation makes use of a tree-based data structure called a rope to represent character strings. In this data structure, sequences of characters are stored in leaf nodes (see 2.1.4). Each leaf node stores characters as an encoded array of bytes. Unlike in an array-based string implementation, these sequences cannot be modified. This is because leaf nodes may be subject to aliasing by multiple rope nodes (figure 3.2).

The TruffleRuby ropes implementation uses structural sharing for substring, string concatenation and string repetition nodes (for performance reasons). As a result of the modification of the backing array of the leaf node of the string variable `a`, the contents of `b`, `c` and `d` will also be modified, breaking the semantics of the substring, concatenation and repetition operations in Ruby.

[1] Number calculated as total number of lines of related source files

```
a = "foobar"
b = a[1:3]
c = a + "baz"
d = a * 8
```

Listing 3.1: Operations on strings which lead to shared state

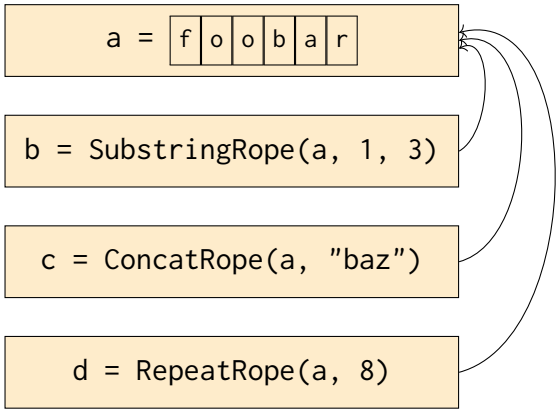

Figure 3.2: Shared state due to persistence in ropes

3.3.2 Proposed design

To work around these limitations, the proposed design adds the boolean flag `isReadOnly` to track the occurrence of structural sharing for each leaf node. The flag is false only if the node is **known to not be aliased**. The flag is true if the node is **potentially aliased**.

The `isReadOnly` flag implicitly creates two subtypes of leaf nodes: read-only leaf nodes, and mutable leaf nodes. The idea is that the mutable leaf ropes can be treated as array-based strings, with similar performance characteristics.

It is helpful to think of this as *intrusive ownership tracking*. A node is 'exclusively referenced' when this node is only referenced from a single Ruby string, it is the root node of its tree, and it is only node of that one tree. A node is 'shared' whenever any of those conditions cannot be established to be true (because it is false or because the node is created by the TruffleRuby runtime and passed to the executing Ruby program, see the note about reference counting below).

The proposed design establishes the invariant that *mutable leaf nodes are never aliased in a way which could lead to side-effects*. This allows for the performance advantages of an array-based strings to be exploited whenever deemed useful by the TruffleRuby developer.

Reference counting

Reference counting can in principle also be used to establish the type of ownership (unique or shared). It was not preferred over the `isReadOnly` flag approach for three important reasons.

The first reason why reference counting was not used is that in addition to establishing when the rope is referenced by a new referee (aliasing may occur), in order to decrement the reference count, one must also establish when that reference is no longer used. When a leaf rope is referenced by a string object, the reference count must be decremented when that object becomes unreachable. Unfortunately, the use of finalizers in Java is discouraged as it can be unpredictable and slow [3], which leaves no option for correctly decrementing the reference count.

The second reason why reference counting was not used is the same as the reason the phrase 'known to not be aliased' was used to explain the semantics of the `isReadOnly` flag. TruffleRuby uses ropes internally for string processing. This means that not all ropes are refer-

enced from a string variable or referenced from another rope. Using reference counting would require a very careful analysis of all places where rope nodes are used and is thereby error prone.

The third reason is that reference counting is more expensive to do. Since Ruby supports multithreading, reference increments and decrements must be atomic. These operations require additional or slower running instructions on all platforms supported by the TruffleRuby runtime.

Note on multithreading

Since read-only leaf nodes are immutable, their contents can be accessed without the need for synchronisation. Their values cannot be changed, and thus no race conditions can occur. Since mutable leaf nodes guarantee no aliasing may occur, their values can be changed without the need for synchronisation.

Note: Concurrent modification of a (string) variable is a race condition (by definition).

3.4 Object-oriented implementation of the mutable rope design

This section covers the initial object-oriented implementation that was developed. In contrast to the Truffle implementation presented in section 3.5, this implementation does not use any of the Truffle runtime features. Its purpose is to verify that the proposed design for allocation-free mutable strings is sound and that it can bring the expected performance improvement.

3.4.1 Motivation

The object-oriented implementation was the first implementation of the proposed design. It does not require a deep understanding of the Truffle runtime. As opposed to the Truffle implementation which requires reasoning on the DAG node level, this implementation only requires understanding of the rope data structure.

As established in section 3.1, optimising the expensive `String#[]=` is one of the key objectives of this project.

3.4.2 Relevant classes

Basic understanding of the core classes mentioned below is necessary to understand this implementation. It is important to stress that Ruby code being executed is not aware of any of these implementation details.

Rope

The `Rope` class is the base class for all 8 types of concrete ropes used in TruffleRuby: `NativeRope`, `AsciiLeafRope`, `ValidLeafRope`, `InvalidLeafRope`, **`ConcatRope`**, **`SubstringRope`**, **`Repeat-`**

Rope, `LazyIntRope`. Some of those classes can introduce structural sharing (reference other ropes) and have been emboldened as they are relevant to ownership tracking.

LeafRope

`LeafRope` extends the `Rope` abstract base class and is in turn the base class of the three types of specialised leaf ropes `AsciiLeafRope`, `ValidLeafRope` and `InvalidLeafRope`. These classes support code points in overlapping but distinct ranges: 7-bit code points, valid code points and invalid code points, respectively. The use of separate classes allows specialised and faster operations to be used when possible [22].

RubyString

`RubyString` is the TruffleRuby implementation of the `String` class in Ruby. Objects of this class encapsulate a reference to a `Rope` and well as some additional information regarding the state of the `String` object.

3.4.3 List of API changes

The existing API has been extended with new operations. Their simplified prototypes are presented as source code for brevity.

```
1 class ManagedRope {
2  ManagedRope getShared();
3  /* ... */
4 }
```

Listing 3.2: New operations defined on the `ManagedRope` class

```
1 class LeafRope {
2  boolean isReadOnly();
3  LeafRope getMutable(CodeRange
     outCodeRange);
4  void replaceRange(int index, byte[]
     bytes);
5  /* ... */
6 }
```

Listing 3.3: New operations defined on the `LeafRope` class

Shared node ownership

Read-only ropes are obtained from the `getShared` method by (1) returning the callee object (`this`) when the rope is not a `LeafRope` or by (2) returning a read-only leaf rope when the callee object is a leaf rope.

Mutable ropes are converted into read-only ropes by **defensively copying** (revised in section 3.5) the contents of the mutable ropes into a new read-only rope. This is done to ensure that later operations performed on the mutable rope do not affect the newly created read-only rope.

Exclusive node ownership

Mutable ropes are obtained from the `getMutable` method by (1) returning the callee object if it is a mutable leaf node (the read-only flag is false) **and has the requested code range** or by

(2) copying the contents of the rope node into a new mutable leaf node with the requested code range.

As mentioned in section 2.4.1, TruffleRuby encodes the code range of ropes in the type system. If the code range passed to the `getMutable` method does not match the code range of the target, a new mutable leaf rope is constructed by **copying** (revised in section 3.5) the underlying byte array.

Read-write random access

Most importantly, the `replaceRange` method is used to directly modify the contents of the backing array. This method can only be called on mutable leaf ropes (enforced by runtime assertions).

Note about hash invalidation

Rope instances compute their corresponding hash code lazily and cache the resulting integer. This cache must be invalidated whenever the array of a mutable leaf rope is modified. This is computationally inexpensive and is done by setting the hash code to the initial value used for that field (zero).

3.4.4 Unsafe aliasing

In order to prevent unsafe aliasing, only read-only ropes should be referenced from multiple places in the code. This is straightforward to implement with the semantics proposed in the design — recall that we want to establish when there is potential aliasing or definitely no aliasing occurring (section 3.3.2). This implementation takes a conservative approach. All ropes except the ones returned by `getMutable` are created as read-only ropes (potentially aliased) from the beginning. The only way to obtain a mutable rope is through that method.

This conservative approach is preferrable, because it does not require as extensive validation and eliminates the need to explicitly reason about all uses of the `Rope` class or any of its subclasses in the TruffleRuby codebase. As mentioned previously, TruffleRuby uses the `Rope` classes internally in many places.

The only points of interest for our implementation are the places in the code where:

- rope nodes become referenced as child nodes from other rope nodes
- rope nodes become referenced from a `RubyString`
- rope nodes may be added to a polymorphic inline cache (section 2.3.7)

The first amounts to inserting calls to `getShared` in the constructor of `ConcatRope`, `SubstringRope` and `RepeatRope`. The second to inserting another call to `getShared` in the `setRope` accessor of `RubyString`. The last possibility for unsafe aliasing only made itself apparent after running the full suite of benchmarks and was not caught during ordinary testing of the code base. This is because this type of unsafe aliasing occurs in very specific cases: when a mutable rope (returned only from `getMutable`) is compared to another rope in a loop. Nonetheless, this situation is easily remedied by requiring that rope nodes are read-only before entering the cache.

3.4.5 Optimising `String#[]=`

The tools developed above are necessary to support the optimisation of `String#[]=`.

The `String#[]=` method is defined in Ruby. Since the exact behaviour of this method depends on the type of arguments passed to it, some runtime checks are done before execution finally reaches the Java side. The Truffle node `StringSplicePrimitiveNode` is where the resulting rope is computed.

Several specialisations of this method are present (for specialisations, see section 2.3.5). As part of this implementation, a new specialisation called `spliceExactReplace` is added. This newer specialisation replaces the existing specialisation which uses a concatenation of ropes to compute the resulting rope (figure 2.1).

```
class StringSplicePrimitiveNode extends Node {
  @Specialization(guards = {
    "ropeByteLengthEquals(other, byteCountToReplace)",
  })
  protected RubyString spliceExactReplace(
      RubyString string, RubyString other, int spliceByteIndex,
      int byteCountToReplace, RubyEncoding rubyEncoding) {

    Rope rope = string.getRope();
    Rope otherRope = other.getRope();
    // Ensure the resulting node has the more general code range
    CodeRange outCodeRange = commonCodeRange(
      rope.getCodeRange(), otherRope.getCodeRange());
    LeafRope newRope = rope.getMutable(outCodeRange);
    // Modify the internal byte array directly
    newRope.replaceRange(spliceByteIndex, otherRope.getBytes());
    // Update the reference held by the string
    string.setRope(newRope, rubyEncoding);
    return string;
  }
  /* ... */
}
```

Listing 3.4: Optimised `String#[]=` (simplified)

Using the `guards` property of the `Specialization` attribute, one can specify the preconditions for enabling that specific specialisation. In the case of `spliceExactReplace`, the specialisation may only be used when the contents of the source rope `other` is used to replace a **range of bytes of the same length**. This is because otherwise, the backing array will have to be resized by copying its contents into a new differently sized array. The cost of resizing might be greater than that of concatenation-based splicing.

3.4.6 Benchmarks

A set of 3 benchmarks were used to verify that the optimisation presented above had the desired effect:

1. *buffer-10k*, which is a microbenchmark which exercises `String#[]=` on ASCII strings of 10,000 characters
2. *raytracer* (see 3.1)
3. *liquid-cart-render* (HTML templating, see 2.4.1))

Experimental setup

The experimental setup is described in section 4.4.

Results

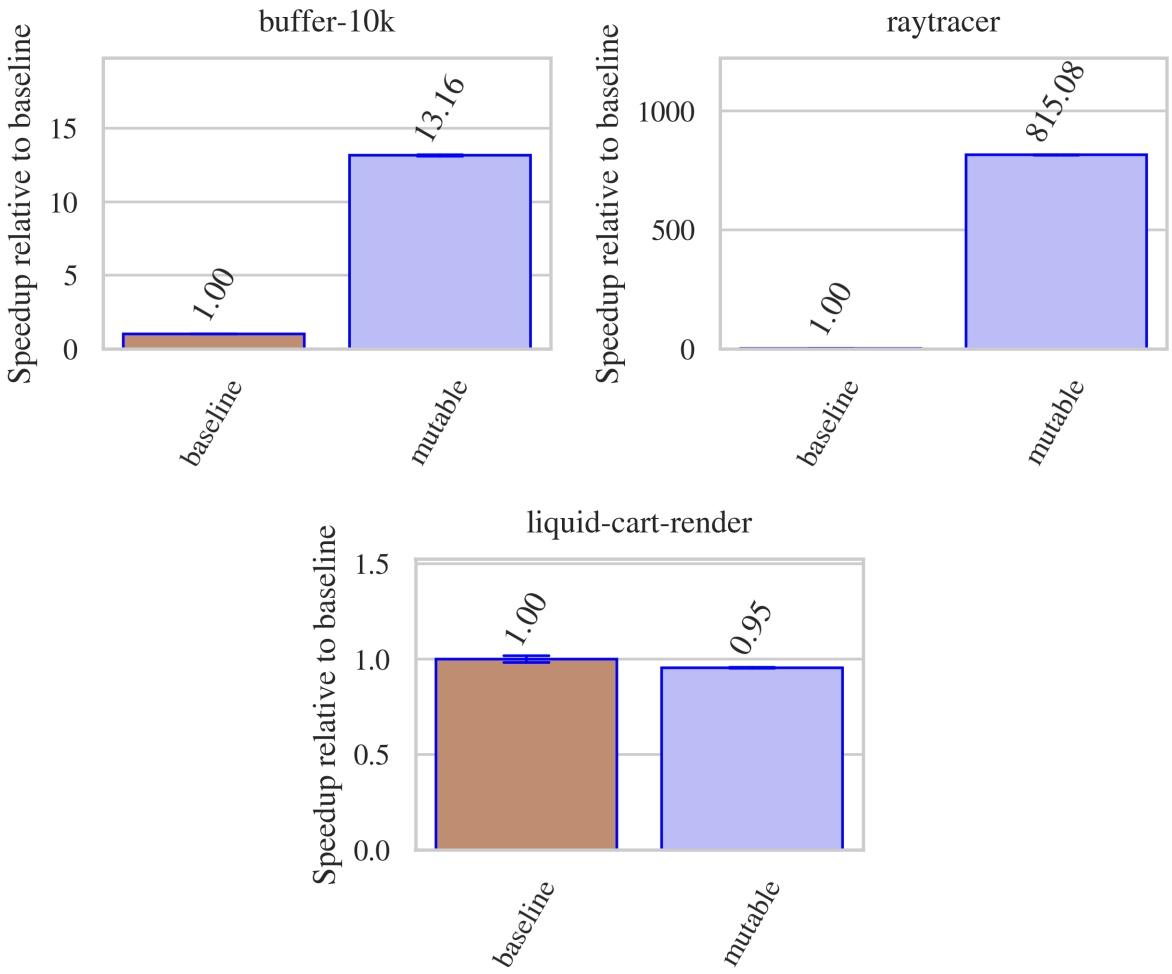

Figure 3.3: Benchmarks of the optimised `String#[]=`

The preliminary benchmark results in figure 3.3 indicate that the design is functional. The implementation fulfills the implementation requirements set out in 3.2. There are still at least two problems with this implementation which are addressed in 3.5.

3.5 Truffle implementation of the mutable rope design

3.5.1 Motivation

The object-oriented implementation has a regressive case not covered by the benchmarks above. Recall that the first time a read-only string is altered, its backing array needs to first be copied into a mutable string. For a small number of string mutations made to a large string or for many

string mutations interleaved with non-mutating operations (like concatenation), this implementation leads to excessive copying. An extreme case of the latter is shown in listing 3.5. This regression is fixed in the Truffle implementation using a custom profiling technique.

```
1 n.times {
2   # String#[]= calls getMutable on a read-only rope -> string is copied
3   string[i] = replacement
4   # ConcatRope calls getShared on a mutable rope -> string is copied
5   string << str
6 }
```

Listing 3.5: Regressive case for optimised `String#[]=`

The second problem with the object-oriented implementation is that it is not optimised for partial evaluation (2.3.5). This is because of the presence of virtual function calls which cannot be inlined; the checks needed to establish whether the rope is a `LeafRope` subtype; whether the `isReadOnly` property is true, along with both code blocks of that conditional are necessarily compiled. TruffleRuby is able to automatically detect some of those issues and emits performance warnings when the `--engine.TracePerformanceWarnings` command line option is used. The Truffle implementation moves the implementation of these methods into Truffle nodes and uses specialisation and branch profiling to prevent the compilation of unused branches [30] and virtual function calls.

3.5.2 Specialised profiling for mutable ropes

Listing 3.5 shows a case where the mutable rope optimisation is undesirable. A custom profiling technique was developed as part of this project to detect those cases with minimal overhead.

The technique is based on observed usage patterns of index assignment operations like `String#[]=` and `String#setbyte`. An excerpt from the Raytracer benchmark is shown in figure 3.6.

The code in listing 3.6 shows how the `set_pixel` method is called in a loop $screenHeight * screenWidth$ times to update the values of each channel (red, green, blue and transparency) for each pixel using the `String#[]=` method.

Rope reuse

The specialised profiling technique developed as part of this project is based on the observation that in loops where strings are modified using `String#[]=`, *the assignee rope node of the i-th iteration is the result of the* $i-1$*-th operation at the same source location*. This situation will be referred to as 'rope reuse'.

Explanation

When the mutable ropes implementation is used to implement `String#[]=`, **the mutable rope object** resulting from the assignment to `@array[pointindex]` on line 18 in listing 3.6 is **the same rope object** that will be assigned to (assignee) when line 18 is executed again (up to $screenHeight * screenWidth$ times).

Conversely, the rope object resulting from the assignment to `string[i]` on line 3 in listing 3.5 **is not the same rope object** that will be assigned to (assignee) when line 3 is executed again. This is because a new `ConcatRope` object is created on line 5.

```
/* raytracer.rb */
def render(scene, image, screenWidth, screenHeight)
  screenHeight.times do |y|
    screenWidth.times do |x|
      color = self.traceRay(/* ... */)
      r,g,b = Color.toDrawingColor(color)
      image.set_pixel(x,y, ImageRuby::Color.from_rgba(r,g,b, 255))
    end
  end
end

/* imageruby/../rbbitmap.rb */
def set_pixel(x,y,color)
  index = (y*@width + x)
  pointindex = index*3
  @array[pointindex+2] = color.r.chr
  @array[pointindex+1] = color.g.chr
  @array[pointindex] = color.b.chr
  if color.a != 255 then
    @alpha = 255.chr*(@height * @width) unless @alpha
    @alpha[index] = color.a.chr
  end
  self
end
```

Listing 3.6: Usage of `String#[]=` in Raytracer

Implementation

The special Truffle profiling node `RopeReuseMonitorNode` must be inserted after a string assignment operation and accepts two values as its input: the assignee rope object and the result of the assignment.

Truffle uses an integer value called a 'node cost' when making decisions regarding code path optimisation. Because `RopeReuseMonitorNode` only gathers information about the execution, its node cost is set to zero.

Truffle nodes are stateful and that state is generally used for polymorphic inline caches [30], but is available for arbitrary bookkeeping. `RopeReuseMonitorNode` uses this state to track three values (1) the last assignment result, (2) the number of times it is executed when *the assignee object and the last assignment result are the same object* (`ropeReuseCount`), and (3) the number of times it is executed when *the assignee object and the last assignment result are not the same object* (`ropeNoReuseCount`).

Because holding a reference to a rope object will prevent that object from being subject to garbage collection, the Java object identity hash code is used to identify objects. The object identity hash code is not guaranteed to be unique throughout the execution of the program, which means that the profiling code might assume two objects are identical even if they are not. This has not been an issue in practice.

The return value of `RopeReuseMonitorNode` is called the 'rope reuse factor' and is the ratio between the number of rope reuses and the total number of executions of the node. The rope reuse factor is high ($\gg 0.5$) for string assignments in listing 3.6 and low ($\ll 0.5$) for string assignments in listing 3.5.

$$ropeReuseFactor = \frac{ropeReuseCount}{ropeReuseCount + ropeNoReuseCount}$$

To prevent rapid fluctuation in the rope reuse factor, the `ropeReuseCount` and `ropeNoReuseCount` variables are initialised to a (heuristic) non-zero value.

3.5.3 Profiling-guided mutable rope optimisation

Truffle allows compiler developers to rewrite parts of the DAG by disabling specialisations at runtime (figure 2.4). This technique is used in conjunction with the custom profiling technique to disable the mutable rope optimisation in undesirable cases such as the one shown in listing 3.5.

Interpreting profiling results

The `EphemeralProfileRopeMutationNode` is another profiling node which builds on top of the `RopeReuseMonitorNode`, and applies thresholding to select between the values of the `OptimizationHint` enumeration (listing 3.7). The node is called 'ephemeral', because it uses DAG rewriting to replace itself with a constant value once an `OptimizationHint` other than `UNDEFINED` is selected. During the rewrite, the generated code is pruned. After that happens, the `OptimizationHint` value for the calling node becomes a partial evaluation constant, allowing the runtime to apply further optimisations during code generation.

```
1 enum OptimizationHint {
2   UNDEFINED,
3   MUTABLE,
4   IMMUTABLE,
5 }
```

Listing 3.7: The `OptimizationHint` enumeration

The thresholding values as well as the initial values for the `ropeReuseCount` and `ropeNoReuseCount` fields are heuristic and selected through reasoning and observation. The DAG rewriting operation is expensive, because it prunes the generated code [30]. On the other hand, it is beneficial to allow for as much profiling to occur.

In the submitted code, the initial value for `ropeReuseCount` and `ropeReuseCount` is 1,000. The threshold for selecting the `MUTABLE` hint is 0.9, and the threshold for selecting the `IMMUTABLE` hint is 0.1. The number of invocations necessary before the profiling code is eliminated and replaced with a constant is calculated in figure 3.4 for the cases of maximum rope reuse (listing 3.6) and none/minimum rope reuse (figure 3.5), respectively. Because all numbers reported as part of the evaluation at the end of the report only include measurements taken after code generation has stabilised, the exact values here do not affect them.

Note: These values were selected assuming a single-tier compilation mode with the threshold set to the default value of 1,000.

$$\frac{100+N-1}{100+100+N+1} \geq 0.9 \qquad \frac{100}{100+N} \leq 0.1$$
$$N \geq 819 \qquad N \geq 900$$

Figure 3.4: Minimum number of executions for selected rope reuse thresholds

Specialising code using optimisation hints

The Truffle implementation of the mutable rope design uses the `OptimizationHint` returned by the `EphemeralProfileRopeMutationNode` to selectively toggle the mutable rope optimisation for `String#[]=` and `String#setbyte`.

```
@Primitive(name = "string_splice_with_hint", /* ... */)
@ImportStatic(StringProfilingNodes.RopeOptimizationHint.class)
public abstract static class StringSpliceWithHintPrimitiveNode extends
    PrimitiveArrayArgumentsNode {

  // Default to mutable implementation
  @Specialization(guards = { "hint != IMMUTABLE" })
  protected RubyString spliceMutable(
      RubyString string, Object other, int spliceByteIndex,
      int byteCountToReplace, RubyEncoding rubyEncoding,
      StringProfilingNodes.RopeOptimizationHint hint,
      @Cached MutableStringSpliceNode spliceNode) {

    return spliceNode.execute(string, other, spliceByteIndex,
      byteCountToReplace, rubyEncoding);
  }

  @Specialization(guards = { "hint == IMMUTABLE" })
  protected RubyString splice(
      RubyString string, Object other, int spliceByteIndex,
      int byteCountToReplace, RubyEncoding rubyEncoding,
      StringProfilingNodes.RopeOptimizationHint hint,
      @Cached StringSpliceNode spliceNode) {

    return spliceNode.execute(string, other, spliceByteIndex,
      byteCountToReplace, rubyEncoding);
  }
}
```

Listing 3.8: Using the mutable rope optimisation hint for `String#[]=`

The `StringSpliceWithHintPrimitiveNode` is the core part of the optimised implementation of `String#[]=` and its definition is shown in listing 3.8. The mutable rope optimisation is enabled by default. This is done by putting the relevant specialisation first. When the hint value is `IMMUTABLE`, the specialisation for the legacy concatenation-based assignment is used (figure 2.1).

3.5.4 Reducing defensive copying

The need for defensive copying when converting from mutable ropes to read-only ropes (see 3.4.3) can be eliminated, by making a small amendment to the mutable rope design. By allowing the `isReadOnly` flag to be changed only from `false` to `true`, a mutable rope can become read-only. This is done by `LeafRope#makeReadOnly()`.

It is not immediately obvious that this will not lead to unsafe aliasing. Considering the relationship between rope mutability and the number of allowed references to the rope (see 3.4.3), a mutable rope can only be referenced from one `RubyString` object at any time. Section 3.5.7 talks about how runtime checks can be used to verify that.

By using `makeReadOnly()`, the need for defensive copying of the backing array to a new read-only leaf is eliminated without any downside.

3.5.5 Dynamic arrays for mutable ropes

The Java `StringBuilder` class uses a dynamic array for incremental string building (listing 2.3). These operations are currently implemented using a `ConcatRope`. By growing the backing array of mutable ropes to a length larger than is strictly necessary and writing the contents of the source rope in the unused space, the behaviour of the `StringBuilder` class can be emulated.

Dynamic arrays and code-compatibility

The implementation of dynamic arrays required much care, because of engineering decisions made by the TruffleRuby developers. A major complicating factor was the fact that the underlying `byte[]` Java object stored in `Ropes` is not accessed through an abstract interface. Java `byte[]` objects are non-dynamic arrays with a fixed capacity. Dynamic arrays in Java are implemented by pairing a non-dynamic array like `byte[]` with an integer value indicating the number of used slots in the array. Without knowledge of the additional value, any existing code which performs operations on the underlying `byte[]` is free to assume that an array `arr` stores exactly `arr.length` bytes.

My initial attempt to migrate the code base away from the use of `byte[]` to a more abstract representation required more than 1,000 lines of code to be changed before the successful compilation. Many of those occurrences required additional reasoning to be applied, because of the dependency on external libraries which only accept `byte[]` arguments paired with either an offset and a length, or an integer range. The changed code did not pass the included unit tests (likely due to programming errors) and the approach was deemed unsuitable given the time frame of the project.

Because of these considerations, and alternative approach was taken. Instead of replacing the use of `byte[]` with a different type of array, a lazy truncation approach was used. By introducing an additional `byteLength` field to the `Rope` class, and lazily copying the internal array whenever its length is greater than the `byteLength`, full compatibility with the existing code base is achieved. In the cases where the changes to the calling code were relatively simple, the calling code was modified to be aware of the `byteLength` field and the fact that a dynamic array is being used.

```
class Rope {
  private byte[] bytes;
  public byte[] getRawBytes() {
    return bytes;
  }
}
```

```
class Rope {
  private byte[] bytes;
  private int byteLength;
  public byte[] getRawBytes() {
    if (bytes == null) {
      return null;
    }
    if (byteLength < bytes.length) {
      bytes = Arrays.copyOfRange(
        bytes, 0, byteLength);
    }
    return bytes;
  }
}
```

Figure 3.5: Rope with support for dynamic arrays

Note: The TruffleRuby team is in the process of replacing the use of `byte[]` with a more abstract implementation. Unfortunately, their work was not ready by the time this project was due. Their work will obviate the need for lazy truncation.

New operations

The `AppendMutableNode` appends the contents of the source node to the destination mutable rope and resizes the dynamic array when necessary. The same resize policy as the one used by the Oracle JDK is used (figure 3.6). The `byteLength` and `characterLength` properties are updated and any cached hash code is cleared. The contents of the source string is written directly into the backing array of the destination mutable rope without allocation of a temporary array and without flattening. This is supported by an additional node called `WriteBytesMutableNode`.

$$newCapacity = max(currentCapacity * 1.5, minCapacity)$$

Figure 3.6: Dynamic array resize policy

The implementation of the Ruby string in-place append method `String#<<` is changed to use `AppendMutableNode` only when **the destination rope is already a mutable leaf rope**. This is to ensure that the mutable implementation is not used after a different type of operation is used to modify the string.

The implementation of the Ruby string duplication method `Kernel#dup` is changed to return a mutable rope when the source rope is mutable or *when the source rope is empty* [2]. This is done because of the common Ruby pattern of using `'+""'` to **initialise a mutable string that will be used as a buffer**, which internally calls `Kernel#dup` on the string object.

[2] The latter condition was not enabled in the submitted code due to an error

3.5.6 Small string optimisation for ropes

Note: This optimisation should not be confused with the short/small-string optimisation often used in implementations of the `std::string` type in the C++ Standard Library.

Strings require more memory to be represented by trees of concatenation and substring ropes when compared to an array-based implementation. This is because of the need to allocate additional memory to represent the rope nodes. In practice, this overhead is often dwarfed by the speedup gained from the laziness property of ropes, which is why they perform so well [22].

The source code of the HotSpot JVM reveals that the object header of a Java object is at least 12 bytes long [2]. Assuming no padding anywhere else in the object, pointer compression, and standard sizes for the fields of the `ConcatRope` class, a `ConcatRope` object itself will require at least $12 + 8 * 4 = 46$ bytes.

This optimisation is based on the idea that for ropes representing strings with relatively few bytes, it might be more beneficial to eagerly compute the resulting byte array, instead of allocating additional rope nodes to represent operations on leaf nodes.

1. Use a partial-evaluation constant configurable as a command line option.
2. Apply a threshold to operations producing `SubstringRope` or `ConcatRope` with a number of bytes less than the threshold.
3. Eagerly evaluate the result and return a new `LeafRope` instead of a tree of rope nodes (figure 3.7).

The threshold amount is configurable via the `--small-string-size` command line option and defaults to 64 (see the comparison of different values in 4.8).

$$Concatenate(x,y) = \begin{cases} LeafRope(x||y) & \text{if } len(x)+len(y) \leq T \\ ConcatRope(x,y) & \text{if } len(x)+len(y) > T \end{cases}$$

$$Substring(x,s,n) = \begin{cases} LeafRope(x_s, x_{s+1}, \cdots, x_{s+n}) & \text{if } n \leq T \\ SubstringRope(x,s,n) & \text{if } n > T \end{cases}$$

Figure 3.7: Small string optimisation for concatenation and substring operations

3.5.7 Runtime detection of unsafe aliasing of mutable ropes

A runtime detection system for potentially-unsafe aliasing of mutable ropes is also implemeted. Its purpose is to detect programming errors stemming from the failure to call `GetReadOnlyRope` (the equivalent of the `RopegetShared` method) when a mutable leaf rope is used in a `RubyString` or a non-leaf `Rope` node. This runtime detection is enabled only when Java assertions[3] are en-

[3] Assertions are enabled with the `-ea` JVM option

abled, because of its associated overhead and because it complements the full set of sanity checks implemented as assertions in the TruffleRuby implementation.

The runtime detection system implemented in the `RopeSharingValidator` class maintains the set of all mutable ropes known to be referenced from either `RubyString` or a non-leaf `Rope` node. The class exposes a single boolean method, which establishes whether the reference `newReference` can be safely[4] used to substitute the (possibly null) reference `oldReference` (listing 3.9 for example usage):

```
boolean checkAttach(Rope newReference, Rope oldReference)
```

Unsafe sharing is detected by (1) removing `oldReference` from the set and (2) checking whether the current known set of mutable ropes contains `newReference`. The sharing is also trivially safe if `newReference` and `oldReference` reference the same object.

The set of mutable ropes is stored in a weakly-keyed hash set with identity-based key comparison[5]. This allows the mutable rope objects to be collected by the garbage collector.

```
1 class RubyString {
2   public void setRope(Rope rope) {
3     assert RopeSharingValidator.checkAttach(rope, this.rope);
4     this.rope = rope;
5   }
6   /* ... */
7 }
```

Listing 3.9: `RopeSharingValidator` for runtime detection of unsafe sharing

Effectiveness

This runtime detection system was able to detect one previously overlooked instance of unsafe aliasing occurring in the code base.

[4] Only read-only ropes are allowed to be referenced from multiple objects (see 3.4.3)

[5] For identity-based key comparison in `WeakHashMap`, the keys are wrapped in `IdentityKey` (own solution)

3.5.8 List of API changes

Truffle Node	Description	Section
`RopeReuseMonitorNode`	Profiles rope reuse and returns the rope reuse factor	3.5.2
`ProfileRopeMutationNode`	Produces an `OptimizationHint` using a `RopeReuseMonitorNode` folds to a partial evaluation constant after a certain threshold	3.5.3
`CommonCodeRangeNode`	Computes the most general code range for the arguments (PE-friendly)	
`MakeMutableLeafRopeNode`	Creates a new mutable leaf rope with the given byte array	
`IsBytesMutableNode`	Checks whether the rope is mutable leaf rope	
`GetMutableRopeNode`	Copies the argument if it is a read-only rope	
`GetReadOnlyRopeNode`	Calls `makeReadOnly()` on the argument if it is a mutable rope	3.5.4
`ElementAssignmentNode`	Call target for `String#[]=` which profiles the assignments using an ephemeral `ProfileRopeMutationNode` and delegates to `[Mutable]StringSpliceNode`	3.5.3
`StringSpliceNode`	Computes the result of the string assignment using the concatenation-based approach	3.5.3
`MutableStringSpliceNode`	Computes the result of the string assignment using the mutable rope approach	3.5.3
`MutableSetByteNode`	Modifies the backing array of the mutable rope	3.5.3
`AppendMutableNode`	Appends the contents of the source rope to the backing array of the mutable rope, allocating more memory if necessary and according to the resize policy	3.5.5
`WriteBytesMutableNode`	Writes the byte representation of the rope to the destination buffer without flattening the input rope	3.5.5
`BytesCopyNode`	Returns the byte array representation of the rope without flattening the input rope	3.5.5
`NewStringOrSelfNode`	Create mutable ropes when `Kernel#dup` is called on a mutable rope	3.5.5

Table 3.1: List of API changes in the Truffle implementation

Chapter 4

Testing and evaluation

This chapter presents a quantitative analysis of the optimisations implemented as part of this project and used methodology.

4.1 Choice of benchmarks

Several benchmarks were used in order to faithfully assess the performance improvement gained or lost when applying the presented optimisations.

4.1.1 Asciidoctor

'A fast text processor & publishing toolchain for converting AsciiDoc to HTML5, DocBook & more.' [23]

The benchmark *asciidoctor-convert* exercises the tasks of parsing and producing plain text documents using the 'asciidoctor' gem. Although the code exercised in this benchmark does not use the string mutation operations optimised in this report, this benchmark was used to detect regressions in other parts of the code. *asciidoctor-convert* is included in the TruffleRuby source repository.

4.1.2 ChunkyPNG filtered image decoding

ChunkyPNG is an image processing gem written in Ruby which has been downloaded more than 87 million times [28]. The gem is featured in an example in the user-facing documentation of TruffleRuby [21]. The specific example in mentioned in a GitHub issue. The author of the issue reports that TruffleRuby hangs during the execution of the ChunkyPNG example [21]. In the discussion of the issue, a member of the TruffleRuby development team mentions that similarly to Raytracer (section 3), this is caused by the code author of ChunkyPNG deciding to use a string to encode and process the image.

While a total of 13 benchmarks exercising various functionalities of ChunkyPNG are included in the TruffleRuby source repository, none exercise the code where the string encoding the image is modified via `String#setbyte` is performed (see 3.5.2 for a code example of image processing using strings).

ChunkyPNG needs to modify the string encoding the image whenever the PNG image uses a filter. Filtering is a technique used to improve the compressibility of the image [4, 26]. Filtering is extremely common and recommended for all types of images except colormapped

or grayscale images less than 8 bits deep [26]. Because of how common it is, it is most-likely that the 'hang' mentioned in the GitHub issue [21] is caused by the use of readily available PNG images which use filtering, thereby reaching the 'slow' string mutation implementation in the baseline revision of TruffleRuby.

The benchmark *chunky-decode-png-image-pass-filter* was created to specifically test the performance of filtered PNG decoding using ChunkyPNG.

4.1.3 CSV parsing

CSV parsing is the task of interpreting the values recorded in a comma-separated values (CSV) file. The benchmark *csv-parse* uses the 'csv' gem to parse a CSV file and was developed as part of this project.

4.1.4 Liquid templates

Liquid is an HTML templating engine used and written by Shopify. Two of the main tasks performed by Liquid are (1) parsing an HTML template and (2) producing an HTML document using the parsed template.

The TruffleRuby source repository includes benchmarks to test the performance of Ruby runtimes at both tasks: *liquid-cart-parse* and *liquid-cart-render*, respectively. The aim of the dynamic array optimisation (see 3.5.5) is to improve the performance on *liquid-cart-render* specifically.

4.1.5 Raytracer

As discussed in 3.1, the Raytracer benchmark was one of the main reasons why mutable string optimisation was targeted for optimisation and as such, it was an important part of this evaluation.

4.2 Evaluation difficulties

Truffle agressively optimises certain code paths based on a set of heuristics and assumptions. When these assumptions prove to be wrong, the runtime throws away the generated code, and falls back to interpretation. When the code path is determined to be 'hot' and thus likely benefit from JIT compilation, machine code is generated from the DAG used in the intermediate representation. This cycle repeats, until an optimal representation is reached.

This makes benchmarking difficult because the interpreter and JIT-produced code have different performance characteristics, and because the code generation itself is expensive and affects the results.

My original attempts at benchmarking TruffleRuby relied on the available tools and benchmarking harnesses with the default options. The default options resulted in a large standard deviation in the results ($>>5\%$) and thus needed manual fine-tuning.

4.3 Testing

The main form of testing used was the extensive RubySpec test suite, the MRI unit tests, and ecosystem tests, which verify that key Ruby applications are running properly. Defensive coding techniques in the form of runtime assertions were also used. Finally, a runtime verification system was developed specifically to detect unsafe sharing of mutable ropes (section 3.5.7).

4.4 Experimental setup

All experiments were run on a system with an AMD Ryzen 7 5800HS processor with 8 cores each at 3.19 GHz and 16 GB of RAM, running Fedora 35 with Linux kernel version 5.14. The baseline numbers reported were obtained using TruffleRuby e7746c0256 with GraalVM 21.3.0. The numbers reported as 'MRI' were obtained using MRI 2.7.6 (released on 12 April 2022) [1]. Peak performance was measured by running each benchmark in a loop for warmup until code generation stabilised, before recording the number of iterations executed. This process was repeated thrice for each benchmark, to account for the non-deterministic nature of the heuristic optimisations used in TruffleRuby and GraalVM [2]. The reported errors are the standard deviation.

4.5 Overall comparison with baseline

In order to make the results from these benchmarks easier to interpret, the numbers were linearly scaled such that the number corresponding to the performance of the baseline TruffleRuby implementation is always 1.

4.5.1 Mutable ropes

The mutable ropes optimisation (labelled 'mutable' in the charts) achieved all the implementation requirements set out in section 3.2. Any difference in results between it and the baseline TruffleRuby implementation is always positive or within the margin of error. The results on the *raytracer* and *chunky-decode-png-image-pass-filter* show an improvement of around **three orders of magnitude compared to the baseline implementation**, with the results being 840 (figure 4.4) and 1710 (figure 4.3) times higher, respectively.

4.5.2 Dynamic arrays

The dynamic arrays optimisation is an extension of the mutable ropes optimisation and is labelled 'mutable + dyn' in the charts. The optimisation resulted in **mixed results**. Some benchmarks such as *chunky-decode-png-image-filter* and *csv-parse* see a **considerable performance improvement**.

Special attention was given to the *liquid-cart-render* (figure 4.6) benchmark. This benchmark was the main target of this optimisation. Unfortunately, it sees a slight decrease compared to the baseline. Using a simple print debugging technique, I was able to verify that the output

[1] MRI 3.1.2 is also available, but it was not used because it targets Ruby ≥ 3.0

[2] For some benchmarks, the difference between runs could not be eliminated even by longer warmup

HTML string was indeed computed by *repeatedly growing the dynamic backing array of an initially empty mutable rope*.

Surprisingly, the *asciidoctor-convert* benchmark reports a 40% decrease in performance compares to the baseline.

4.5.3 Small string optimisation

Initial experiments on microbenchmarks showed that this optimisation can be beneficial.

Unfortunately, this optimisation **negatively affects** the results of *liquid-cart-parse* (figure 4.7) and *liquid-cart-render* (figure 4.8) regardless of the exact threshold value provided to `--small-string-size`.

One possible explanation is that the branching introduced by this optimisation makes it harder for GraalVM to perform some types of optimisations. I attempted to remedy this using a `BranchProfiler`, but this did not make a difference — likely because both branches were still being executed, so none could be pruned. In any case, this situation is non-ideal and is the reason why this optimisation was abandoned.

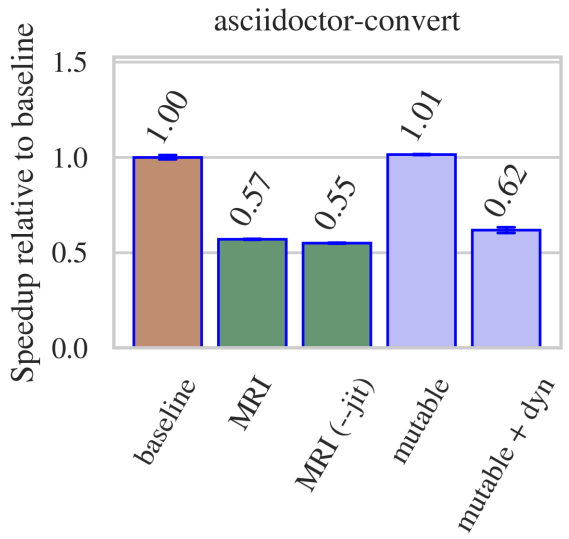

Figure 4.1: Comparison between Ruby runtimes on the *asciidoctor-convert* benchmark

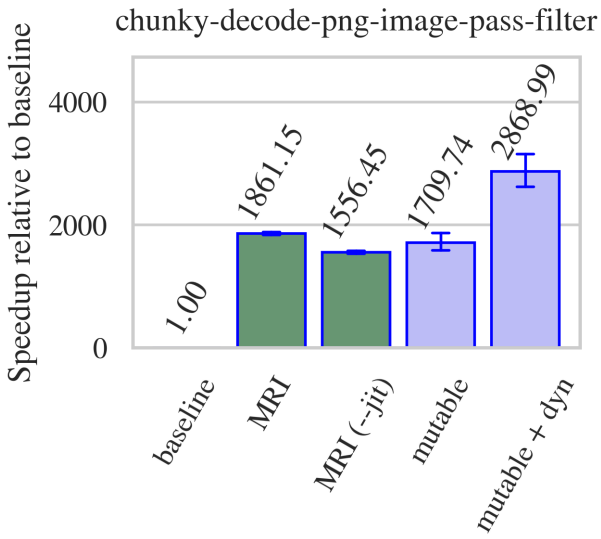

Figure 4.3: Comparison between Ruby runtimes on the *chunky-decode-png-image-pass-filter*

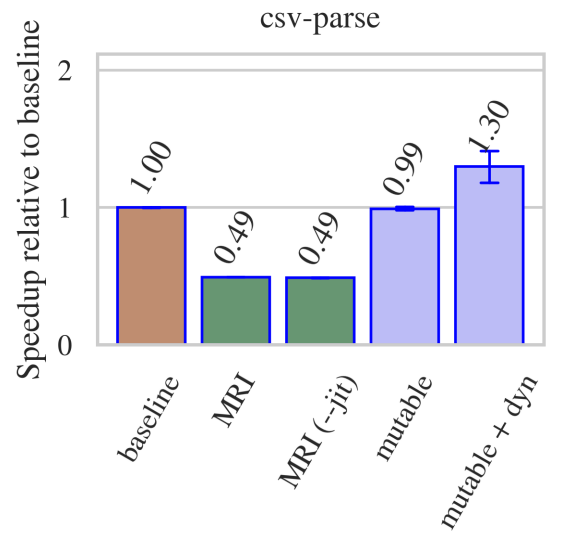

Figure 4.2: Comparison between Ruby runtimes on the *csv-parse* benchmark

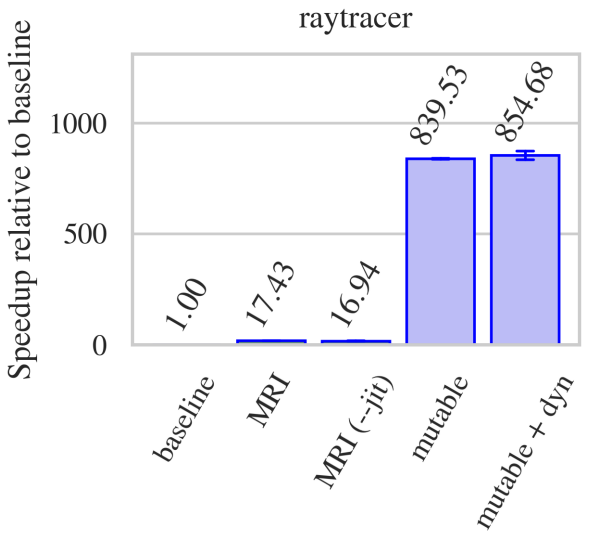

Figure 4.4: Comparison between Ruby runtimes on the *raytracer* benchmark

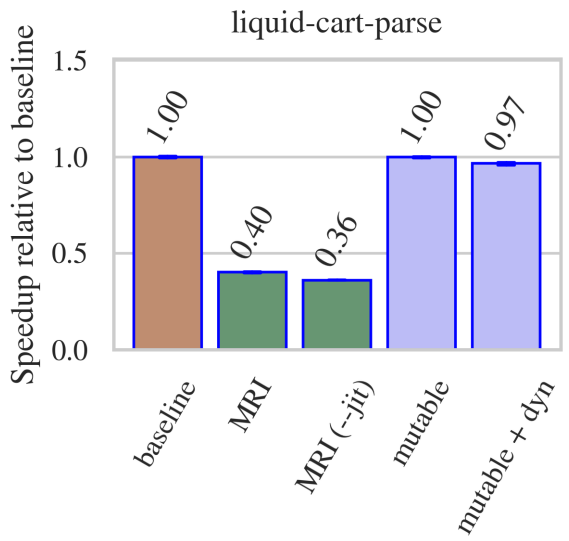

Figure 4.5: Comparison between Ruby runtimes on the *liquid-cart-parse* benchmark

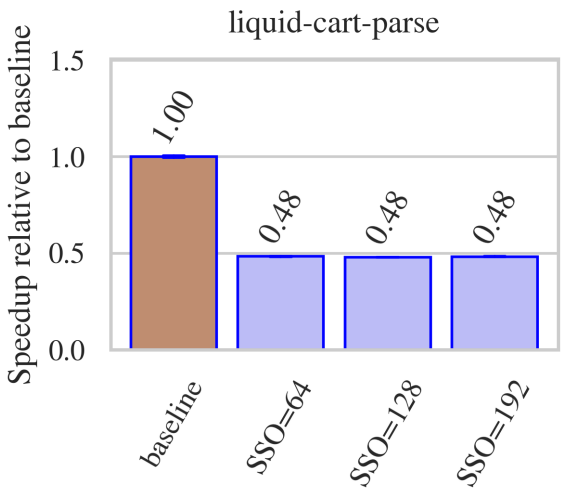

Figure 4.7: Comparison between different values of `--small-string-size` on the *liquid-cart-parse* benchmark

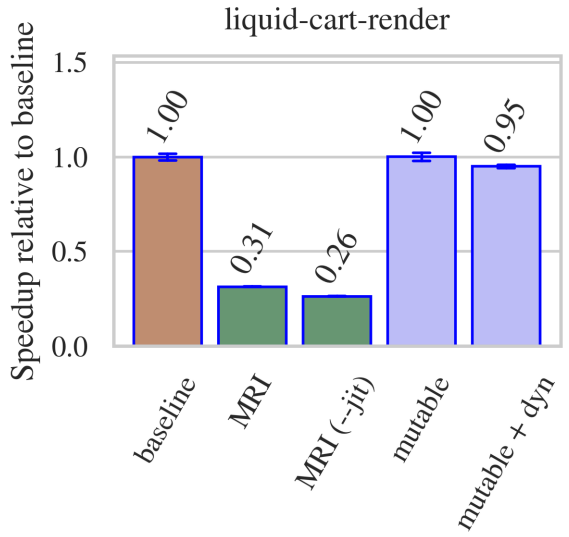

Figure 4.6: Comparison between Ruby runtimes on the *liquid-cart-render* benchmark

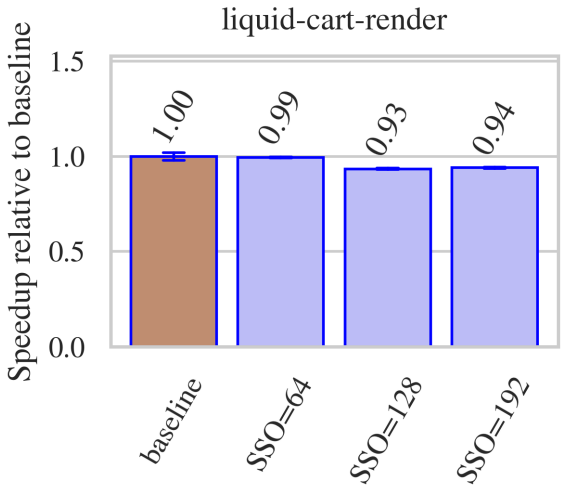

Figure 4.8: Comparison between different values of `--small-string-size` on the *liquid-cart-render* benchmark

Chapter 5

Conclusions

5.1 Summary

This paper surveys the most popular runtimes for the top 15 languages in current use and presents the main characteristics of their implementation of the string data type. It then takes an in-depth look at the non-standard tree-based string implementation in TruffleRuby and discovers that several reports of poor performance remain unresolved.

The core issue is identified to be related to inefficient representation of strings in certain scenarios. An improved design of the string data type in TruffleRuby is presented and implemented. The improvement by the new implementation is quantified by benchmarks and found to result in a close to 1,000x speedup compared to the baseline implementation in the reported use cases. The implementation is then further improved to make use of a new profiling technique, which ensures that no performance degradation is observed in scenarios in which the developed optimisations are not beneficial.

The new design introduces the possibility of improper use by TruffleRuby developers. To combat this, a runtime detection system which can detect when the new implementation is used improperly is also implemented and enabled only during development, with zero cost to the end user.

5.2 Future work

Given the limited amount of time available to complete this project, some interesting ideas for optimisation could not be fully explored.

5.2.1 Mutable ropes profiling

While the custom profiling technique developed as part of this project was successful, more evaluation is needed to find the best set of heuristic values for the thresholds used there. It is likely impossible to find a global optimal set of values, as a longer profiling period implies a longer warmup time.

A heuristic could be used is to initialise the counter values in such a way that no or little generated code needs to be pruned. One way of doing this could be by using the (earliest) compilation threshold value of the executing runtime as the initial value. Using appropriate low and high reuse thresholds, one can guarantee that at least in the cases of perfect reuse and

no reuse, a rope implementation will be selected (by rewriting `EphemeralRopeReuseProfile`) before code generation happens.

5.2.2 Dynamic arrays for mutable ropes

This implementation resulted in mixed results. There might be several reasons for that, but unfortunately, these could not be confirmed due to time constraints. One possible explanation for the lower performance on the *asciidoctor-convert* and *liquid-cart-render* benchmarks could be that the strings that are being appended to the mutable rope are themselves often represented not by leaf ropes, but by `ConcatRope` or `SubstringRope`. To be appended to the output, their contents is evaluated via `AppendMutableNode` (figure 3.1). This likely results in additional overhead which is not recouped by the laziness property of ropes.

Another possible explanation is the overhead brought on by the lazy truncation operation (section 3.5.5). Apart from requiring an additional copy of the underling array, the lazy truncation makes the `getBytes` operation more difficult to inline and introduces two new branches in that accessor method. As mentioned in the note at the end of section 3.5.5, the TruffleRuby team is currently working on a revision of TruffleRuby which will not require this lazy truncation. Once their work is completed, the code developed as part of this project will have to be ported and the dynamic arrays implementation will have to be re-evaluated.

Both of these explanations are supported by the large standard deviation in the results — it is likely that de-optimisation loops are occurring.

Finally, the use of dynamic arrays can also be used to implement the `String#[]=` with mutable ropes even when the lengths of the replaced substring and the new substring differ (section 3.4). This optimisation has not been explored due to time constraints.

5.3 Reflection

A considerable amount of time was spent first on researching dynamic language runtimes and learning about their implementation. The variety is overwhelming: AST interpretation, IR interpretation, AOT compilation, JIT compilation, etc. TruffleRuby was suggested to me by my project supervisor, Mikel Luján. I liked TruffleRuby because its architecture was somewhat different from the other runtimes I had previously reviewed. It was also relatively new and actively developed, so there certainly were areas for improvement.

After selecting the dynamic language runtime to use, another period of research began, as an area for improvement needed to be selected. After several weeks of research and learning about TruffleRuby, mostly through published literature and from Chris Seaton's blog, I stumbled upon the paper by Menard et al. about the use of ropes to represent strings. Being familiar with a few string implementations and having only ever heard of ropes being used in specific situations (text editors), I found the idea of using ropes as a general-purpose implementation curious. After some experimentation and reading through performance issues on the TruffleRuby issue tracker, I stumbled upon the issue about Raytracer [29]. That solidified the main objective of this project.

The evaluation difficulties described in section 4.2 were problematic for me on several occasions. Initial results made me think I have had a breakthrough, when in fact, the results were skewed and were not reproducible. This is something that could have been remedied by repeating each experiment several times and by cautious use of microbenchmarks.

Bibliography

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[2] Oracle and/or its affiliates. HotSpot JDK 8 source code. http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/87ee5ee27509/src/share/vm/oops/markOop.hpp, 2014. [Online; accessed 27-April-2022].

[3] Thomas Bøgholm, René R Hansen, Anders P Ravn, Bent Thomsen, and Hans Søndergaard. Schedulability analysis for java finalizers. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 1–7, 2010.

[4] Thomas Boutell. Png (portable network graphics) specification version 1.0. Technical report, 1997.

[5] Ruby community. Ruby programming language. https://www.ruby-lang.org/en/, 2022. [Online; accessed 29-April-2022].

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[7] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. What's wrong with my benchmark results? studying bad practices in JMH benchmarks. *IEEE Transactions on Software Engineering*, 47(7):1452–1467, 2019.

[8] Edin Omeragic (edin). Raytracer. https://github.com/edin/raytracer, 2015. [Online; accessed 21-April-2022].

[9] Barry A. Feigenbaum. *Go for Java programmers : learn the Google Go programming language*. Apress, California, 2022. ISBN 9781484271995.

[10] Jeff "JavaJeff" Friesen. *Learn Java for Android Development*. Springer, 2010.

[11] Jacek Galowicz. *C++ 17 STL Cookbook*. Packt Publishing Ltd, 2017.

[12] Alexander Gofen. From Pascal to Delphi to Object Pascal-2000. *SIGPLAN Not.*, 36 (6):38–49, jun 2001. ISSN 0362-1340. doi: 10.1145/504359.504363. URL https://doi.org/10.1145/504359.504363.

[13] Poonam Goyal. Comparative study of C, Java, C# and Jython. 2014.

[14] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[15] Dave Grundgeiger. *Programming Visual Basic. NET*. O'Reilly, 2018.

[16] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# programming language*. Pearson Education, 2008.

[17] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European conference on object-oriented programming*, pages 21–38. Springer, 1991.

[18] Michael R Jantz and Prasad A Kulkarni. Exploring single and multilevel JIT compilation policy for modern machines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):1–29, 2013.

[19] Robert Kerr. *Beginning Swift : master the fundamentals of programming in Swift 4*. Packt, Birmingham ;, 2018. ISBN 9781789538649.

[20] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–32, 2008.

[21] Mateusz Czerwiński (mc2dx). Problem with installing stackprof with native extensions. https://github.com/oracle/truffleruby/issues/2044, 2020. [Online; accessed 21-April-2022].

[22] Kevin Menard, Chris Seaton, and Benoit Daloze. Specializing ropes for ruby. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 1–7, 2018.

[23] Asciidoctor Project. Asciidoctor. https://asciidoctor.org/, 2022. [Online; accessed 27-April-2022].

[24] Luciano Ramalho. *Fluent Python: Clear, concise, and effective programming*. " O'Reilly Media, Inc.", 2015.

[25] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[26] Greg Roelofs. Chapter 9. compression and filtering. *PNG The Definitive Guide. San Jose, CA: O'Reilly & Associates*, 1999.

[27] Akihiko Tozawa, Michiaki Tatsubori, Tamiya Onodera, and Yasuhiko Minamide. Copy-on-write in the PHP language. *SIGPLAN Not.*, 44(1):200–212, jan 2009. ISSN 0362-1340. doi: 10.1145/1594834.1480908. URL https://doi.org/10.1145/1594834.1480908.

[28] Willem van Bergen. Chunkypng. https://rubygems.org/gems/chunky_png/versions/1.3.5, 2022. [Online; accessed 27-April-2022].

[29] Vlad Zarakovsky (vlazar). Raytracer benchmark is 34x slower than with MRI. https://github.com/oracle/truffleruby/issues/2336, 2021. [Online; accessed 21-April-2022].

[30] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14, 2012.

[31] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–676, 2017.

Glossary

aliasing Aliasing refers to a situation in which a data location can be accessed using different names in the program. 20, 24

API (Application Programming Interface) The software interface through which two or more pieces of software can communicate. 2, 3, 5, 23, 35

ASCII (American Standard Code for Information Interchange) A fixed-width 7-bit character encoding. 49

C A low-level general-purpose programming language that is often used for system programming or high-performance applications. 11

C++ A high-level general-purpose programming language that is often used for system programming. 11

C# A high-level general-purpose programming language that runs on the open-source .NET runtime (similar to JVM). 11

code point A numerical value that can be mapped to a specific character. 11, 23

Delphi A proprietary dialect of Object Pascal. 11

domain-specific programming language Domain-specific programming languages are programming language specialised for a specific problem domain. 11

dynamic array (also known as resizable array or array list) A dynamic array is data structure with variable-length which stores its elements contiguously in memory and allows elements to be added or removed. 31

finalizer Finalization is a deprecated feature which allows arbitrary code to be executed after the GC has found an object to be unreachable. 21

gem A Ruby gem is a Ruby-specific format of packaging source code and other files for reuse. 36, 37

Go A high-level general-purpose programming language developed by Google. 11

HTML templating HTML templating refers to the process of building strings of text representing an HTML document with some user-defined tokens replaced with values from the source program. 17, 18, 25, 37

Java A high-level class-based general-purpose programming language that runs on the JVM. 11

JIT (Just-in-time compilation) A technique for delaying machine code generation until the execution of the program. 12–17

JVM (Java Virtual Machine) A piece of software that enables a computer to execute Java bytecode. 12, 14, 15, 33

Latin-1 (ISO/IEC 8859-1:1998) An 8-bit fixed-width ASCII-compatible character encoding. 11

LLVM (Low-Level Virtual Machine) An open-source compiler toolchain. 14

method lookup Method lookup refers to the algorithm used by the Ruby language when resolving the call target of a method call. 16

null-terminated A string is null-terminated when exactly and only the character in the string is the ASCII NUL character. 11

object identity Object identity is a concept in which refers to the ability to distinguish individual objects even when they are structurally equal. 16

object identity hash code An integer value which is selected non-deterministically and is guaranteed not to change for the lifetime of the object. 28

Object Pascal A high-level general-purpose programming language a popular dialect of which is Delphi. 11

PHP A high-level general-purpose dynamic programming language that is most often used for web development and is often interpreted. 11

pointer compression Pointer compression is a memory-saving technique used in some JVM implementations to represent object references as 32-bit integers when the executing JVM is compiled for a 64-bit processor. 33

Python A high-level general-purpose dynamic programming language that is often interpreted. 11

ray tracing Ray tracing is an image rendering technique which involves sampling light reflected from objects. 19

stack trace A stack trace is a record of the active stack frames at a point in time of the execution of a program. 17

Swift A high-level general-purpose programming language developed by Apple. 11

TIOBE index ('The Importance Of Being Earnest' Index) An indicator of the popularity of programming languages that is updated monthly. 5, 11

Unicode Standard The Unicode Standard defines the representation and coding of characters. 49

UTF-16 A variable-width character encoding defined by the Unicode Standard in which the basic code unit is 16-bits long code points, and which uses either one or two code units for code point encoding. 11

UTF-32 A fixed-width character encoding defined by the Unicode Standard. 11

UTF-8 An ASCII-compatible variable-width character encoding defined by the Unicode Standard in which the basic code unit is 8-bits long and which uses between one and four code units for code point encoding. 11

Visual Basic A high-level general-purpose programming language that runs on the open-source .NET runtime (similar to JVM). 11